UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Master's Thesis

Submitted to the Secure Software Engineering Research Group
in Partial Fulfilment of the Requirements for the Degree of

Master of Science

LLMs for Call Graph Construction: Benchmarking and Evaluation Across Languages

by
Rose Sunil

Thesis Supervisors:
Prof. Dr. Eric Bodden
and
Jun.-Prof. Dr. Mohamed Aboubakr Mohamed Soliman

Paderborn, December 14, 2024

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Paderborn, 14.12.2024	[illegible]
Ort, Datum	Unterschrift

Abstract. Large Language Models (LLMs) have immense capabilities in understanding and generating language. They are becoming increasingly powerful and hold significant potential in software engineering, where their applications are being actively explored and investigated. Within the field of software engineering, static analysis involves evaluating a program without executing its source code, enabling early detection of bugs and vulnerabilities.

In this study, we focus on evaluating the performance of LLMs in static code analysis tasks, particularly call graph construction for Python, JavaScript, and Java programs. We assessed 26 LLMs, including OpenAI's GPT series and open-source models such as LLaMA and Mistral, using both existing and newly designed micro-benchmarks. As part of this study, we introduced SWARM-CG, a comprehensive benchmarking suite aimed at evaluating call graph construction tools across multiple programming languages including Python, JavaScript, and Java. SWARM-CG facilitates cross-language comparisons and consistent analysis. Additionally, we developed SWARM-JS, a specialized micro-benchmark tailored for JavaScript call graph analysis tasks. The performance of LLMs was also systematically compared with traditional static analysis approaches, highlighting their relative strengths and limitations.

The results of this study reveal that, in Python, traditional tools like PyCG significantly outperform LLMs. For JavaScript, the static analysis tool Jelly surpasses LLMs in soundness, while another tool, TAJS, underperforms due to its limited support for modern language features. Interestingly, LLMs achieve considerably better results in Python compared to JavaScript, where several models produce weak outputs. In the case of Java, the static analysis tool SootUp falls short due to its limited support for the dynamic language features present in the CATS benchmark used for Java evaluations. However, LLMs demonstrate strong performance in Java. These findings highlight the potential of LLMs to assist static code analysis tasks, while also underscoring their current limitations in call graph construction. This study establishes a foundation for integrating LLMs into static analysis workflows and advocates for further research into their optimization and broader applications.

Contents

1 Introduction

With the advancements in artificial intelligence and natural language processing, there has been a lot of research regarding the application of Large Language Models (LLMs) in Software Engineering (Hou et al., 2024; Li et al., 2023a; Pearce et al., 2023; Wang et al., 2023; Song et al., 2023). LLMs are increasingly powerful neural network models trained on large datasets of text and code, allowing them to learn the underlying patterns in data and produce human-like responses across a variety of tasks (Ozkaya, 2023). With applications ranging from natural language processing and translation to code generation and software engineering, LLMs have great potential across various fields.

Several studies investigate the role LLMs can play in Static Code Analysis (SA) which is a fundamental aspect of Software Engineering (SE) (Li et al., 2023a; Ozturk et al., 2023; Fan et al., 2023; Ziems and Wu, 2021). SA focuses on evaluating code without executing it, allowing developers to detect potential errors, maintain code quality, and identify security vulnerabilities early in the development lifecycle. Historically, SA tools have faced challenges, such as the high rates of false positives, the difficulty of scaling to large codebases, and the limited ability to handle ambiguous or incomplete code. Recent works have shown how different SA tasks can benefit from LLMs, such as static bug detection and false-positive warning removal (Mohajer et al., 2024), precise function summaries generation (Li et al., 2023a), type annotation (Seidel et al., 2023), and general enhancements in precision and scalability of SA tasks (Li et al., 2023b), both fundamental issues of SA.

Goal: This thesis is dedicated to examining the effectiveness of LLMs in SA within SE. It aims to **evaluate the accuracy of LLMs in performing static call graph analysis tasks in Python, JavaScript, and Java programs**. *Call-graph analysis* helps understand the relationships and interactions between different components of a program.

Methodology: This thesis conducts a comprehensive analysis of 26 different LLMs on call graph analysis tasks, using data from micro-benchmarks and customized prompts. This allows direct comparison with traditional approaches in static analysis. To assess the performance of LLMs, the PyCG (Salis et al., 2021) and Jarvis (Huang et al., 2024) micro-benchmarks are used for call-graph analysis in Python and CATS (Reif et al., 2019; Reif, 2021) micro-benchmark is used for Java. A newly created *SWARM-JS* micro-benchmark is used for call-graph analysis in JavaScript. The use of micro-benchmarks in evaluating the performance of LLMs in this study is grounded in the following key considerations:

- Micro-benchmarks are designed to target specific aspects of the features under test and various characteristics of the programming language involved. This helps highlight the models' strengths and weaknesses, allowing for a more nuanced understanding of their capabilities in SA tasks. In this way, micro-benchmarks offer a systematic way to understand feature-specific limitations and identify areas for improvement.

- Micro-benchmarks development involves rigorous manual inspection and adherence to scientific methods, ensuring reliability and accuracy in evaluation. Benchmarks that provide ground truths allow for precise comparisons of precision and soundness (Mordahl, 2023).
- This foundational approach is essential to understanding LLMs' basic limitations and capabilities before progressing to tests involving complex, large-scale code bases. Conversely, obtaining large-scale, real-world data that can serve as ground truth is often a challenging endeavor. Where such data is available, it is susceptible to human errors, which can skew the results.

The insights from this thesis are intended to offer a preliminary understanding of the role of LLMs in SA for call-graph construction tasks, contributing to the Artificial Intelligence for Software Engineering (AI4SE) and Software Engineering for Artificial Intelligence (SE4AI) fields.

Results: The results of this study show that for Python, static analysis tools like PyCG significantly outperform LLMs in call-graph generation. In JavaScript, static analysis tools such as Jelly outperformed LLMs in terms of soundness. TAJS, another static analysis tool for JavaScript, underperformed due to its inability to handle modern language features introduced in ECMAScript 6. For Java, LLMs showed notable improvements, with several models achieving high soundness rates. However, SootUp, a static analysis tool for Java, exhibited relatively lower performance due to its limitations in handling dynamic language features present in the micro-benchmark used for evaluation.

Contributions: The primary contributions of this thesis are as follows:

- Performed an empirical evaluation of 26 LLMs across Python, Java, and JavaScript for call-graph inference.
- Introduced SWARM-CG, a comprehensive benchmarking suite for evaluating call-graph construction tools across multiple programming languages, starting with Python, Java, and JavaScript, to enable cross-language comparisons and consistent analysis evaluations.
- Developed SWARM-JS, a call-graph micro-benchmark for JavaScript.
- Compared LLM performance with existing traditional approaches in SA.

Structure: This thesis is structured as follows:

In Chapter 2, we provide background information, including (a) an overview of static analysis and call graphs, (b) a description of existing call graph construction tools for Python, JavaScript, and Java, and (c) a brief discussion of large language models (LLMs). Chapter 3 reviews relevant research, focusing on exploring the application of LLMs in software engineering and various static analysis tasks.

The research questions that guide this thesis are outlined in Chapter 4. In Chapter 5, we detail the micro-benchmarks used for evaluations in this study. Chapter 6 presents the methodology of this study, covering: (a) LLM model selection, (b) prompt design, (c) evaluation metrics, and (d) implementation details. The results of the evaluation are presented in Chapter 7, which addresses each research question in detail. These findings are then discussed in Chapter 8. Chapter 9 discloses the potential threats to the validity of our study. Finally, the thesis concludes with a summary of findings and outlines directions for future research in Chapter 10.

Availability:

- SWARM-CG is published on GitHub as open-source software: `https://github.com/secure-software-engineering/SWARM-CG`

2 Background

2.1 Static Analysis and Call Graph

Static Analysis. Static Analysis, an essential aspect of SE is the method of debugging a program without executing its source code (Novak et al., 2010). It is used for compiler optimization, coding support, and for the detection of bugs and vulnerabilities. Since SA can be performed without program execution, it enables early detection of bugs and vulnerabilities. Unlike dynamic analysis, which requires running the code, static analysis tools can examine a program's structure with less effort and may be applied early in the development process, even before the program is complete enough for meaningful testing (Chess and McGraw, 2004).

Call Graph. A Call Graph is an essential data structure for conducting SA. It consists of nodes representing program methods and edges representing calling relationships between the methods. The primary goal of constructing a call graph is to determine the call edge relation. A call edge links a call site, which may be an instruction within a method, to the method that can potentially be invoked from that call site (Ali and Lhoták, 2012). Call graph construction plays an important role in various SA tasks, including malware detection (Hu et al., 2009), SA for security vulnerabilities (Chess and McGraw, 2004; Livshits and Lam, 2005), and software fault diagnosis (Chen et al., 2020).

In the following Java code, the `main` method creates an instance of the `Main` class at line 3 and `Car c` is instantiated using the nested class constructor `m.new Car()` at line 4. The `manufacture()` method is then called on the `Car` object `c` at line 5.

```
1  public class Main {
2      public static void main(String[] args) {
3          Main m = new Main();
4          Car c = m.new Car();
5          c.manufacture();
6      }
7
8      public class Car {
9          public void manufacture() {
10             System.out.println("Manufacture me!");
11         }
12     }
13 }
```

The complete call graph for the Java code snippet is as follows:

main → *main.<init>*
main → *car.<init>*
main → *car.manufacture()*

2.2 Call Graph Construction Tools

This section briefly describes the existing call graph construction approaches across multiple programming languages, specifically Python, JavaScript, and Java. First, we discuss PyCG (Salis et al., 2021), a call graph construction technique for Python. Next, we introduce Jelly (Laursen et al., 2024) and TAJS (Jensen et al., 2009), both of which are static tools designed for JavaScript. Finally, we discuss SootUp (Karakaya et al., 2024) for Java, which is a reimplementation of the Soot (Vallée-Rai et al., 2000) static analysis framework.

2.2.1 PyCG

PyCG (Salis et al., 2021) is a static call-graph construction technique for Python. PyCG conducts a flow-insensitive analysis and works in two steps. Initially, it computes an *assignment graph* which is a structure that shows relations between program identifiers such as functions, variables, classes, and modules, through an inter-procedural analysis. Next, the call graph of the given Python program is constructed by analyzing the assignment graph.

The tool is capable of handling complex Python features such as modules, generators, lambdas, and multiple inheritance. It is evaluated on both micro-benchmarks and real-world Python packages. The PyCG micro-benchmark is a test suite of minimal Python programs organized into distinct feature categories. In this study, we are utilizing PyCG micro-benchmark for the evaluation of call graph tasks in Python. Details of the micro-benchmark are discussed in Chapter 5.1. PyCG exhibits strong performance in terms of precision and recall, achieving a precision of 99.2% and a recall rate of around 69.9%.

2.2.2 Jelly

Jelly (Laursen et al., 2024) is a hybrid approach combining static and dynamic analysis to improve JavaScript analysis accuracy. The approach involves two steps. The first step is to perform a dynamic *pre-analysis* which is referred to as *approximate interpretation.* It is a form of forced execution that generates *hints* about the possible values involved and dynamic property accesses. The information produced in this step is then used by the main static call graph analysis to improve the soundness of the generated call graphs.

Jelly supports the latest ECMAScript language features, and can handle multi-file test cases as input. Depending on the options enabled, it generates a call graph both in JSON format and for HTML visualization. The *–approx* option may be used to enable static analysis with approximate interpretation. The Jelly analyzer was found to be more accurate for real-world JavaScript programs than other existing static analysis tools for JavaScript such as TAJS (Jensen et al., 2009) and ACG (Feldthaus et al., 2013).

2.2.3 TAJS

TAJS (Jensen et al., 2009) is a static analysis tool designed for JavaScript that performs type inference and constructs call graphs. It fully supports ECMAScript 3rd edition and provides partial support for ECMAScript 5, including its standard library, HTML DOM, and browser APIs. However, TAJS does not support features introduced in ECMAScript 6 (ECMA, 2015), such as classes, arrow functions, and modules, which limits its effectiveness in analyzing modern JavaScript applications. TAJS offers a command-line option to export these call graphs as DOT files, which can be converted into JSON for further analysis or integration with other tools.

2.2.4 SootUp

SootUp (Karakaya et al., 2024) for Java is a complete reimplementation of the widely used Soot (Vallée-Rai et al., 2000) static analysis framework. SootUp is designed as a modular library, with its components separated into independent modules. This modular architecture enables its clients to include only the necessary functionality in their applications. This not only improves performance but also reduces memory usage, making it suitable for analyzing large-scale applications. Additionally, SootUp's modularity allows it to be extended to support programming languages beyond Java, making it a versatile framework for multi-language static analysis.

The *core module* of SootUp contains the core building blocks such as the Jimple IR, control flow graphs, and frontend interfaces. The rest of the modules are built on the core module. The *callgraph module* contains implementations of common call graph construction algorithms such as *Class Hierarchy Analysis* (CHA) and *Rapid Type Analysis* (RTA).

2.3 Large Language Models

Language models are computational models that can understand and generate human language (Chang et al., 2023). Studies have shown that scaling Pre-trained Language Models (PLMs) by increasing their size or using larger training datasets often results in better performance across various tasks. These larger PLMs were found to display behaviors that differ from smaller models, enabling them to handle more complex tasks effectively (Hou et al., 2023). This distinction led to the introduction of the term *Large Language Models* (LLMs) (Zhao et al., 2023; Hou et al., 2023) to refer to these larger models. Hou et al. (2023), in their paper categorizes the LLMs they investigated into three groups: encoder-only, encoder-decoder, and decoder-only LLMs based on their architectures. The decoder-only models are the most recent type among encoder, encoder-decoder, and decoder models. They use a decoder module to generate the target output text, relying heavily on the model's ability to understand language structure, syntax, and context (Hou et al., 2023). Models such as the GPT series models (OpenAI, 2022), Llama (Touvron et al., 2023), Llama 2, and Bard (Google, 2023) are a few examples of decoder-only implementations.

Due to their remarkable capabilities, there has been significant research into the application of LLMs in Software Engineering (Koide et al., 2024; Zan et al., 2022; Lajkó et al., 2022; Sridhara et al., 2023). Studies indicate that within SE, LLMs are primarily used in the software development domain, especially in areas like coding and development (Hou et al., 2023). They have also been utilized in software maintenance, quality assurance, design, software management, and requirements engineering domains.

Fine-tuning is one of the most widely used approaches for LLM optimization. It involves adapting models to perform specific tasks and training on various datasets to enhance their performance. Fine-tuning LLMs using a variety of datasets that describe tasks in natural language is called instruction tuning (Zhao et al., 2023). With instruction tuning, the models can understand and respond to new tasks, even if they haven't seen those specific tasks before.

Prompt engineering is another prominent approach to improving the performance of LLMs on specific tasks. Studies show that well-designed prompts can significantly enhance results across various tasks(Liu et al., 2021, 2023; White et al., 2023; Liu and Chilton, 2021). It is an iterative process that involves creating customized prompts to effectively guide LLMs in performing particular tasks (Hou et al., 2023). Schulhoff et al. (2024), in their study presents a taxonomy of existing prompt engineering techniques and examines over 200 different methods.

Evaluating the performance of LLMs and assessing the reliability of their outputs is an important aspect of their development and use in various software engineering tasks (Kang et al., 2023). To evaluate the performance and effectiveness of LLMs, it is common practice to compare them against existing datasets and established baselines (Hou et al., 2023). By doing so, we can measure how well LLMs perform on specific tasks compared to previous models or benchmark standards. This process helps quantify improvements, identify strengths and weaknesses, and provide a clearer understanding of a model's capabilities.

3 Related Work

In this chapter, we discuss the role of LLMs in software engineering, as well as their application to various static code analysis tasks such as call graph construction, bug detection, and code summarization.

LLMs for Software Engineering. Xia and Zhang (2024) propose an automated program repair approach called ChatRepair, which leverages ChatGPT to perform code repairs. Their evaluations demonstrate that this LLM-based tool achieves state-of-the-art performance on the Defects4J dataset. Hey et al. (2020) present NoRBERT, an approach based on the language model BERT for requirements classification. The approach achieves F1-scores of up to 94% for classifying functional and non-functional requirements and outperforms recent methods in subclassifying non-functional requirements. The results of their study demonstrate NoRBERT's ability to improve requirements classification, even in unseen project settings. Pearce et al. (2023) examined the effectiveness of LLMs in addressing software vulnerabilities and generating zero-shot fixes. The results of their study indicate that while models are capable of producing accurate fixes in simple scenarios, they struggle when dealing with real-world scenarios. Pei et al. (2024) introduce SYMC, a new approach to improve how LLMs learn code semantics for program analysis tasks. SYMC modifies Transformer architectures to naturally incorporate code symmetries, enabling the model to focus on semantic structure rather than syntax. Lemieux et al. (2023) present CODAMOSA, a hybrid approach that combines Search-Based Software Testing (SBST) with Large Language Models (LLMs) like OpenAI's Codex. CODAMOSA utilizes LLMs to generate additional test cases for under-covered functions, enabling SBST to explore more effectively. Evaluations show that CODAMOSA achieves significantly higher coverage on numerous benchmarks compared to SBST and LLM-only methods, demonstrating the potential of integrating LLMs with traditional testing techniques.

LLMs for Vulnerability Detection. Sun et al. (2024) introduces GPTScan, a tool combining GPT with static analysis for detecting logic vulnerabilities in smart contracts. Traditional tools struggle with detecting 80% of Web3 security bugs due to their lack of domain-specific vulnerability descriptions. GPTScan utilizes GPT's ability to understand and reason about code. Evaluations show that GPTScan identifies ground-truth logic vulnerabilities with a recall of over 70%. Khare et al. (2024) explores the potential of LLMs for detecting security vulnerabilities across diverse code samples. Evaluating 16 pre-trained LLMs on 5,000 code samples from five security datasets, the study finds that while LLMs show modest overall effectiveness, with an average accuracy of 62.8% and F1 score of 0.71, they excel in detecting vulnerabilities requiring intra-procedural reasoning, such as OS Command Injection and NULL Pointer Dereference, outperforming popular static analysis tools like CodeQL. The study highlights the impact of advanced prompting strategies on improving LLM performance, yielding up to an 18% improvement in F1 score on real-world datasets. These findings suggest that LLMs can effectively perform parts of vulnerability detection, such as identifying vulnerability-related specifications and understanding code behavior, which could inspire future developments in LLM-assisted vulnerability detection systems. Li et al. (2024) proposes LLift, an automated framework for enhanced bug detection using LLMs. The approach gives promising results and underscores the potential of integrating LLMs with SA tasks.

LLMs for SA tasks. The paper by Sun et al. (2023) investigates ChatGPT's effectiveness in automatic code summarization for Python programs, comparing it with state-of-the-art models like NCS, Code-BERT, and CodeT5. Despite ChatGPT's promising potential, the results show that it underperforms in code summarization compared to these models. The authors identify key challenges, including the need for improved prompt engineering and evaluation metrics in future research. Ma et al. (2024) evaluated LLMs' proficiency in syntax and semantic comprehension for SE tasks. The study evaluated LLM models (GPT-4, GPT-3.5, StarCoder, CodeLlama) in Python, Java, C, and Solidity. Their findings indicate that the models do well at syntax comprehension and basic static analysis but struggle to understand dynamic behaviors. Mohajer et al. (2023) introduce an LLM-based tool to detect specific vulnerabilities in the source code specifically null dereferencing and resource leaks. The tool is also capable of false-positive warning removal and bug repair tasks. The static analyzer Infer (Havelund et al., 2015) is used to detect errors in code and generate warnings. The study focuses on two models from OpenAI, ChatGPT-4 and ChatGPT-3.5 Turbo, and is limited to Java projects. The results show that the LLM-based tool achieves up to 12.86% and 43.13% higher precision compared to Infer in detecting null dereference and resource leak bugs. Li et al. (2023a) explores the use of LLMs, particularly GPT-3.5 and GPT-4, to assist static analysis tools in reducing false positives and false negatives. Their approach focuses on generating more precise function summaries by leveraging LLMs.

Venkatesh et al. (2024b) in their work provides a comprehensive evaluation of 26 LLMs, focusing on call graph analysis and type inference for Python programs. Their findings indicate that while LLMs surpass traditional methods in type inference accuracy, their performance in callgraph analysis is limited. This underscores the need for task-specific fine-tuning and additional research across programming languages. Based on these findings, this thesis focuses on call graph analysis across multiple programming languages (Python, Java, JavaScript), broadening the scope to cross-language evaluations and benchmarking LLMs against existing static analysis tools.

4 Research Questions

Call graphs are a vital component of static analysis. It provides insights into program behavior and the relationships between its components. While traditional static analysis tools have been instrumental in this domain, they face challenges with precision, scalability, and high false-positive rates (Li et al., 2023a; Johnson et al., 2013; Park et al., 2022). Recent advances in LLMs present an opportunity to examine their potential in the field of static analysis. The research questions of this thesis are intended to bridge this gap, providing a comprehensive understanding of how well LLMs perform call graph analysis on micro-benchmarks and how their performances vary across languages. This thesis focuses on answering the following research questions:

RQ1: *What is the accuracy of LLMs in performing callgraph analysis against micro-benchmarks?*

RQ2: *How does the accuracy of LLMs vary across different languages?*

RQ3: *How does the performance of LLMs compare to that of existing static analysis tools?*

To address these research questions, we use language-specific micro-benchmarks which provide a structured way to test the capabilities of LLMs. The scope of this study is Python, JavaScript, and Java. For Python, we use the PyCG micro-benchmark, for JavaScript we use SWARM-JS and for Java, we use the CATS micro-benchmark In the following chapter, we discuss each of these micro-benchmarks in detail.

5 Micro-benchmarks

In this thesis, we evaluate the performance of LLMs and static analysis tools for call graph generation using micro-benchmarks tailored for each language.

To answer RQ1 and RQ2, the PyCG (Salis et al., 2021) and Jarvis (Huang et al., 2024) micro-benchmarks are used for call-graph analysis in Python and CATS (Reif et al., 2019; Reif, 2021) micro-benchmark is used for Java. A newly created *SWARM-JS* micro-benchmark is used for call-graph analysis in JavaScript. To answer RQ3 we choose the call graph construction tools discussed in Section 2.2: PyCG for Python, Jelly and TAJS for JavaScript, and SootUp for Java. The following sections detail the micro-benchmarks used in this thesis.

5.1 PyCG: Call-graph Micro-Benchmark

The PyCG micro-benchmark suite is a test suite for benchmarking call graph generation in Python (Salis et al., 2021). It is designed to evaluate and compare various call graph generation approaches in Python. PyCG consists of 112 minimal and focused test cases categorized into distinct feature areas including classes, decorators, imports, and generators. Each category contains a number of test cases and each test case comprises of:

(1) source code
(2) call graph in JSON format
(3) short description

The ground-truth call graphs are in a simple JSON format, detailing the relationships between functions as adjacency lists. An edge (`src`, `dst`) is represented as an entry of `dst` in the list assigned to the key `src`:

```json
{
    "srcA": ["dstB", "dstC"],
    "srcB": ["dstD"],
    "srcC": []
}
```

The tests are structured to be easy to categorize and expand, with each focusing on a single execution path, without the use of conditionals or loops. This design ensures that the generated call graph accurately reflects the execution of the source code. To validate the micro-benchmark suite, PyCG was reviewed by two researchers who have professional experience as Python developers. The test suite was assessed based on its completeness, code quality, and description clarity. Based on the feedback received, the micro-benchmark was refactored and enhanced.

Additional tests were included to address missing features, such as built-in functions and generators.

In this thesis, we use an extended version of the PyCG benchmark by adding new test cases from Jarvis (Huang et al., 2024). The Jarvis micro-benchmark was developed by enhancing PyCG to ensure a more comprehensive coverage of features. For this study, we have incorporated 14 additional test cases from Jarvis, including 4 in the *args* category, 4 in *assignments*, 5 in *direct_calls*, and 1 in *imports*. The final extended PyCG benchmark used in this study includes 126 test cases in the following 17 categories: *Arguments (10), Assignments (8), Builtins (3), Classes (21), Decorators (7), Dicts (12), Direct Calls (9), Dynamic (1), Exceptions (3), Functions (4), Generators (6), Imports (15), Kwargs (3), Lambdas (5), Lists (8), Mro (7), Returns (4)*

5.2 SWARM-CG: Swiss Army Knife of Call Graph Benchmarks

The lack of structured and standardized call-graph benchmarks across diverse programming languages poses several challenges in evaluating and comparing call-graph construction tools. This gap makes cross-language comparisons difficult and unreliable, hindering consistent assessments of different analysis techniques.

To address this issue, we developed the *Swiss Army Knife of Call Graph Benchmarks (SWARM-CG)*, a benchmarking suite designed to provide a standardized platform for evaluating call graph construction tools across multiple programming languages. The primary goal of SWARM-CG is to create a unified environment that facilitates consistent comparisons and promotes further research in the field of call-graph analysis, especially in the current landscape, where ML models are being explored as alternatives to traditional static analysis. ML models often lack the transparency and verifiability that static analysis provides. As researchers investigate these models in call-graph construction, having a standardized framework is essential for accurately comparing their effectiveness with established methods. SWARM-CG fulfills this need by offering a well-organized, comprehensive set of call-graph benchmarks with ground truth annotations for each code snippet, enabling reliable and consistent evaluations.

Furthermore, each tool that SWARM-CG supports is dockerized to simplify the evaluation process. As a proof of concept, we have added support for the following tools: (1) PyCG, (2) Transformers, (3) Ollama, (4) Jelly, (5) TAJS, and (6) SootUp. SWARM-CG supports multiple programming languages, starting with Python, JavaScript, and Java, with plans to extend to additional languages. The suite is designed to be community-driven, encouraging contributions from both static analysis experts and enthusiasts. Thus, it is a dynamic and evolving resource for the research community.

5.3 SWARM-JS: Call-graph Micro-Benchmark

Despite the increasing importance of JavaScript analysis, the availability of well-defined benchmarks tailored for JavaScript call-graph construction remains limited.

Existing benchmarks, such as SunSpider (WebKit, 2010), part of the WebKit browser engine, are primarily designed to test the performance aspects of JavaScript engines rather than facilitating program analysis. SunSpider includes single-file JavaScript examples that represent real-world scenarios in varying complexities and with multiple function types, but it does not provide explicit ground truth for call graphs. In a recent study by (Antal et al., 2023), the authors assessed static call-graph techniques using the SunSpider benchmark by manually comparing the call graphs generated by the tools with the source code. Precision was measured by verifying whether specific edges in the graph were accurately identified. However, this manual approach limits the scope of the evaluation and limits the extensibility of the respective research. Furthermore, the lack of attention to recall in this manual evaluation process results in an incomplete understanding of the tools' performance.

To address these limitations, we developed a new JavaScript micro-benchmark, SWARM-*JS*, tailored specifically for call-graph construction. Inspired by call-graph micro benchmarks in Python, such as PyCG and Jarvis, our benchmark aims to provide a systematic and comprehensive set of test cases that reflect the diverse features of JavaScript.

We first surveyed existing generic JavaScript benchmarks. This survey enabled the identification and compilation of essential JavaScript-specific features and edge cases. A comparative analysis was performed against Python benchmarks, specifically PyCG and Jarvis, to determine which features could be directly adapted and which were unique to JavaScript. For example, Python's lambda functions were mapped to JavaScript's arrow functions due to their analogous roles in both languages.

SWARM-JS comprises 126 JavaScript code snippets, organized into 18 feature categories. The feature categories are: *args (10), assignments (8), builtins (3), classes (21), decorators (7), objects (12), direct_calls (9), dynamic (1), exceptions (3), functions (4), generators (6), imports (15), kwargs (3), arrow functions (5), arrays (8), inheritance (4), mixins (3), returns (4)*. Categories such as arrow functions and mixins ensure comprehensive coverage, allowing the benchmark to evaluate a broad spectrum of JavaScript features.

Each snippet in the benchmark is accompanied by a corresponding ground truth file, which provides the expected call graph. The ground truth schema follows the PyCG benchmark, allowing for a consistent framework for evaluating call-graph accuracy across different languages. The code snippets and ground truth information were manually inspected and iteratively refined to ensure correctness. Additionally, another researcher, who is an expert in JavaScript, verified a randomly selected subset of 25 test cases. An example code snippet is shown in Listing 5.1 and its corresponding ground truth is given in Listing 5.2.

```javascript
function paramFunc() {}


function func(a) {
    a();
}


func(paramFunc);
```

Listing 5.1: Code snippet of main.js

```json
{
    "main.func": [
      "main.paramFunc"
    ],
    "main.paramFunc": [],
    "main": [
      "main.func"
    ]
}
```

Listing 5.2: Ground truth for main.js

5.4 CATS: Call-graph Assessment and Test Suite

CATS (Reif et al., 2019) is a Java project for testing and comparing call graph algorithms. The micro-benchmark is designed to systematically evaluate call-graph construction algorithms by providing small, targeted test cases. These test cases focus on verifying the correct handling of Java language features, core APIs, and runtime (JVM) callbacks. The test suite was created by analyzing the Java Virtual Machine Specification (JVMSpec) (Gosling et al., 2018) and Java's core APIs. The goal was to craft test cases that would only succeed if a call-graph algorithm explicitly handled the specific language feature being tested.

To define the expected behavior, ground truth annotations (GT annotations) are used to capture the call edges relevant to the specific feature being tested in each test case. A call graph algorithm is considered sound if all ground truth annotated call edges are present in its computed call graph. However, this may not always work as some tests may pass unintentionally because of the algorithm's inherent imprecision. Also, the test suite doesn't include test cases related to custom native methods due to limited framework support for cross-language analyses.

In this thesis, we use 109 test cases from CATS micro-benchmark, organized into 15 feature categories. These are: *Classloading (1), DynamicProxies (1), JVMCalls (5), Java8InterfaceMethods (7), Java8Invokedynamics (11), Library (5), ModernReflection (8), NonVirtualCalls (5), Reflection (20), Serialization (14), SignaturePolymorphicMethods (7), StaticInitializers (8), Types (6), Unsafe (7), VirtualCalls (4)*

Test Categories. *Classloading:* Tests related to the usage of *java.lang.ClassLoader*.
DynamicProxies: Java's Dynamic Proxy API is used to create type-safe proxy classes. These proxy classes use reflection to forward the calls to a previously configured handler class.
JVMCalls: Calls of those methods that are (only) done by the JVM due to some event, such as JVM calling the finalize method during garbage collection.
Java8InterfaceMethods: Tests the resolution of Java 8 interface default methods and static interface methods.
Java8Invokedynamics: Tests related to the invokedynamic instructions created by Java 8 method references and lambda expressions.
Library: This category comprises test cases for the analysis of software libraries.
ModernReflection: Reflective calls using the *java.lang.invoke.** APIs and Java 7's *MethodHandle* API which are not signature polymorphic.
NonVirtualCalls: Comprises test cases related to non-virtual methods: static method calls, constructor calls (initializers), super method calls, and private method calls.
Reflection: Tests related to Java's reflection API in combination with *java.lang.Class*'s methods.
Serialization: Comprises test cases that model callbacks that must be handled when dealing with *java.io.Serializable* classes or *java.io.Externalizable* classes. Also tests Java's serialization mechanism when Lambdas are (de)serialized.
SignaturePolymorphicMethods: This category includes calls to signature-polymorphic methods, specifically Java's *java.lang.MethodHandle* methods *invoke* and `invokeExact`, as well as *java.lang.invoke.VarHandle.*
StaticInitializers: Static initializers are called by the JVM when a class is loaded. Those calls are implicit and, therefore, must be explicitly modeled by the CG algorithm.
Types: Test cases that test both: API-based and language-feature-based type casts and instanceof checks.
Unsafe: Test cases related to the usage of *sun.misc.Unsafe.*
VirtualCalls: Tests related to the resolution of standard virtual methods.

The CATS micro-benchmark organizes its test cases into markdown files, with each file corresponding to a specific test category, such as *Reflection* or *Serialization.* Each markdown file is structured using first-level headers to indicate the sub-category of test cases and second-level headers to identify individual test cases. Each test case includes a small, runnable Java program, and the ground truth is embedded directly within the Java source code using annotations. In this study, CATS micro-benchmark was adapted to suit our requirements and to be integrated seamlessly with SWARM-CG. The ground truth information, embedded as annotations within the Java source code, was converted into separate JSON files by a fellow researcher who is an expert in Java. This adaptation decoupled the ground truth from the Java source code with each test case now consisting of the following components:

(1) Java source code
(2) call graph in JSON format
(3) markdown file providing short description

An example Java code snippet is shown in Listing 5.3 and its corresponding ground truth is given in Listing 5.4.

```java
// vc/Class.java
package vc;
class Class {
    public void target(){ }
    public static void main(String[] args){
        Class cls = new Class();
        cls.target();
    }
}
```

Listing 5.3: Code snippet of Class.java

```json
{

    "vc.Class:main(java.lang.String[])": [

        "vc.Class:target()"

    ]

}
```

Listing 5.4: Ground truth for Class.java

6 Methodology

This section discusses the implementation details, model selection criteria, prompt design, and the evaluation metrics used in this study to address the research questions.

6.1 Model Selection

In this study, we selected the LLMs by focusing on organizations actively researching and releasing state-of-the-art models on the Hugging Face platform.[1] We selected 26 LLMs from prominent organizations including OpenAI, Alibaba, Google, Meta, Microsoft, and Mistral, all known for their contributions to advancing LLM research.

The Qwen2 models from Alibaba are expected to perform well in a variety of tasks such as language understanding, language generation, coding, and reasoning. Gemma models, developed by Google are text-text, decoder-only large language models. They are optimized for various text generation tasks, such as summarization and question-answering. The Llama 3.1 and Llama 3.2 instruction-tuned models developed by Meta are optimized for multilingual dialogue tasks. They are designed to provide high-quality summarization capabilities. TinyLlama is a smaller and more compact version of Llama with just 1.1B parameters. It is trained on massive datasets and its compactness makes it suitable for lightweight applications while still delivering impressive performance in various tasks. The Phi-3 models from Microsoft are instruction-tuned and demonstrate capabilities in language understanding, coding, math, long context, and logical reasoning tasks. Mistral models, specifically the Mistral-Large-Instruct-2407 (AI, 2024) is a 123B parameter large language model known for its strong reasoning, knowledge, and coding capabilities. It supports multiple languages, including English, French, and Spanish, and is trained in over 80 programming languages including Python, Java, and JavaScript. The model excels in agentic tasks with function calling and JSON output and offers state-of-the-art mathematical and reasoning abilities.

We also included other specialized code models, which are optimized for code understanding and related tasks. These models are expected to perform better on code-specific benchmarks. CodeLlama models are specifically fine-tuned for coding tasks. They specialize in code synthesis and comprehension, can generate code in various programming languages, and are expected to assist well with debugging, test writing, and even handling unfinished code. The Codestral model is trained in over 80 programming languages. It is also optimized for various coding tasks such as code generation and completion.

[1]`https://huggingface.co/`

Table 6.1: Selected Models and Parameter Sizes

Organization	Model Name	Parameter Size (billion)
Alibaba	Qwen/Qwen2-7B-Instruct	7B
	Qwen/Qwen2-72B-Instruct	72B
Google	google/gemma-2-2b-it	2B
	google/gemma-2-9b-it	9B
	google/gemma-2-27b-it	27B
GPT	gpt-4o	-
	gpt-4o-mini	-
Meta	meta-llama/CodeLlama-7b-Instruct-hf	7B
	meta-llama/CodeLlama-13b-Instruct-hf	13B
	meta-llama/CodeLlama-34b-Instruct-hf	34B
	meta-llama/Meta-Llama-3.1-8B-Instruct	8B
	meta-llama/Meta-Llama-3.1-70B-Instruct	70B
	TinyLlama/TinyLlama-1.1B-Chat-v1.0	1.1B
	meta-llama/Llama-3.2-1B-Instruct	1B
	meta-llama/Llama-3.2-3B-Instruct	3B
Microsoft	microsoft/Phi-3-medium-128k-instruct	14B
	microsoft/Phi-3.5-mini-instruct	3.8B
	microsoft/Phi-3.5-MoE-instruct	41.9B
	microsoft/Phi-3-mini-128k-instruct	3.8B
Mistral	mistralai/Mixtral-8x7B-Instruct-v0.1	46.7B
	mistralai/Mistral-7B-Instruct-v0.3	7B
	mistralai/Mistral-Nemo-Instruct-2407	12.2B
	mistralai/Mistral-Large-Instruct-2407	123B
	mistralai/Mistral-Small-Instruct-2409	22B
	mistralai/Ministral-8B-Instruct-2410	8B
	mistralai/Codestral-22B-v0.1	22B

Two closed-source models from OpenAI, GPT-4o and GPT-4o-mini, were included due to their superior performance in general-purpose tasks, providing a benchmark for comparison against open-source models. The list of models evaluated in this study is listed in the Table 6.1.

6.2 Prompt Design

To optimize prompt design, we adopted an iterative and experimental approach (Chen et al., 2024; Schulhoff et al., 2024). Initial efforts focused on enhancing the prompt by including detailed task descriptions and specifying the expected response format. We used a one-shot prompting technique, embedding an example question and answer within the prompt. Despite these refinements, we encountered challenges with the LLM's ability to produce *structured* outputs. Our experiments revealed that even with explicit instructions to generate outputs in JSON format, models struggled to deliver results that could be reliably parsed. To address this, we explored a question-answer based method, querying the model and then translating its natural-language responses into a structured JSON format. Note that the same prompt is used to evaluate all models, including code models.

The prompt design employed in this study follows a structured two-part approach to guide the LLM through the task. The first part, which is the task description outlines the specific requirements for analyzing the call graph and instructions for formatting the output to ensure consistency in the model's responses. As an example, we have listed the task descriptions for Python, JavaScript, and Java in Listing A.1, A.2, and A.3 respectively. This is followed by the second part, which includes an example input-output pair in line with the one-shot prompting technique. Finally, the code relevant to the task is added to the prompt. Note that for test cases with file imports, all the relevant file contents are added to the prompt with relative file names to indicate the file structure of the test case.

Despite the careful structuring, we encountered difficulties in the initial attempts to generate valid JSON output using this approach. Specifically, the model often failed to consistently produce JSON in the required format. The primary issues observed were missing keys or the inclusion of unexpected keys, attributed to the LLM's inability to adhere to the complex output schema. To address these limitations, the task complexity was reduced by breaking the task down into a series of *question-answer* pairs and using the one-shot prompting technique. This approach simplifies the requirement to follow specific output schema and enhances its ability to follow the prompt more accurately.

For Python and JavaScript, the first question typically addresses function calls at the module level, followed by questions regarding each call made within function definitions as illustrated in Listing A.4 and Listing A.5. For Java, the questions focus on identifying the target functions invoked by a caller function within a given class as shown in Listing A.6. In practical scenarios, these questions can be generated by iterating through the AST of the program. By identifying function definitions and call nodes within the AST, the necessary information can be extracted. However, for this study, ground truth data was used to formulate the questions, allowing for a more straightforward implementation. To clarify, consider the full prompt for a call graph analysis task in Python in Listing A.7. In this case, the first question addresses function calls at the module level *main*. The remaining questions in the prompt are generated by iterating over the ground-truth data and extracting the caller function names. In theory, this information could be obtained by parsing the AST of the program. Examples of the full prompts for JavaScript and Java are also provided in Listing A.8 and A.9, respectively.

Finally, the model's responses were parsed using regular expressions, which enabled the correct mapping of answers back to the original questions. This method allowed for generating JSON outputs that adhered to the respective microbenchmark's ground truth schema, which were then used for the evaluation. To demonstrate this in practice, we have listed examples for Python, Java, and JavaScript, including the source code, ground truth, model responses, and parsed JSON, in Appendix A.2.

The prompts used in this study were comfortably within the context limits of all the LLMs evaluated. The model with the largest context size, GPT-4o, supports up to 128,000 tokens, while the smallest context size was offered by TinyLlama-1.1b, which has a limit of 2,048 tokens.

6.3 Evaluation Metrics

In this study, we measure completeness, soundness, and exact matches to assess flow-insensitive callgraph construction.

6.3.1 Completeness and Soundness

In this study, we use the terms completeness and soundness as they have been pre-established in callgraph research (Salis et al., 2021; Venkatesh et al., 2024a). The terms completeness and soundness are closely related to the precision and recall metrics. Precision is directly tied to completeness, as it measures the proportion of correctly identified call edges relative to all edges produced by the model. A complete call graph would inherently have high precision, as it avoids false positives. This terminology can be a bit confusing at first because it implies that a call graph that is "incomplete" in the above sense is not one that misses call edges but one that has false or incorrect edges. Recall is closely related to soundness, as it measures the proportion of true call edges that are correctly identified. A sound call graph would demonstrate high recall by including all true call edges, without omitting any.

Here, completeness and soundness are measured at the individual test case level within the benchmark. A test case is considered complete if there are no false positives in the generated call graph for that specific case. Similarly, it is considered sound if there are no false negatives. This means that if even a single false positive or false negative is detected in the responses generated for a particular test case, it is marked as a failure in terms of completeness or soundness, respectively. Precision and recall have specific implications when evaluated at the level of individual test cases, particularly in a micro-benchmark setting. Rather than measuring how precise or recall-efficient a system is overall, it is more insightful to determine whether a test case is fully complete or sound with respect to the specific feature being tested. This binary evaluation, either complete or sound, provides clearer insights into whether specific features are fully captured, without the ambiguity that partial correctness metrics like precision or recall might introduce. This evaluation approach mirrors the methodologies used in previous studies, specifically in PyCG (Salis et al., 2021).

6.3.2 Exact matches

The exact-matches metric measures the number of function calls that exactly match the ground truth. To compute this, we compare the expected calls for each node in the ground truth with those produced by the model. For nodes where both lists are non-empty, we count exact matches when every element in the generated list appears in the ground truth. For nodes with empty lists, an exact match is counted if the model also produces an empty list.

6.3.3 Time

Time measurements were taken on open models, as they were all executed on the same hardware using identical parameters for model loading and inference. To ensure uniformity in the testing setup, all models were loaded using 4-bit quantization, with a batch size of 2. To ensure a fair comparison, we applied the same batch size across all models. While smaller models could, in practical scenarios, process more prompts per batch due to lower memory requirements, we chose to standardize the testing conditions. This approach prevents smaller models from having an advantage and allows for a fair assessment.

The time recorded represents the total time needed to process all benchmark test cases. Time measurements for OpenAI models were omitted, as they were inferred using a batch API that returns results after 24 hours at a 50% lower cost. Given that these models were not run on our hardware, a direct comparison with the open models would not be appropriate.

6.4 Implementation Details

6.4.1 LLM Experiments

For the implementation of our LLM experiments, we used the Hugging Face transformers (Wolf et al., 2020) Python interface to run LLMs on our hardware. This interface provides a flexible and efficient environment to manage inference tasks across multiple models.

The models were loaded using 4-bit quantization, with a batch size of 2, and configured to use greedy search. Greedy search was chosen to always select the most probable next token, ensuring deterministic outputs across all runs. All experiments were run on the following hardware configuration: one NVIDIA H100-80C GPU, 16 Intel(R) Xeon(R) Platinum 8462Y+ processors, and 78 GB of memory.

6.4.2 Static Analysis Tools

For the comparative study with existing static analysis tools, we use the PyCG tool for Python, Jelly and TAJS tool for JavaScript, and SootUp for Java. From SootUp, we use the $\text{SootUp}_{\text{CHA}}$ and $\text{SootUp}_{\text{RTA}}$ algorithms.

We built a custom adaptor within the SWARM-CG framework for all call-graph experiments involving these static analysis tools. As the outputs of different tools vary in format, we developed translation functions where necessary to adapt their outputs to the respective micro-benchmark schemas. For JavaScript, the JSON outputs of Jelly and TAJS were translated into the *SWARM-JS* schema, which is the micro-benchmark used for JavaScript evaluations. For the translation of Jelly output JSON to the required schema, the *esprima*[2] tool is employed to generate abstract syntax trees (ASTs), which are then used to restructure and adapt the call graph data. This enables us to evaluate the results effectively within the SWARM-CG framework.

To provide a concrete illustration, an example JavaScript code snippet and its ground truth is shown in Listing 6.1 and 6.2, respectively. The resulting Jelly call graph output is shown in Listing 5.4, and the translated SWARM-JS format output is shown in Listing 5.4.

```
function paramFunc() {
}

function func(a) {
    a();
}

func(paramFunc);
```

Listing 6.1: Code snippet of main.js

```
{
    "main.func": [
        "main.paramFunc"
    ],
    "main.paramFunc": [],
    "main": [
        "main.func"
    ]
}
```

Listing 6.2: Ground truth for main.js

[2] https://esprima.org/

```
{
 "time": "Sat, 30 Nov 2024 11:27:44 GMT",
 "entries": [
  "main.js"
 ],
 "files": [
  "main.js"
 ],
 "functions": {
  "0": "0:1:1:2:2",
  "1": "0:4:1:6:2",
  "2": "0:1:1:8:17"
 },
 "calls": {
  "3": "0:5:5:5:8",
  "4": "0:8:1:8:16"
 },
 "fun2fun": [
  [2, 1], [1, 0]
 ],
 "call2fun": [
  [3, 0], [4, 1]
 ],
 "ignore": []
}
```

Listing 6.3: Jelly output

```
{
    "main.paramFunc": [],
    "main.func": [
        "main.paramFunc"
    ],
    "main": [
        "main.func"
    ]
}
```

Listing 6.4: Translated output

7 Results

We next address the research questions and highlight the key results of our different analyses.

7.1 RQ1: Accuracy of LLMs in performing Callgraph Analysis

The results of our experiments for flow-insensitive call-graph analysis for Python, JavaScript, and Java are presented in Tables 7.1, 7.2 and 7.3 respectively. Specific rows and values that are discussed in the text are highlighted in the table for clarity. In the tables, we've included the existing static analysis tools used for our comparative study of LLMs and traditional tools: PyCG for Python, Jelly and TAJS for JavaScript, and SootUp for Java. While the performance of these tools is discussed in greater detail in Section 7.3 when addressing RQ3, they are included here to give an overall view of the performance metrics.

The Python results are based on the extended PyCG micro-benchmark suite, JavaScript results use the SWARM-JS micro-benchmark, and Java results use the CATS micro-benchmark. For Java, the evaluation only uses soundness as the metric. This is because, the CATS micro-benchmark used for Java, focuses on testing whether the call graph construction tools correctly capture the expected call edges for specific features. Since the ground truth annotations are feature-specific and do not provide an exhaustive list of all possible call edges, evaluating completeness or exact matches would not be meaningful. Focusing solely on soundness ensures fairness and consistency with the benchmark's intended purpose. In contrast, for JavaScript and Python, the evaluation uses soundness, completeness, and exact match metrics. In the following sections, we discuss each of these results in detail.

7.1.1 Python Callgraph Analysis

As evident from the first row of Table 7.1, the static analysis tool PyCG outperforms all 26 LLMs used in this study in terms of completeness, soundness, exact matches, and processing time. For this research question, however, our focus is on analyzing the performance of LLMs.

Top Models. Among the LLMs, the best-performing one was the closed-source model GPT-4o, which achieved a high exact match score of 87.6%, soundness of 71.4%, and a lower completeness score of 60%. This indicates that while it correctly identifies many function call edges, it also introduces false positives and misses some valid edges. The Mistral-Large-IT-2407-123B follows closely, achieving an exact match score of 86.6%, but shows moderate performance in both completeness and soundness.

As we move from smaller to larger models, performance improves generally, with better exact match scores and higher soundness. However, Gemma2-IT-9B with 9B parameters outperforms Gemma2-IT-27B with 27B parameters in terms of exact matches, achieving a score of 64% compared to 58%. Both models have the same soundness score (31.7%), but the 9B model shows a slight edge in completeness (35.7% vs. 23.8%). This indicates that increasing model size does not always lead to better performance.

Models from the Phi-3 family such as Phi3-Mini-IT-3.8B and Phi3.5-Mini-IT-3.8B have shown very poor results compared to others, likely due to their smaller size and limitations in processing capabilities for this task. The phi3.5-moe-it-41.9b model was found to achieve a similar level of language understanding and better reasoning capability compared to mistral-nemo-it-2407-12.2b and llama3.1-it-8b models [1]. However, for this task, it lags in soundness (19.0%) and exact matches score (27.7%) compared to the other two models indicating its limitation for certain tasks. The model also had a high runtime of 1744.23 seconds.

There is a noticeable trade-off between completeness and soundness scores across the smaller models. For instance, models such as the llama3.1-it-8b, llama-3.2-3B-Instruct, and qwen2-it-7b show higher soundness scores (37.3%, 42.0%, and 36.5%) but much lower completeness scores (0.3%, 0.0%, and 0.0%), meaning they tend to produce a lot of false positives. The tinyllama-1.1b model, one of the smallest in terms of parameter size, demonstrated poor performance across all metrics.

Code Models. The codestral-v0.1-22b is designed as a code model with 22B parameters trained in over 80 programming languages, making it suitable for various code-related tasks[2]. However, for this task, the model showed weak results with a 15.9% completeness and 25.5% exact match score, while only slightly surpassing the mistral-Small-Instruct-2409 (a 22.2B model) in soundness (27.8% vs. 26.2%). This performance could suggest limitations in handling specific call graph tasks. The Code Llama instruction-tuned models tested in this study included the 34B, 13B, and 7B variants. These models are optimized for general code generation and understanding. While the codellama-it-34b achieved 48.4% completeness and 19.05% soundness, the smaller models (13B and 7B) showed weaker performance, likely due to their reduced parameter sizes.

To clarify how the results are parsed and evaluated, we provide an example in the appendix, showcasing the source code, ground truth, raw LLM response, and parsed call-graph JSON for both the top-performing model, GPT-4o, and the least-performing model, phi3.5-mini-it-3.8b, for the same test case in our benchmark. These examples can be found in Appendix A.2.1 and A.2.2, respectively.

7.1.2 JavaScript Callgraph Analysis

The results from analyzing the JavaScript SWARM-JS micro-benchmark are presented in Table 7.2. Results of the static tools for JavaScript, Jelly and TAJS are also included.

Top Models. Among the LLMs, GPT-4o achieved 48.4% completeness and 61.9% soundness, making it the top-performing model overall, with an exact match score of 82.7% which indicates a strong performance in capturing valid function calls. The mistral-large-it-2407-123b model achieved 42.0% completeness and 46.0% soundness, and an exact match score of 77.5%, performing comparably to GPT-4o in terms of exact match rates. However, most other open-source LLMs demonstrated weak performance.

Notably, the gemma2-it-9b model outperformed its larger variant, gemma2-it-27b, across all metrics. It achieved an exact match score of 57%, soundness of 17.4%, and completeness of

[1] `https://huggingface.co/microsoft/Phi-3.5-MoE-instruct`

[2] `https://mistral.ai/news/codestral/`

Table 7.1: Comparative analysis across LLMs for flow-insensitive call-graph analysis on the **PyCG Python** micro-benchmark

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Model	**Complete**	**Sound**	**Exact Matches**	**Time (s)**
	126 cases		599 cases	
PyCG	**104** ●	**107** ●	**566** ●	**12.24**
gpt-4o	75 ◕	90 ◕	525 ●	n/a
mistral-large-it-2407-123b	75 ◕	78 ◕	519 ●	1136.99
qwen2-it-72b	34 ◔	69 ◑	431 ◕	640.65
llama3.1-it-70b	40 ◔	57 ◑	413 ◕	576.95
mistral-Small-Instruct-2409	80 ◕	33 ◔	399 ◕	207.86
gpt-4o-mini	44 ◔	47 ◔	394 ◕	n/a
gemma2-it-9b	45 ◔	40 ◔	384 ◕	171.73
mistral-nemo-it-2407-12.2b	48 ◔	43 ◔	366 ◕	104.59
gemma2-it-27b	30 ◔	40 ◔	348 ◑	332.18
ministral-8B-Instruct-2410	18 ○	53 ◑	282 ◑	112.8
llama3.1-it-8b	5 ○	47 ◔	202 ◔	171.7
phi3.5-moe-it-41.9	7 ○	24 ○	166 ◔	1744.23
qwen2-it-7b	2 ○	46 ◔	154 ◔	99.93
codestral-v0.1-22b	20 ○	35 ◔	153 ◔	970.27
llama-3.2-3B-Instruct	0 ○	53 ◑	153 ◔	98.25
phi3-medium-it-14b	3 ○	29 ◔	143 ◔	263.79
codellama-it-34b	61 ◑	24 ○	132 ◔	782.02
mistral-v0.3-it-7b	5 ○	26 ◔	128 ◔	166.03
tinyllama-1.1b	36 ◔	5 ○	86 ○	806.37
gemma2-it-2b	13 ○	10 ○	83 ○	104.07
llama-3.2-1B-Instruct	2 ○	19 ○	78 ○	100.49
mixtral-v0.1-it-8x7b	2 ○	18 ○	50 ○	879.32
codellama-it-13b	19 ○	19 ○	44 ○	921.53
codellama-it-7b	6 ○	4 ○	30 ○	456.33
phi3-mini-it-3.8b	1 ○	7 ○	17 ○	138.36
phi3.5-mini-it-3.8b	2 ○	6 ○	11 ○	138.58

33.3%. In comparison, the gemma2-it-27b achieved an exact match score of 50%, soundness of 11.9%, and completeness of 11%. The phi3.5-moe-it-41.9b model performed poorly and had a very high runtime of 2627.33 seconds, making it both inefficient and ineffective. The codellama-it-34b model, one of the code-specific models, showed high completeness score (83/126), but very low soundness (9/126). This indicates that although the model generated few false positives, it missed nearly all valid function calls, leading to mostly empty call graphs. The least-performing model was mixtral-v0.1-it-8x7b with very poor results across all metrics.

An example showing the source code, ground truth, raw LLM response, and parsed call graph response for GPT-4o and mixtral-v0.1-it-8x7b is provided in Appendix A.2.3 and A.2.4, respectively.

7.1.3 Java Callgraph Analysis

The results of Java call-graph analysis on CATS micro-benchmark are presented in Table 7.3. Results of the static tool, SootUp for Java are also included in the table. From SootUp, we utilized the SootUp$_{\text{CHA}}$ and SootUp$_{\text{RTA}}$ algorithms.

For Java, the evaluation focuses solely on soundness as the primary metric. This decision aligns with the design of CATS micro-benchmark, which emphasizes testing whether the call-graph construction tools correctly capture the expected call edges for specific features. Since the CATS ground truth is feature-specific and does not provide an exhaustive list of all possible call edges, evaluating only soundness ensures fairness.

Top Models. Among the 26 models evaluated, GPT-4o achieved the highest overall performance with a high soundness score of 82.5%, showcasing its robustness in identifying valid function call edges. The mistral-Small-Instruct-2409, a 22B parameters model followed very closely behind with a soundness of 80.7%. Llama3.1-it-70b also demonstrated strong performance, having a soundness of 78%. Both models outperformed larger models such as the mistral-large-it-2407-123b. These results underscore GPT-4o's dominance, but also the competitive capabilities of open-source models like Mistral and LLaMA.

An interesting observation is the performance of the gemma2-it-9b model. It achieved a soundness score of 68.8%, which is comparable to the 69.7% score achieved by mistral-large-it-2407-123b. This near-equivalent performance between gemma2-it-9b and mistral-large-it-2407-123b, despite the vast difference in parameter sizes, underscores the potential of smaller, optimized models to rival significantly larger ones in performing specific tasks.

The smaller models, such as phi3.5-mini-it-3.8b, mistral-nemo-it-2407-12.2b, mistral-v0.3-it-7b, qwen2-it-7b, and llama-3.2-1B-Instruct, exhibited very weak performance. For instance, the phi3-mini-it-3.8b model achieved a soundness score of only 2/109. This suggests that the model struggled to identify valid function calls, leading to numerous test case failures and the generation of nearly empty call graphs.

Code Models. The code-specific models showed moderate to weak performance, with codellama-it-7b emerging as the top performer among the codellama models. It achieved a soundness score of 32/109 (29.3%), indicating limitations in identifying valid edges. The codestral-v0.1-22b model, despite being larger in terms of parameters, performed even worse, achieving a soundness score of just 10/109 (9.1%).

We have provided an example in the appendix, showcasing the source code, ground truth, raw LLM response, and parsed call-graph JSON for GPT-4o (see Appendix A.2.5) and Llama-3.2-1B-Instruct (see Appendix A.2.6).

Table 7.2: Comparative analysis across LLMs for flow-insensitive call-graph analysis on the **SWARM-JS JavaScript** micro-benchmark

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Model	**Complete**		**Sound**		**Exact Matches**		**Time (s)**
	126 cases				596 cases		
gpt-4o	61	◑	78	◕	493	●	n/a
Jelly	**49**	◔	**85**	◕	**490**	●	**643.51**
mistral-large-it-2407-123b	53	◑	58	◑	462	◕	1124.65
qwen2-it-72b	31	◔	52	◑	393	◕	704.53
llama3.1-it-70b	27	◔	34	◔	359	◕	565.94
mistral-Small-Instruct-2409	65	◑	18	○	357	◕	210.63
gpt-4o-mini	28	◔	27	◔	341	◑	n/a
gemma2-it-9b	42	◔	22	○	340	◑	162.29
mistral-nemo-it-2407-12.2b	45	◔	25	○	307	◑	103.64
gemma2-it-27b	14	○	15	○	298	◑	316.47
ministral-8B-Instruct-2410	16	○	47	◔	265	◑	115.93
llama3.1-it-8b	4	○	39	◔	194	◔	108.29
phi3.5-moe-it-41.9b	3	○	11	○	161	◔	2627.33
llama-3.2-3B-Instruct	0	○	45	◔	156	◔	95.73
mistral-v0.3-it-7b	6	○	22	○	145	◔	185.56
codestral-v0.1-22b	40	◔	14	○	126	◔	1006.89
phi3-medium-it-14b	2	○	18	○	125	◔	258.44
qwen2-it-7b	2	○	27	◔	99	○	88.68
gemma2-it-2b	10	○	6	○	95	○	105.76
TAJS	**119**	●	**14**	○	**83**	○	**136.74**
codellama-it-34b	83	◕	9	○	83	○	835.92
llama-3.2-1B-Instruct	0	○	14	○	72	○	130.7
codellama-it-7b	14	○	2	○	42	○	484.55
phi3-mini-it-3.8b	3	○	2	○	41	○	112.45
tinyllama-1.1b	11	○	1	○	29	○	872.21
codellama-it-13b	5	○	10	○	23	○	1180.64
phi3.5-mini-it-3.8b	3	○	3	○	20	○	151.07
mixtral-v0.1-it-8x7b	2	○	1	○	14	○	765.36

Table 7.3: Comparative analysis across LLMs for flow-insensitive call-graph analysis on the **CATS Java** micro-benchmark

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Model	**Sound** *(109 cases)*		**Time (s)**
gpt-4o	90	●	n/a
mistral-Small-Instruct-2409	88	●	258.88
llama3.1-it-70b	85	◕	400.32
gpt-4o-mini	76	◕	n/a
mistral-large-it-2407-123b	76	◕	1270.26
gemma2-it-9b	75	◕	119.06
qwen2-it-72b	73	◕	427.13
ministral-8B-Instruct-2410	69	◕	323.97
llama3.1-it-8b	66	◕	118.12
gemma2-it-27b	62	◑	330.12
phi3-medium-it-14b	50	◑	162.63
llama-3.2-3B-Instruct	45	◑	148.32
SootUp$_{CHA}$	41	◔	649.48
SootUp$_{RTA}$	41	◔	647.02
codellama-it-7b	32	◔	282.07
codellama-it-34b	24	◔	1400.1
mixtral-v0.1-it-8x7b	21	◔	396.86
gemma2-it-2b	19	○	53.85
phi3.5-moe-it-41.9b	17	○	669.72
codellama-it-13b	17	○	482.28
codestral-v0.1-22b	10	○	850.35
tinyllama-1.1b	5	○	867.49
phi3.5-mini-it-3.8b	4	○	228.42
mistral-v0.3-it-7b	3	○	74.81
qwen2-it-7b	3	○	40.1
llama-3.2-1B-Instruct	3	○	78.7
mistral-nemo-it-2407-12.2b	2	○	59.95
phi3-mini-it-3.8b	2	○	290.05

7.2 RQ2: LLM Accuracy in Call Graph Analysis Across Languages

To address this research question, we focus on comparing the performance of LLMs in call graph analysis across Python and JavaScript. Although Java is included in this study, a direct comparison with Python and JavaScript is not feasible due to the specific characteristics of the CATS micro-benchmark used for Java evaluations. The CATS micro-benchmark focuses on capturing call edges relevant to the specific features under test rather than providing a comprehensive view of all possible call edges. This feature-specific focus means that the ground truth annotations in CATS are not exhaustive and therefore, metrics such as completeness and exact matches cannot be meaningfully applied. As a result, Java evaluations are limited to the soundness metric, which reflects feature-specific correctness. In contrast, Python and JavaScript evaluations use micro-benchmarks that are designed to evaluate call graphs comprehensively. These benchmarks enable the computation of metrics such as soundness, completeness, and exact matches. Since Python and JavaScript results are based on these directly comparable metrics, their comparison is both fair and meaningful. Thus, to ensure fairness and validity, we limit this section to comparing Python and JavaScript results. Table 7.4 presents the top 10 performing LLMs, with completeness, soundness, and exact match rates reported for both languages.

GPT-4o, the top-performing model for both languages, achieved a high exact match score of 88% in Python, compared to 83% in JavaScript. Similarly, GPT-4o's soundness scores were 71% for Python and 62% for JavaScript, while its completeness scores were 60% and 48%, respectively. Across both languages, GPT-4o exhibited consistent dominance, outperforming other models across all metrics. Following GPT-4o, the second-best model, mistral-large-it-2407-123b, demonstrated a similar trend, with Python results outperforming JavaScript across all three metrics. The exact match rates were 87% in Python compared to 78% in JavaScript, soundness scores were 62% and 46%, and completeness scores were 60% and 42%, respectively.

The top four models for both Python and JavaScript were the same: GPT-4o, mistral-large-it-2407-123b, llama3.1-it-70b, and qwen2-it-72b. This indicates that these models maintained consistent performance irrespective of the programming language. However, the performance gap between these top models was more pronounced in JavaScript. For instance, the difference in exact match rates between GPT-4o and the second-best model (mistral-large-it-2407-123b) was 5 percentage points (83% vs. 78%) in JavaScript, compared to just 1 percentage point (88% vs. 87%) in Python.

Interestingly, smaller variants of certain models outperformed their larger counterparts in both Python and JavaScript. For example, gemma2-it-9b consistently outperformed gemma2-it-27b across all metrics. In Python, gemma2-it-9b achieved an exact match rate of 64%, compared to 58% for gemma2-it-27b. Similarly, in JavaScript, gemma2-it-9b achieved 57%, while the larger variant achieved only 50%. This trend highlights that scaling model size does not always guarantee better performance. The smallest model in the comparison, llama3.1-it-8b, exhibited notably weak performance, particularly in completeness scores across both Python and JavaScript. In Python, it achieved a completeness score of only 4%, and in JavaScript, the score was 3%. While its exact match and soundness scores were slightly better, llama3.1-it-8b still lagged behind the top-performing models, indicating the challenges faced by smaller models in tackling call graph analysis tasks.

Table 7.4: Percentage comparison of models across Python and JavaScript flow-insensitive call-graph evaluations

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Model	**Python (%)**			**JavaScript (%)**		
	Comp.	**Sound**	**EM.**	**Comp.**	**Sound**	**EM.**
gpt-4o	60 ◕	71 ◕	88 ●	48 ◑	62 ◕	83 ●
mistral-large-it-2407-123b	60 ◕	62 ◕	87 ●	42 ◑	46 ◑	78 ◕
llama3.1-it-70b	32 ◔	45 ◑	69 ◕	21 ◔	27 ◔	60 ◕
qwen2-it-72b	27 ◔	55 ◑	72 ◕	25 ◔	41 ◑	66 ◕
mistral-Small-Instruct-2409	64 ◕	26 ◔	67 ◕	52 ◑	14 ○	60 ◕
gpt-4o-mini	35 ◔	37 ◔	66 ◕	27 ◔	22 ◔	59 ◑
gemma2-it-9b	36 ◔	32 ◔	64 ◕	33 ◔	18 ○	57 ◑
gemma2-it-27b	24 ◔	32 ◔	58 ◑	11 ○	12 ○	50 ◑
mistral-nemo-it-2407-12.2b	38 ◔	34 ◔	61 ◕	36 ◔	20 ◔	52 ◑
llama3.1-it-8b	4 ○	37 ◔	34 ◔	3 ○	31 ◔	33 ◔

7.3 RQ3: Comparison of LLMs and Traditional Static Analysis Tools

To address this research question, we compare the top-performing LLM for each programming language with an existing static analysis tool for that language. Details of the existing call graph construction tools used in this study are discussed in Section 2.2. For Python, the results of call graph analysis using the static analysis tool PyCG and the top-performing model GPT-4o, are presented in Table 7.5. This comparison is based on the PyCG micro-benchmark. For Java, the results of call graph analysis using GPT-4o and SootUp on the CATS micro-benchmark are summarized in Table 7.7. For JavaScript, Table 7.6 presents the results of Jelly and GPT-4o on the SWARM-JS JavaScript micro-benchmark.

The static analysis tool for JavaScript, TAJS was also evaluated because it had demonstrated strong performance in call-graph generation in a study by Antal et al. (2023). However, in our study (Table 7.2), TAJS produced poor results. While its completeness score appeared high at first glance, this was largely due to 102 out of 126 test cases generating errors and producing empty outputs, which reduced the false positive count. This limitation arises because TAJS only fully supports ECMAScript 3rd edition, whereas the SWARM-JS benchmark includes modern features from ECMAScript 6th edition, such as classes and arrow functions. Jelly, which supports the latest ECMAScript features, delivered significantly better performance.

7.3.1 Python

We present the call graph analysis results of PyCG and GPT-4o on the PyCG micro-benchmark in Table 7.5. To provide a detailed overview of performance across different feature categories, the results are presented category-wise. The results highlight the superior performance of the static analysis tool PyCG compared to the top-performing model, GPT-4o in terms of completeness, soundness, and exact matches.

Table 7.5: Comparison of **Pycg** and **GPT-4o** on the **PyCG** Python micro-benchmark

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Category	**Pycg**			**GPT-4o**		
	Complete	**Sound**	**E.M.**	**Complete**	**Sound**	**E.M.**
args	6/10	10/10	38/38	4/10	6/10	33/38
assignments	4/8	7/8	36/38	7/8	7/8	37/38
builtins	3/3	1/3	7/17	1/3	1/3	8/17
classes	21/21	21/21	92/92	7/21	18/21	85/92
decorators	5/7	5/7	37/39	4/7	3/7	32/39
dicts	9/12	11/12	40/41	10/12	11/12	38/41
direct_calls	4/9	4/9	63/70	5/9	7/9	66/70
dynamic	0/1	0/1	1/3	0/1	0/1	0/3
exceptions	3/3	3/3	6/6	3/3	3/3	6/6
functions	4/4	4/4	9/9	4/4	4/4	9/9
generators	6/6	6/6	36/36	3/6	3/6	30/36
imports	12/15	11/15	70/75	11/15	11/15	69/75
kwargs	3/3	3/3	15/15	0/3	0/3	9/15
lambdas	5/5	5/5	23/23	5/5	3/5	18/23
lists	8/8	7/8	35/36	7/8	5/8	33/36
mro	7/7	5/7	34/37	3/7	4/7	31/37
returns	4/4	4/4	24/24	1/4	4/4	21/24
Total	104/126 ●	107/126 ●	566/599 ●	75/126 ◕	90/126 ◕	525/599 ●

Overall Performance. In a benchmark of 126 test cases, PyCG achieved 83% completeness and 85% soundness, significantly surpassing GPT-4o, which attained 61% completeness and 71% soundness. This means that for the majority of test cases, PyCG produced no false positives (completeness) and missed very few valid function calls (soundness). Additionally, PyCG generated 566 exact matches out of 599, outperforming GPT-4o by a margin of 41 matches.

Category-wise Performance. Across the 17 feature categories evaluated, PyCG achieved 100% soundness in 8 categories (*args, classes, exceptions, functions, generators, kwargs, lambdas, and returns*). This demonstrates PyCG's accuracy in generating call graphs with no false negatives across diverse test cases. GPT-4o achieved a 100% soundness score in 3 categories (*exceptions, functions, and returns*). Some other categories in which GPT-4o performed well in terms of soundness scores are *assignments* (7/8) and *dicts* (11/12). For exact matches, PyCG outperformed GPT-4o in all categories except *direct calls* and *assignments*, where GPT-4o demonstrated a slight edge. In categories like *classes*, PyCG reached a perfect 92/92 exact match score, exceeding GPT-4o's performance (85/92). There are some categories in which both PyCG and GPT-4o achieved similar scores across all metrics. For example, *imports, functions,* and *exceptions.*

7.3.2 JavaScript

We present the results of Jelly and GPT-4o on the SWARM-JS micro-benchmark in Table 7.6. The SWARM-JS micro-benchmark consists of 18 categories, each corresponding to distinct JavaScript language features such as arrow functions, classes, and functions. To provide a detailed understanding of performance across these features, the results are presented category-wise.

Overall Performance. When we look at the overall performance, both tools achieved comparable results for exact matches, with Jelly at 490/596 (82.2%) and GPT-4o slightly ahead at 493/596 (82.7%). This indicates that in terms of exact match rates, GPT-4o has a slight edge over Jelly. However, Jelly performed better in soundness, achieving 85/126 (67.5%) compared to GPT-4o's 78/126 (61.9%). This suggests that Jelly identified more valid function call edges and produced less number of false negatives. On the other hand, GPT-4o achieved a higher completeness score (48.4% vs. 38.9%), indicating fewer false positives in the generated call graphs. In this study, we ran the Jelly tool with its *approximate interpretation* feature enabled. This approach, which combines elements from static and dynamic analysis, focuses on improving the accuracy of JavaScript analysis and reducing unsoundness (Laursen et al., 2024). This hybrid methodology likely contributed to Jelly's strong performance in soundness.

Category-wise Performance. Jelly outperformed GPT-4o in soundness for certain JavaScript features such as *classes* (20/21 vs. 14/21), *args* (10/10 vs. 7/10), and *arrow_functions* (4/5 vs. 3/5). On the other hand, GPT-4o demonstrated higher completeness in most categories, such as *direct_calls* (5/9 vs. 2/9), *imports* (6/15 vs. 0/15) and *assignments* (7/8 vs. 4/8). In terms of exact match scores, Jelly matched or surpassed GPT-4o in categories such as *args* (37/37 vs. 33/37), *arrow_functions* (21/23 vs. 19/23), *classes* (87/92 vs. 80/92), *inheritance* (17/18 vs. 14/18), and *mixins* (12/16 vs. 5/16). GPT-4o, however, showed better results in categories like *arrays* (32/38 vs. 26/38), *builtins* (13/19 vs. 6/19), *imports* (66/80 vs. 61/80), and *direct_calls* (63/69 vs. 57/69).

Overall, while GPT-4o demonstrated marginally better exact match rates and higher completeness, Jelly outperformed in soundness, showcasing its ability to identify valid function call edges. This strong performance from Jelly is likely due to its approximate interpretation feature.

7.3.3 Java

We present the results of SootUp$_{\text{CHA}}$ and GPT-4o on the CATS micro-benchmark in Table 7.7. From SootUp, we utilized the SootUp$_{\text{CHA}}$ and SootUp$_{\text{RTA}}$ algorithms. As evident from the overall results in Table 7.3, both algorithms struggled to perform well on the CATS benchmark. In this section, we focus on comparing SootUp$_{\text{CHA}}$ with GPT-4o, the top-performing model in Java.

Overall Performance. If we look at the overall performance, GPT-4o demonstrated a significant advantage in terms of soundness. GPT-4o achieved 90/109 (82.5%) in soundness, compared to SootUp$_{\text{CHA}}$'s 41/109 (37.6%) for soundness. This indicates that GPT-4o was significantly better at identifying valid function call edges for the test cases in the CATS benchmark.

Category-wise Performance. GPT-4o achieved near-perfect soundness scores in several categories, including *Reflection* (20/20), *Static Initializers* (8/8), *Virtual Calls* (4/4), *Java8 Invoked Dynamics* (10/11), and *JVM Calls* (4/5). In contrast, SootUp$_{\text{CHA}}$ struggled significantly in some of these categories, scoring 4/20 for *Reflection*, 0/11 for *Java8 Invoked Dynamics*, and 0/5 for *JVM Calls*. However, SootUp$_{\text{CHA}}$ performed relatively well in a few categories achieving high soundness, such as *Types*, *Virtual Calls*, *Unsafe*, and *Java8 Interface Methods*. The weak

Table 7.6: Comparison of **Jelly** and **GPT-4o** on the **SWARM-JS** JavaScript micro-benchmark

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Category	**Jelly**			**GPT-4o**		
	Complete	**Sound**	**E.M.**	**Complete**	**Sound**	**E.M.**
args	4/10	10/10	37/37	5/10	7/10	33/37
arrays	3/8	5/8	26/38	4/8	6/8	32/38
arrow_functions	4/5	4/5	21/23	3/5	3/5	19/23
assignments	4/8	8/8	38/38	7/8	7/8	37/38
builtins	2/3	0/3	6/19	2/3	1/3	13/19
classes	5/21	20/21	87/92	6/21	14/21	80/92
decorators	1/7	2/7	25/39	5/7	3/7	33/39
direct_calls	2/9	3/9	57/69	5/9	6/9	63/69
dynamic	1/1	0/1	1/4	0/1	0/1	1/4
exceptions	3/3	2/3	4/6	0/3	3/3	3/6
functions	3/4	4/4	9/9	3/4	3/4	8/9
generators	6/6	1/6	16/29	3/6	1/6	19/29
imports	0/15	7/15	61/80	6/15	6/15	66/80
inheritance	0/4	3/4	17/18	1/4	2/4	14/18
kwargs	2/3	2/3	13/14	1/3	1/3	9/14
mixins	0/3	1/3	12/16	0/3	1/3	5/16
objects	7/12	9/12	37/41	9/12	11/12	38/41
returns	2/4	4/4	23/24	1/4	3/4	20/24
Total	49/126 ◔	85/126 ◕	490/596 ●	61/126 ◑	78/126 ◕	493/596 ●

Table 7.7: Comparison of Java Call Graph Analysis using **SootUp$_{\mathbf{CHA}}$** and **GPT-4o** on the **CATS** Java micro-benchmark

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Category	**SootUp$_{\mathbf{CHA}}$**	**GPT-4o**
	Sound	**Sound**
Classloading	0/1	1/1
DynamicProxies	0/1	0/1
JVMCalls	0/5	4/5
Java8InterfaceMethods	7/7	4/7
Java8Invokedynamics	0/11	10/11
Library	0/5	0/5
ModernReflection	2/8	7/8
NonVirtualCalls	4/5	4/5
Reflection	4/20	20/20
Serialization	0/14	12/14
SignaturePolymorphicMethods	0/7	6/7
StaticInitializers	7/8	8/8
Types	6/6	5/6
Unsafe	7/7	5/7
VirtualCalls	4/4	4/4
Total	41/109 ◔	90/109 ●

performance of SootUp$_{\text{CHA}}$ in the CATS benchmark can largely be attributed to its lack of support for dynamic language features, which are essential for handling several Java constructs present in this benchmark.

8 Discussion

8.1 Performance of LLMs in Call Graph Analysis

The results of our study show that LLMs demonstrate varying levels of effectiveness in call graph analysis across Python, JavaScript, and Java, with differences observed in completeness, soundness, and exact match metrics for Python and JavaScript, while only soundness was evaluated for Java.

Among the 26 models evaluated, the closed-source model GPT-4o outperformed its open-source counterparts, achieving the highest scores across all metrics in Python and JavaScript and demonstrating strong soundness in Java. Some open-source models also achieved competitive results in specific metrics. For example, in Java, the Mistral-Small-Instruct-2409 model performed particularly well, achieving soundness comparable to GPT-4o. Similarly, in Python, the Mistral-Large-Instruct-2407-123B model demonstrated competitive performance, closely matching GPT-4o in terms of exact matches and completeness. This suggests that open-source models are beginning to catch up with closed-source models.

While model size played a role in performance, this relationship was not always linear. Larger models, such as Mistral-Large-Instruct-123B, demonstrated steady performance and adaptability across languages. However, in Java, the smaller Mistral-Small-Instruct-2409 significantly outperformed the larger model. Similarly, the Gemma2-it-9B model outperformed the larger, Gemma2-it-27B consistently across all three languages. These findings indicate that a higher parameter count does not always guarantee better results. On the other hand, the consistently poor performance of smaller models like TinyLlama-1.1B and Llama-3.2-1B-Instruct suggests that a certain level of model complexity is necessary for effective call graph analysis. The code-specific models included in this study, such as Codestral-v0.1-22B and CodeLlama-it-34B, did not rank among the top ten performing models in any of the languages.

8.2 Performance Across Programming Languages

Python and JavaScript. For Python and JavaScript, completeness, soundness, and exact match metrics were evaluated. Across these metrics, Python outperformed JavaScript overall with most models achieving higher scores in Python. In Python, GPT-4o was the best-performing model, while mistral-large-it-2407-123b demonstrated competitive performance, especially in exact matches. In JavaScript, GPT-4o was again the leading model, but the performance gap between GPT-4o and other open-source models such as mistral-large-it-2407-123b, was more pronounced. Also, most other open-source models showed poor performance compared to Python. Early studies have indicated that LLMs perform better on code generation tasks in Python (Buscemi, 2023), which can be attributed to its simple and cleaner syntax and well-established conventions, as speculated in previous work (Chen et al., 2021). Yet, further research is needed to fully understand the performance gap between the languages.

Java. Java was analyzed solely for soundness due to the specific characteristics of the CATS benchmark used in this study. GPT-4o, Mistral-Small-Instruct-2409, and Llama-3.1-70B were the top three models, all demonstrating comparable soundness rates. Models like gemma2-it-9 achieved nearly equivalent performance to much larger models such as Mistral-Large-It-2407-123b. This highlights the potential for smaller, optimized models to compete with significantly larger ones in performing specific tasks.

8.3 Traditional SA Tools and LLMs

Similar to findings in previous work (Ma et al., 2023; Sun et al., 2023; Venkatesh et al., 2024b), our results show that the construction of call graphs does not yet significantly benefit from the use of LLMs, especially in Python.

The static analysis tool PyCG outperformed all LLMs in Python. PyCG achieved near-perfect results, underscoring the current limitations of LLMs in handling static analysis tasks for Python. In JavaScript, the static analysis tool Jelly performed better in soundness but was surpassed by GPT-4o in completeness and exact match rates. However, this difference can be attributed to Jelly's primary focus on achieving better soundness through its approximate interpretation feature, which emphasizes identifying valid call edges while reducing unsoundness. The TAJS tool for JavaScript, however, produced poor results on modern JavaScript programs. This can be attributed to the fact that TAJS does not have support for the latest JavaScript features and has not been actively developed or maintained for the past four years. For Java, SootUp underperformed compared to the LLMs, mainly due to its inability to handle dynamic language features effectively. In contrast, the LLMs demonstrated strong performance in Java, with around five models achieving soundness and exact match rates above 70%.

One notable limitation of the LLMs tested in this study is their size. Most of the models had over seven billion parameters, making them unsuitable for deployment on standard machines equipped with a single GPU. In contrast, traditional SA tools like PyCG and Jelly can operate effectively within these hardware constraints, making them more practical for many SA tasks.

A potential direction for future research is to explore the performance of fine-tuned smaller variants of LLMs across different languages. This could help us understand if fine-tuning smaller models for specific tasks could improve their performance in call graph analysis, without the resource constraints that come with using larger models.

9 Threats to Validity

We acknowledge the following limitations and threats to the validity of our study:

- We applied the same prompt to all models, which may not have optimized performance for each individual model. Tailored prompts could potentially extract better results from specific models.
- Open-source models frequently deviated from the expected output formats provided in the prompt. To mitigate this, we manually identified response patterns and added a preprocessing step to standardize the format. However, this approach may not account for all variations, further underscoring the challenge of consistently generating structured data with LLMs.
- While we tested several prompts iteratively, our approach did not focus exclusively on optimizing prompt engineering. A dedicated experiment to explore different prompting strategies could lead to better results. Our modular framework can serve as a foundation for future research aimed at refining prompts to improve performance.
- We used greedy search for token prediction, always selecting the highest-probability token. Future research could explore higher temperature settings and incorporate a voting mechanism to identify the best output, potentially yielding better results.
- While micro-benchmarks are useful for isolating and evaluating specific aspects of system performance, they may miss the complexity and variability of real-world workloads and use cases.
- We employed translation functions to adapt the outputs of static analysis tools like Jelly and TAJS to the respective micro-benchmark schemas. This approach allowed for standardized evaluation across different tools, but it also introduced a limitation in directly comparing the outputs from these tools.

10 Conclusion

This study provides a comprehensive evaluation of LLMs in static analysis tasks, particularly call-graph construction, using specialized micro-benchmarks across Python, JavaScript, and Java programs. Our results reaffirm that while LLMs offer promising capabilities in various software engineering tasks, traditional static analysis methods remain more effective for call-graph construction, particularly in Python. Similar to findings in previous studies, LLMs have yet to surpass the efficiency of static tools like PyCG for this task. In JavaScript, LLMs demonstrated competitive performance, with models like GPT-4o achieving strong completeness and exact match rates. However, most other open-source models exhibited very weak performance in JavaScript compared to their performance in Python. Further investigation is required to fully understand the performance gap between Python and JavaScript. Static tools such as Jelly for JavaScript outperformed the LLMs in terms of soundness, highlighting the strengths of traditional analysis methods in handling language-specific features.

In the case of Java, LLMs showed a notable improvement, with several models achieving above 70% soundness. However, the inability of SootUp to handle dynamic features effectively contributed to its lower performance compared to LLMs. Despite these promising results, LLMs are still limited by their size and resource requirements. Most models evaluated in this study had over seven billion parameters, making them unsuitable for deployment in resource-constrained environments. In contrast, traditional static analysis tools can run efficiently within these hardware constraints, making them more practical for many real-world scenarios.

Future work should explore how fine-tuned models perform across different languages and tasks, as this could potentially lead to better results without the significant resource demands associated with larger models. This study demonstrates the potential of LLMs in software engineering tasks, while also emphasizing their limitations and the continued strengths of traditional methods especially in languages like Python. However, with continued advancements and further exploration of task-specific fine-tuning, LLMs could become a valuable addition to existing static analysis methods.

A Appendix

A.1 Prompts

```
## Task Description

**Objective**: Examine and identify the function calls in the given Python code and
    answer the questions.

**Instructions**:
1. For each question below, provide a concise answer indicating the function calls.
2. List every function call as a comma separated list.
3. Do not include additional explanations or commentary in your answers.
4. Include both explicit and implicit function calls in your answers. An implicit
    function call is a function that is called as a result of another operation, such as
    the __init__ method being called when an object is created.
5. If a function is called through an alias or a reference, identify and list the actual
    function that is called after resolving the alias.
6. If a passed argument is not invoked within the function, do not include the function
    call in the answer.
7. Examples of Python code, questions, and answers are given below. This example should
    be used as training data.
```

Listing A.1: Prompt for call graph task in Python in question-answer format (Part 1 of 2)

```
## Task Description

**Objective**: Examine and identify the function calls in the given JavaScript code and
    answer the questions.

**Instructions**:
1. For each question below, provide a concise answer indicating the function calls.
2. List every function call as a comma separated list.
3. Do not include additional explanations or commentary in your answers.
4. Include both explicit and implicit function calls in your answers. An implicit
    function call is a function that is called as a result of another operation, such as
    the constructor method being called when an object is created.
5. If a function is called through an alias or a reference, identify and list the actual
    function that is called after resolving the alias.
6. If a passed argument is not invoked within the function, do not include the function
    call in the answer.
7. Example of JavaScript code, questions, and answers are given below. This example
    should be used as training data.
```

Listing A.2: Prompt for call graph task in JavaScript in question-answer format (Part 1 of 2)

```
## Task Description

**Objective**:
Examine the provided Java code, analyze the function calls, and answer the questions
    about the caller-target relationships.

**Instructions**:
1. For each question provide the target functions as a comma-separated list.

2. **IMPORTANT**: Ensure that the target function in each response is formatted as
    specified in the method signature format:

    - [Fullname of Class]:[method name]([Parameterlist])

    - Example: vc.Class:target(java.lang.String[])

3. Do not include additional explanations or comments in your answers.

4. Do not include the questions in your response. Only list the answers using the
    question number format shown below, that is, provide your answer for each question
    next to the corresponding question number

    - Response format example:
    - 1. vc.Class:(java.lang.String[]), vc.Class:(java.lang.Integer)

5. For methods called through aliases or references, resolve and return the actual target
     method.

6. Examples of Java code, questions, and answers are given below. This example should be
    used as training data.
```

Listing A.3: Prompt for call graph task in Java in question-answer format (Part 1 of 2)

~~~
**Format for Answers**:
- Provide your answer next to each question number, using only one word.
- Do not include the questions in your answer.
- Example:
    1. simple.func
    2. simple.examplefunc
    3. func

**Example Python Code**:
```main.py
def return_func():
    func()

def func():
    a = return_func
    return a

a = func
a()()
~~~

Example Questions:
1. What are the module level function calls in the file "main.py"?
2. What are the function calls inside the "main.return_func" function in the file "main.
    py"?
3. What are the function calls inside the "main.func" function in the file "main.py"?

Example Answers:
1. main.func, main.return_func
2. main.func
3.

{language} Code Provided:

{code}

Questions:
{questions}

Answers:
{answers}"""
```

Listing A.4: Prompt for call graph task in Python in question-answer format (Part 2 of 2)
```

~~~
**Format for Answers**:
- Provide your answer next to each question number, using only one word.
- Do not include the questions in your answer.
- Example:
    1. simple.func
    2. simple.examplefunc
    3. func

**Example Javascript Code**:
```main.js
function returnFunc() {{
    func();
}}

function func() {{
    a = returnFunc;
    return a;
}}

a = func;
a()();
~~~

**Example Questions**:
1. What are the module level function calls in the file "main.js"?
2. What are the function calls inside the "main.return_func" function in the file "main.
    js"?
3. What are the function calls inside the "main.func" function in the file "main.js"?

**Example Answers**:
1. main.func, main.return_func
2. main.func
3.

**{language} Code Provided**:

{code}

**Questions**:
{questions}

**Answers**:
{answers}"""
```

Listing A.5: Prompt for call graph task in JavaScript in question-answer format (Part 2 of 2)
```

~~~
**Format for Response**:
    1. vc.Class:targetFunction(), vc.Class:targetFunction2()
    2. vc.Class:anotherTargetFunction()

**Example Java Code**:
```java
// vc/Class.java
package vc;

class Class {{

    public void target(){{ }}

    public static void main(String[] args){{
        Class cls = new Class();
        cls.target();
    }}
}}
~~~

Example Questions:
1. What are the target functions invoked by the caller "main(java.lang.String[])" in the
    "vc/Class" class?

Example Answers:
1. vc.Class:target()

{language} Code Provided:

{code}

Questions:
{questions}

Answers:
{answers}"""
```

Listing A.6: Prompt for call graph task in Java in question-answer format (Part 2 of 2)
```

~~~
## Task Description

**Objective**: Examine and identify the function calls in the given Python code and
    answer the questions.

**Instructions**:
1. For each question below, provide a concise answer indicating the function calls.
2. List every function call as a comma separated list.
3. Do not include additional explanations or commentary in your answers.
4. Include both explicit and implicit function calls in your answers. An implicit
    function call is a function that is called as a result of another operation, such as
    the __init__ method being called when an object is created.
5. If a function is called through an alias or a reference, identify and list the actual
    function that is called after resolving the alias.
6. If a passed argument is not invoked within the function, do not include the function
    call in the answer.
7. Examples of Python code, questions, and answers are given below. This example should
    be used as training data.

**Format for Answers**:
- Provide your answer next to each question number, using only one word.
- Do not include the questions in your answer.
- Example:
    1. simple.func
    2. simple.examplefunc
    3. func

**Example Python Code**:
```main.py
def return_func():
    func()

def func():
    a = return_func
    return a

a = func
a()()
~~~

**Example Questions**:
1. What are the module level function calls in the file "main.py"?
2. What are the function calls inside the "main.return_func" function in the file "main.
    py"?
3. What are the function calls inside the "main.func" function in the file "main.py"?

**Example Answers**:
1. main.func, main.return_func
2. main.func
3.

**Python Code Provided**:
```main.py
def param_func():
    pass

def func(a):
    a()

func(param_func)
```

Questions:
1. What are the module level function calls in the file 'main.py'?
2. What are the function calls inside the 'main.param_func' function in the file 'main.py
    '?
3. What are the function calls inside the 'main.func' function in the file 'main.py'?

Answers:
1.
2.
3.
```

Listing A.7: Example of an actual full prompt for call graph task in Python used in the study
```

~~~
## Task Description

**Objective**: Examine and identify the function calls in the given JavaScript code and
    answer the questions.

**Instructions**:
1. For each question below, provide a concise answer indicating the function calls.
2. List every function call as a comma separated list.
3. Do not include additional explanations or commentary in your answers.
4. Include both explicit and implicit function calls in your answers. An implicit
    function call is a function that is called as a result of another operation, such as
    the constructor method being called when an object is created.
5. If a function is called through an alias or a reference, identify and list the actual
    function that is called after resolving the alias.
6. If a passed argument is not invoked within the function, do not include the function
    call in the answer.
7. Example of JavaScript code, questions, and answers are given below. This example
    should be used as training data.

**Format for Answers**:
- Provide your answer next to each question number, using only one word.
- Do not include the questions in your answer.
- Example:
    1. simple.func
    2. simple.examplefunc
    3. func

**Example Javascript Code**:
```main.js
function returnFunc() {{
    func();
}}

function func() {{
    a = returnFunc;
    return a;
}}

a = func;
a()();
~~~

Example Questions:
1. What are the module level function calls in the file "main.js"?
2. What are the function calls inside the "main.return_func" function in the file "main.
    js"?
3. What are the function calls inside the "main.func" function in the file "main.js"?

Example Answers:
1. main.func, main.return_func
2. main.func
3.

JavaScript Code Provided:
```
function func() {}

func();
```

Questions:
1. What are the module level function calls in the file "main.js"?
2. What are the function calls inside the "main.func" function in the file "main.js"?

Answers:
1.
2.
```

Listing A.8: Example of an actual full prompt for call graph task in JavaScript used in the study
```

~~~
## Task Description

**Objective**: Examine the provided Java code, analyze the function calls, and answer the
     questions about the caller-target relationships.

**Instructions**:
1. For each question provide the target functions as a comma-separated list.
2. **IMPORTANT**: Ensure that the target function in each response is formatted as
    specified in the method signature format:
   - [Fullname of Class]:[method name]([Parameterlist])
   - Example: vc.Class:target(java.lang.String[])
3. Do not include additional explanations or comments in your answers.
4. Do not include the questions in your response. Only list the answers using the
    question number format shown below, that is, provide your answer for each question
    next to the corresponding question number
   - Response format example:
   - 1. vc.Class:targetFunction(), vc.Class:targetFunction2()
5. For methods called through aliases or references, resolve and return the actual target
     method.
6. Example of Java code, questions, and answers are given below. This example should be
    used as training data.

**Format for Response**:
    1. vc.Class:targetFunction(), vc.Class:targetFunction2()
    2. vc.Class:anotherTargetFunction()

**Example Java Code**:
```java
// vc/Class.java
package vc;
class Class {{
    public void target(){{ }}
    public static void main(String[] args){{
        Class cls = new Class();
        cls.target();
    }}
}}
~~~

**Example Questions**:
1. What are the target functions invoked by the caller "main(java.lang.String[])" in the
    "vc/Class" class?

**Example Answers**:
1. vc.Class:target()

**Java Code Provided**:
```
// simplecast/Demo.java
package simplecast;
class Demo {
    public static void main(String[] args) throws Exception {
        if (args.length == 0)
          castToTarget(new Target());
        else
          castToTarget(new Demo());
    }
    static void castToTarget(Object o) {
        Target b = (Target) o;
        b.target();
    }
    public String target() { return "Demo"; }
}
class Target {
  public String target() { return "Target"; }
}
```

Questions:
1. What are the target functions invoked by simplecast.Demo:castToTarget(java.lang.Object
) in the simplecast.Demo class?

Answers:
1.
```

Listing A.9: Example of an actual full prompt for call graph task in Java used in the study
```

A.2 Example Responses

A.2.1 Callgraph output generated by gpt-4o model for the Python test case args/assign_return

```python
def return_func(args1):
    return args1


def func1():
    pass


def func2():
    pass

return_func(func1)
return_func(func2)()
```

Listing A.10: Python Source code

```json
{
  "main": [
    "main.return_func",
    "main.func2"
  ],
  "main.func1": [],
  "main.func2": [],
  "main.return_func": []
}
```

Listing A.11: Ground truth

```
1. main.return_func, main.func2
2.
3.
4.
```

Listing A.12: Raw LLM response

```json
{
  "main": [
      "main.return_func",
      "main.func2"
  ],
  "main.func1": [],
  "main.func2": [],
  "main.return_func": []
}
```

Listing A.13: Parsed JSON response

A.2.2 Callgraph output generated by the phi3.5-mini-it-3.8b model for the Python test case args/assign_return

```python
def return_func(args1):
    return args1


def func1():
    pass


def func2():
    pass

return_func(func1)
return_func(func2)()
```

Listing A.14: Python Source code

```json
{
  "main": [
    "main.return_func",
    "main.func2"
  ],
  "main.func1": [],
  "main.func2": [],
  "main.return_func": []
}
```

Listing A.15: Ground truth

```
1. func1, func2
2. None
3. return_func
4. return_func
```

Listing A.16: Raw LLM response

```
{
    "main": [
        "func1",
        "func2"
    ],
    "main.func1": [
        "None"
    ],
    "main.func2": [
        "return_func"
    ],
    "main.return_func": [
        "return_func"
    ]
}
```

Listing A.17: Parsed JSON response

A.2.3 Callgraph output generated by gpt-4o model for the JavaScript test case functions/call

```
function func() {}

func();
```

Listing A.18: JavaScript Source code

```
{
    "main.func": [],
    "main": [
        "main.func"
    ]
}
```

Listing A.19: Ground truth

```
1.
2. main.func
```

Listing A.20: Raw LLM response

```
{
    "main.func": [],
    "main": [
        "main.func"
    ]
}
```

Listing A.21: Parsed JSON response

A.2.4 Callgraph output generated by the mixtral-v0.1-it-8x7b model for the JavaScript test case functions/call

```
function func() {}

func();
```

Listing A.22: JavaScript Source code

```
{
    "main.func": [],
    "main": [
        "main.func"
    ]
}
```

Listing A.23: Ground truth

```
1. (No function calls inside the'main
   .func' function)
2. func
```

Listing A.24: Raw LLM response

```
{
    "main.func": [
        "(No function calls inside the'main.
          func' function)"
    ],
    "main": [
        "func"
    ]
}
```

Listing A.25: Parsed JSON response

A.2.5 Callgraph output generated by gpt-4o model for the Java test case VirtualCalls/vc1

```java
// vc/Class.java
package vc;
class Class {
    public void target(){ }
    public static void main(String[] args
   ){
        Class cls = new Class();
        cls.target();
    }
}
```

Listing A.26: Java Source code

```json
{
    "vc.Class:main(java.lang.String[])":
        [
        "vc.Class:target()"
    ]
}
```

Listing A.27: Ground truth

```
1. vc.Class:target()
```

Listing A.28: Raw LLM response

```json
{
    "vc.Class:main(java.lang.String[])":
        [
        "vc.Class:target()"
    ]
}
```

Listing A.29: Parsed JSON response

A.2.6 Callgraph output generated by the llama-3.2-1B-Instruct model for the Java test case VirtualCalls/vc1

```java
// vc/Class.java
package vc;
class Class {
    public void target(){ }
    public static void main(String[]
   args){
        Class cls = new Class();
        cls.target();
    }
}
```

Listing A.30: Java Source code

```json
{
    "vc.Class:main(java.lang.String[])": [
        "vc.Class:target()"
    ]
}
```

Listing A.31: Ground truth

```
1. vc.Class:target(), vc.Class:target
   2()
2. vc.Class:anotherTargetFunction()
```

Listing A.32: Raw LLM response

```json
{
    "vc.Class:main(java.lang.String[])": [
        "vc.Class:target()",
        "vc.Class:target2()"
    ]
}
```

Listing A.33: Parsed JSON response

Bibliography

Mistral AI. 2024. Large Enough. `https://mistral.ai/news/mistral-large-2407/` Section: news.

Karim Ali and Ondřej Lhoták. 2012. Application-Only Call Graph Construction. In *ECOOP 2012 – Object-Oriented Programming*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and James Noble (Eds.). Vol. 7313. Springer Berlin Heidelberg, Berlin, Heidelberg, 688–712. `https://doi.org/10.1007/978-3-642-31057-7_30` Series Title: Lecture Notes in Computer Science.

Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. 2023. Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools. *IEEE Access* 11 (2023), 25266–25284. `https://doi.org/10.1109/ACCESS.2023.3255984`

Alessio Buscemi. 2023. A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages. `http://arxiv.org/abs/2308.04477` arXiv:2308.04477 [cs].

Yu-Chu Chang, Xu Wang, Jindong Wang, Yuanyi Wu, Kaijie Zhu, Hao Chen, Linyi Yang, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Weirong Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qian Yang, and Xingxu Xie. 2023. A Survey on Evaluation of Large Language Models. *ArXiv* abs/2307.03109 (2023). `https://api.semanticscholar.org/CorpusID:259360395`

Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. 2024. Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review. `http://arxiv.org/abs/2310.14735` arXiv:2310.14735.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] `https://arxiv.org/abs/2107.03374`

Xuliang Chen, Jianhui Jiang, Wei Zhang, and Xuzei Xia. 2020. Fault Diagnosis for Open Source Software Based on Dynamic Tracking. In *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 263–268. https://doi.org/10.1109/DSA51864.2020.00047

B. Chess and G. McGraw. 2004. Static analysis for security. *IEEE Security & Privacy* 2, 6 (Nov. 2004), 76–79. https://doi.org/10.1109/MSP.2004.111 Conference Name: IEEE Security & Privacy.

ECMA. 2015. ECMA-262. https://ecma-international.org/publications-and-standards/standards/ecma-262/

Gang Fan, Xiaoheng Xie, Xunjin Zheng, Yinan Liang, and Peng Di. 2023. Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents. http://arxiv.org/abs/2310.08837 arXiv:2310.08837.

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *2013 35th International Conference on Software Engineering (ICSE)*. 752–761. https://doi.org/10.1109/ICSE.2013.6606621 ISSN: 1558-1225.

Google. 2023. Bard. https://bard.google.com/.

J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith. 2018. *The Java Virtual Machine Specification*. Oracle America, United States.

Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). 2015. *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*. Lecture Notes in Computer Science, Vol. 9058. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-17524-9

Tobias Hey, Jan Keim, Anne Koziolek, and Walter F. Tichy. 2020. NoRBERT: Transfer Learning for Requirements Classification. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*. 169–179. https://doi.org/10.1109/RE48521.2020.00028 ISSN: 2332-6441.

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. http://arxiv.org/abs/2308.10620 arXiv:2308.10620 [cs].

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* (Sept. 2024), 3695988. https://doi.org/10.1145/3695988

Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, Chicago Illinois USA, 611–620. https://doi.org/10.1145/1653662.1653736

Kaifeng Huang, Yixuan Yan, Bihuan Chen, Zixin Tao, and Xin Peng. 2024. Scalable and Precise Application-Centered Call Graph Construction for Python. http://arxiv.org/abs/2305.05949 arXiv:2305.05949.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis*, Jens Palsberg and Zhendong Su (Eds.). Vol. 5673. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17 Series Title: Lecture Notes in Computer Science.

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. https://doi.org/10.48550/arXiv.2209.11515 arXiv:2209.11515 [cs].

Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. 2024. SootUp: A Redesign of the Soot Static Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 229–247.

Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2024. Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. https://doi.org/10.48550/arXiv.2311.16169 arXiv:2311.16169 [cs].

Takashi Koide, Naoki Fukushi, Hiroki Nakano, and Daiki Chiba. 2024. Detecting Phishing Sites Using ChatGPT. https://doi.org/10.48550/arXiv.2306.05816 arXiv:2306.05816.

Márk Lajkó, Viktor Csuvik, and László Vidács. 2022. Towards JavaScript program repair with Generative Pre-trained Transformer (GPT-2). In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 61–68. https://doi.org/10.1145/3524459.3527350

Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. 2024. Reducing Static Analysis Unsoundness with Approximate Interpretation. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 1165–1188. https://doi.org/10.1145/3656424

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 919–931. https://doi.org/10.1109/ICSE48619.2023.00085 ISSN: 1558-1225.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023a. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 2107–2111. https://doi.org/10.1145/3611643.3613078

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023b. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. http://arxiv.org/abs/2308.00245 arXiv:2308.00245.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (April 2024), 474–499. https://doi.org/10.1145/3649828

Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT Prompt for Code Generation. *ArXiv* abs/2305.08360 (2023). https://api.semanticscholar.org/CorpusID:258685413

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *Comput. Surveys* 55 (2021), 1 – 35. https://api.semanticscholar.org/CorpusID:236493269

Vivian Liu and Lydia B. Chilton. 2021. Design Guidelines for Prompt Engineering Text-to-Image Generative Models. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (2021). https://api.semanticscholar.org/CorpusID:237513697

V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (2005).

Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2024. LMs: Understanding Code Syntax and Semantics for Code Analysis. http://arxiv.org/abs/2305.12138 arXiv:2305.12138.

Wei Ma, Shangqing Liu, Wang Wenhan, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The Scope of ChatGPT in Software Engineering: A Thorough Investigation.

Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2023. SkipAnalyzer: A Tool for Static Code Analysis with Large Language Models. http://arxiv.org/abs/2310.18532 arXiv:2310.18532.

Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2024. Effectiveness of ChatGPT for Static Analysis: How Far Are We?. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*. ACM, Porto de Galinhas Brazil, 151–160. https://doi.org/10.1145/3664646.3664777

Austin Mordahl. 2023. Automatic Testing and Benchmarking for Configurable Static Analysis Tools. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Seattle WA USA, 1532–1536. https://doi.org/10.1145/3597926.3605232

Jernej Novak, Andrej Krajnc, and Rok Žontar. 2010. Taxonomy of static code analysis tools. In *The 33rd International Convention MIPRO*. 418–422. https://ieeexplore.ieee.org/document/5533417/?arnumber=5533417

OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://chat.openai.com.

Ipek Ozkaya. 2023. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Software* 40, 3 (May 2023), 4–8. https://doi.org/10.1109/MS.2023.3248401 Conference Name: IEEE Software.

Omer Said Ozturk, Emre Ekmekcioglu, Orcun Cetin, Budi Arief, and Julio Hernandez-Castro. 2023. New Tricks to Old Codes: Can AI Chatbots Replace Static Code Analysis Tools?. In *European Interdisciplinary Cybersecurity Conference*. ACM, Stavanger Norway, 13–18. https://doi.org/10.1145/3590777.3590780

Jihyeok Park, Hongki Lee, and Sukyoung Ryu. 2022. A Survey of Parametric Static Analysis. *Comput. Surveys* 54, 7 (Sept. 2022), 1–37. https://doi.org/10.1145/3464457

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 2339–2356. https://doi.org/10.1109/SP46215.2023.10179324

Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Jana. 2024. Exploiting Code Symmetries for Learning Program Semantics. https://doi.org/10.48550/arXiv.2308.03312 arXiv:2308.03312 [cs].

Michael Reif. 2021. *Novel Approaches to Systematically Evaluating and Constructing Call Graphs for Java Software.* Ph.D. Thesis. Technische Universität Darmstadt, Darmstadt. https://doi.org/10.26083/tuprints-00019286

Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, 251–261. https://doi.org/10.1145/3293882.3330555

Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146 ISSN: 1558-1225.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. 2024. The Prompt Report: A Systematic Survey of Prompting Techniques. http://arxiv.org/abs/2406.06608 arXiv:2406.06608.

Lukas Seidel, Sedick David Baker Effendi, Xavier Pinho, Konrad Rieck, Brink van der Merwe, and Fabian Yamaguchi. 2023. Learning Type Inference for Enhanced Dataflow Analysis. http://arxiv.org/abs/2310.00673 arXiv:2310.00673.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M. Sadler, Wei-Lun Chao, and Yu Su. 2023. LLM-Planner: Few-Shot Grounded Planning for Embodied Agents with Large Language Models. http://arxiv.org/abs/2212.04088 arXiv:2212.04088.

Giriprasad Sridhara, Ranjani H. G, and Sourav Mazumdar. 2023. ChatGPT: A Study on its Utility for Ubiquitous Software Engineering Tasks. https://doi.org/10.48550/arXiv.2305.16837 arXiv:2305.16837.

Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? http://arxiv.org/abs/2305.12865 arXiv:2305.12865.

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. https://doi.org/10.1145/3597503.3639117 arXiv:2308.03314 [cs].

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *ArXiv* abs/2302.13971 (2023). `https://api.semanticscholar.org/CorpusID:257219404`

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2000. Soot: a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers on - CASCON '10*. ACM Press, Toronto, Ontario, Canada, 214–224. `https://doi.org/10.1145/1925805.1925818`

Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Mouli Chekkapalli, Jiawei Wang, Li Li, and Eric Bodden. 2024a. Static Analysis Driven Enhancements for Comprehension in Machine Learning Notebooks. `https://doi.org/10.48550/arXiv.2301.04419` arXiv:2301.04419.

Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Amir M. Mir, Sofia Reis, and Eric Bodden. 2024b. The Emergence of Large Language Models in Static Analysis: A First Look through Micro-Benchmarks. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. ACM, Lisbon Portugal, 35–39. `https://doi.org/10.1145/3650105.3652288`

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. `http://arxiv.org/abs/2305.16291` arXiv:2305.16291.

WebKit. 2010. WebKit/PerformanceTests/SunSpider/tests/sunspider-1.0.2 at main · WebKit/WebKit. `https://github.com/WebKit/WebKit/tree/main/PerformanceTests/SunSpider/tests/sunspider-1.0.2`

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. *ArXiv* abs/2302.11382 (2023). `https://api.semanticscholar.org/CorpusID:257079092`

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, Online, 38–45. `https://doi.org/10.18653/v1/2020.emnlp-demos.6`

Chunqiu Steven Xia and Lingming Zhang. 2024. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831. `https://doi.org/10.1145/3650212.3680323` arXiv:2304.00385 [cs].

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large Language Models Meet NL2Code: A Survey. (Dec. 2022).

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng

Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. `http://arxiv.org/abs/2303.18223` arXiv:2303.18223 [cs].

Noah Ziems and Shaoen Wu. 2021. Security Vulnerability Detection Using Deep Learning Natural Language Processing. `http://arxiv.org/abs/2105.02388` arXiv:2105.02388.