The Emergence of Large Language Models in Static Analysis: A First Look through Micro-benchmarks

Ashwin Prasad Shivarpatna Venkatesh[§], Samkutty Sabu[¶], Amir M. Mir[‡], Sofia Reis[†], Eric Bodden[**]
[§] *ashwin.prasad@upb.de*, Heinz Nixdorf Institut, Paderborn University, Paderborn, Germany
[¶] *samkutty@mail.uni-paderborn.de*, Paderborn University, Paderborn, Germany
[‡] *s.a.m.mir@tudelft.nl*, Delft University of Technology, Delft, The Netherlands
[†] *sofia.o.reis@tecnico.ulisboa.pt*, IST, University of Lisbon & INESC-ID, Lisbon, Portugal
[**] *eric.bodden@upb.de*, Heinz Nixdorf Institut & Fraunhofer IEM, Paderborn University, Paderborn, Germany

ABSTRACT

The application of Large Language Models (LLMs) in software engineering, particularly in static analysis tasks, represents a paradigm shift in the field. In this paper, we investigate the role that current LLMs can play in improving callgraph analysis and type inference for Python programs. Using the PyCG, HeaderGen, and TypeEvalPy micro-benchmarks, we evaluate 26 LLMs, including OpenAI's GPT series and open-source models such as LLaMA. Our study reveals that LLMs show promising results in type inference, demonstrating higher accuracy than traditional methods, yet they exhibit limitations in callgraph analysis. This contrast emphasizes the need for specialized fine-tuning of LLMs to better suit specific static analysis tasks. Our findings provide a foundation for further research towards integrating LLMs for static analysis tasks.

ACM Reference Format:
Ashwin Prasad Shivarpatna Venkatesh[§], Samkutty Sabu[¶], Amir M. Mir[‡], Sofia Reis[†], Eric Bodden[**]. 2024. The Emergence of Large Language Models in Static Analysis: A First Look through Micro-benchmarks. In *AI Foundation Models and Software Engineering (FORGE '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650105.3652288

1 INTRODUCTION

In the dynamic field of Software Engineering (SE), the incorporation of advanced computational models, especially Large Language Models (LLMs), marks a significant shift in the software development processes [8, 9, 22, 23]. Static analysis (SA), an integral component of SE, involves examining source code without executing it, to identify potential errors, code quality issues, and security vulnerabilities. The emergence of LLMs, such as BERT [6], T5 [16], and GPT [15], has transformed several diverse SE tasks, including the SA tasks [22]. Recent works have shown how different SA tasks can benefit from LLMs, such as false-positives pruning [10], improved program behavior summarization [11], type annotation [18], and general enhancements in precision and scalability of SA tasks [11], both fundamental issues of SA.

This study here situates itself at the intersection of SA and LLMs, focusing on the effectiveness of LLMs in SA within SE. It aims to evaluate the accuracy of LLMs in performing specific SA tasks: callgraph analysis and type inference, specifically in Python programs. *Callgraph analysis* helps in understanding the relationships and interactions between different components of a program, while *type inference* aids in identifying potential type errors and improving code reliability. To assess the performance of LLMs in these areas, we use the PyCG [17] and HeaderGen [21] micro-benchmarks for callgraph analysis, and TypeEvalPy [20] for type inference.

The use of micro-benchmarks in evaluating the performance of LLMs in our study is grounded in several key considerations. Firstly, micro-benchmarks are designed to target specific aspects of the features under test and various characteristics of the programming language involved. This helps in highlighting the models' strengths and weaknesses, allowing for a more nuanced understanding of their capabilities in SA tasks. Additionally, their development involves rigorous manual inspection and adherence to scientific methods, ensuring reliability and accuracy in evaluation. 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 [7].

By testing a range of 26 different LLMs, our study provides a comprehensive analysis of their capabilities in the context of SA. Furthermore, the evaluation enables one to make direct comparisons with the existing capabilities of traditional approaches in SA. The insights from this study are intended to offer a preliminary understanding of the role of LLMs in SA, AI4SE, and SE4AI fields.

The structure of the paper is as follows: in Section 2 we discuss the related work. The research questions are outlined in Section 3, while Section 4 describes our methodology. Results are presented in Section 5 and subsequently discussed in Section 6. Section 7 addresses the threats to validity. Finally, the paper is concluded by outlining future research directions in section 8.

Availability. TypeEvalPy is published on GitHub as open-source software: https://github.com/secure-software-engineering/TypeEvalPy

2 RELATED WORK

Ma et al. [12] and Sun et al. [19] explore the capabilities of LLMs when performing different program analysis tasks such as control-flow graph construction, callgraph analysis, and code summarization. They conclude that while LLMs can comprehend basic code syntax, they are somewhat limited in performing more sophisticated analyses, such as pointer analysis and code behavior summarization. In contrast, LLift, an LLM-based approach, showed successful results for different programming analysis tasks, including program

FORGE '24, April 14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0609-7/24/04
https://doi.org/10.1145/3650105.3652288

behavior summarization [11] and how LLMs can be successfully integrated into an SA pipeline. Researchers conjecture that the reasons behind the difference in the results were benchmark selection, prompt designs, and model versions. Li et al. [10] present a solution to prune SA false positives by asking carefully constructed questions about function-level behaviors or function summaries. Seidel et al. [18] propose CodeTIDAL5, a Transformer-based model trained to predict type annotations in TypeScript. In this study, we explore how different LLMs perform on callgraph analysis and type inference for Python programs.

3 RESEARCH QUESTIONS

We focus on the following research questions to evaluate the effectiveness of LLMs using micro-benchmarks in static analysis tasks:

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

RQ2: *What is the accuracy of LLMs in performing type inference against micro-benchmarks?*

4 METHODOLOGY

We next describe the experimental setup, the model selection criteria, prompt design, and metrics used to investigate these RQs.

Micro-benchmarks. To answer RQ1, we choose two benchmarks designed to evaluate callgraph analysis performance, PyCG [17] and HeaderGen [21]. PyCG is the first callgraph construction algorithm that uses a context-*insensitive* and flow-*insensitive* SA as its backend. PyCG includes a micro-benchmark containing 112 unique python programs targeting various Python features organized into 16 categories. HeaderGen is a tool that uses SA to enhance comprehension in computational notebooks. HeaderGen improves PyCG's static analyzer with flow-sensitivity and type inference. HeaderGen includes a micro-benchmark with 121 code snippets with flow-sensitive call sites as ground truth. Note that for this study we have extended PyCG's micro-benchmark with nine additional snippets from the HeaderGen micro-benchmark.

To answer RQ2, we choose the micro-benchmark from TypeEvalPy [20], a general framework for evaluating type inference tools in Python. TypeEvalPy contains a micro-benchmark with 154 code snippets and 845 type annotations as ground truth.

Model Selection. In this study, we evaluate several state-of-the-art LLMs. First, we include two closed-source LLMs, GPT 3.5 Turbo and GPT 4 from OpenAI as it is the leading general-purpose LLM. Furthermore, we include ten popular open-source models based on the download count on the Huggingface [1] platform. This includes llama2, mistral, dolphin-mistral, codellama, codebooga, tinyllama, vicuna, wizardcoder, and orca. We include several variations of these models such as the number of parameters (7b, 13b, etc.,). Overall, we evaluate 24 open-source models and two closed-source models, totaling 26 LLMs.

Furthermore, we create a fine-tuned version of GPT-3.5 Turbo, refined with a training dataset. The dataset created for fine-tuning GPT-3.5 Turbo comprises 15 program categories. It serves as the representative collection of the PyCG, HeaderGen, and TypeEvalPy micro-benchmarks, emphasizing key Python features such as functions, classes, decorators, and exceptions. This approach seeks to enhance the model's adaptability, equipping it to effectively handle a diverse range of challenges.

Prompt Design. To optimize prompt design, we adopted an iterative and experimental approach [5]. Initial efforts focused on enhancing the prompt by including detailed task descriptions and specifying the expected response format. Notably, 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 back into a structured JSON format. Note that the same prompt is used for all models and set up with a temperature of zero to ensure deterministic outputs.

Evaluation Metrics. To assess both flow-insensitive callgraph construction and flow-sensitive call-site extraction, in this study, we measured completeness, soundness, and exact matches. Completeness is the absence of false positives in the callgraph, ensuring that no call edges were included if they did not exist. Soundness, conversely, focuses on the inclusion of every call edge, thereby avoiding any false negatives. Exact matches is measured as the number of function calls that exactly match the ground truth. This evaluation approach mirrors the methodologies used in previous studies, specifically in PyCG [17] and HeaderGen [21]. Furthermore, aligning with the literature [4, 13, 14, 20], for type-inference evaluation we use exact matches as the metric. Additionally, the total runtime of these tools for analyzing the respective micro-benchmark is also included by computing the mean over three runs.

Implementation Details. In the implementation of our experiments with LLMs, we employed Ollama [3], an open-source platform that simplifies running LLMs by providing an efficient HTTP server for lifecycle management. This served as our backend infrastructure. In addition, to create a pipeline for efficient prompting and response handling, we used LangChain [2], a framework designed for building applications that interact with LLMs. Additionally, to implement the type-inference experiments, we extended the TypeEvalPy framework [20], due to its flexibility in adding support for new tools.

5 RESULTS

We next address the research questions and highlight key results.

5.1 RQ1: Accuracy of Callgraph Analysis

Table 1 presents the outcomes of our experiments using LLMs on the flow-insensitive callgraph analysis evaluation micro-benchmark of PyCG, and the flow-sensitive callgraph analysis evaluation micro-benchmark of HeaderGen.

Flow-insensitive Callgraph analysis. The static analysis algorithm PyCG demonstrated superior performance over LLMs in terms of completeness, soundness, exact matches, and processing time. In a benchmark of 121 test cases, PyCG achieved 93.3% completeness and 86.7% soundness, meaning it had no false positives or false negatives, respectively. This significantly surpasses that of the nearest competitor, ft:gpt-3.5-turbo, which achieved 57.8% in

Table 1: Comparative analysis across LLMs for callgraph analysis on PyCG and HeaderGen micro-benchmarks

PyCG Benchmark — Flow-insensitive Callgraphs				
Model	**Complete**	**Sound**	**E.M**	**Time**
PyCG	**113**	**105**	**250**	**0.41**
ft:gpt-3.5-turbo	70	75	207	77.96
gpt-4	59	54	180	264.96
codebooga	22	44	140	462.00
phind-codellama:34b-v2	77	21	70	696.01
wizardcoder:7b-python	21	18	60	157.11
wizardcoder:34b-python	76	13	45	847.04
codellama:34b-instruct	2	7	40	**1644.51**
gpt-3.5-turbo	14	20	40	124.58
orca2:13b	14	20	39	386.47
codellama:13b-instruct	1	11	35	280.62
wizardcoder:13b-python	6	9	29	232.74
mistral:instruct	12	7	28	188.85
mistral:v0.2	12	6	28	185.95
dolphin-mistral	15	9	21	158.11
codellama:7b-instruct	1	6	16	276.94
tinyllama	28	8	13	889.33
orca2:7b	106	7	10	336.35
vicuna:13b	3	9	9	**2383.42**
vicuna:7b	1	8	8	147.05
vicuna:33b	0	6	6	478.70
llama2:70b	0	6	1	**1398.26**
codellama:13b-python	121	0	0	142.09
codellama:34b-python	121	0	0	269.77
codellama:7b-python	121	0	0	92.92
llama2:13b	14	6	0	587.44
llama2:7b	93	0	0	**1825.59**
phind-codellama:34b-python	121	0	0	267.63

HeaderGen Benchmark — Flow-sensitive Callgraphs				
Model	**Complete**	**Sound**	**E.M**	**Time**
HeaderGen	**111**	**113**	**327**	**5.26**
ft:gpt-3.5-turbo	47	48	149	79.98
gpt-4	27	17	70	248.37
gpt-3.5-turbo	17	16	53	160.14
phind-codellama:34b-v2	13	12	42	475.08
vicuna:13b	14	12	27	184.76
wizardcoder:34b-python	8	8	17	360.71
wizardcoder:13b-python	12	9	15	195.17
vicuna:33b	8	7	14	365.01
vicuna:7b	7	6	13	130.42
llama2:7b	10	8	12	128.40
codebooga	6	8	11	357.18
mistral:instruct	7	7	10	191.66
tinyllama	10	6	10	355.92
codellama:34b-python	15	10	9	270.90
dolphin-mistral	11	11	9	129.36
mistral:v0.2	7	7	7	196.42
phind-codellama:34b-python	11	8	7	319.54
wizardcoder:7b-python	8	8	7	135.16
codellama:13b-python	12	9	6	181.04
codellama:7b-python	11	9	5	300.54
orca2:13b	6	6	3	369.16
codellama:7b-instruct	7	6	2	148.13
llama2:70b	6	6	1	**944.54**
codellama:13b-instruct	6	6	0	188.70
codellama:34b-instruct	121	6	0	347.79
llama2:13b	6	6	0	426.48
orca2:7b	6	6	0	224.57

completeness and 61.9% in soundness. Furthermore, PyCG obtained 250 exact matches (out of 284), which is 43 more exact matches than ft:gpt-3.5-turbo. This performance difference is further emphasized in running times, where PyCG processed flow-insensitive callgraphs 190 times faster than ft:gpt-3.5-turbo. Among the LLMs, the best-performing one without fine-tuning is gpt-4; however, the fine-tuned gpt-3.5-turbo model surpasses the vanilla gpt-4, indicating the potential benefits of fine-tuning LLMs for specific applications. Yet, other open-source models lagged significantly in performance. Notably, due to their failure to produce structured outputs in line with our prompts, some LLMs like codellama:34b-instruct, vicuna:13b, llama2:70b, and llama2:7b experienced lengthy running times. Despite clear instructions regarding the output format and the instruction to avoid explanatory content, they sometimes continued to generate irrelevant content and timed out.

Flow-sensitive Callgraph analysis. Here, HeaderGen demonstrated superior performance over LLMs across all evaluated metrics. In particular, HeaderGen achieved 91.7% completeness and 93.3% soundness, which is more than double the performance of its closest LLM competitor, ft:gpt-3.5-turbo, which managed only 38.8% completeness and 39.6% soundness. In terms of exact matches, HeaderGen identified 327 out of 355 call sites, surpassing the best-performing LLM by 178 matches. Moreover, HeaderGen's runtime is 15 times shorter than the fastest LLM in analyzing the entire benchmark. Note that LLMs fared considerably poorer in flow-sensitive analysis compared to flow-insensitive analysis, likely due to the increased complexity and the requirement for precise flow-sensitive pointer information, which may pose challenges to LLMs. And this although in the prompt we did provide specific instructions to ensure the LLMs' awareness of the flow-sensitive aspects.

5.2 RQ2: Accuracy of Type Inference

Table 2 shows the performance of LLMs, HeaderGen, and HiTyper considering the exact-match performance. Note that HiTyper is configured with Type4Py [13]. In general, LLMs significantly here outperform the current state-of-the-art approaches for type inference, namely, HeaderGen and HiTyper models. Specifically, OpenAI's GPT-4 is the best-performing model, correctly inferring 775 of 845 type annotations in the micro-benchmark. This is expected, as GPT-4 is one of the most powerful LLMs in the wild, though it can be slow and expensive to run. It is also interesting to see

Table 2: Exact match comparison of LLMs in type inference

FRT: Function return type, **FPT:** Function parameter type, **LVT:** Local variable type

Model	FRT	FPT	LVT	Total	Time (s)
gpt-4	225	85	465	775	454.54
ft:gpt-3.5-turbo	209	85	436	730	110.45
codellama:13b-instruct	199	75	425	699	221.77
gpt-3.5-turbo	188	73	429	690	167.77
codellama:34b-instruct	190	52	425	667	402.89
phind-codellama:34b-v2	182	60	399	641	488.27
codellama:7b-instruct	171	72	384	627	147.78
dolphin-mistral	184	76	356	616	162.38
codebooga	186	56	354	596	473.76
llama2:70b	168	55	342	565	790.84
HeaderGen	**186**	**56**	**321**	**563**	**18.25**
wizardcoder:13b-python	170	74	317	561	234.14
llama2:13b	153	40	283	476	266.59
mistral:instruct	155	45	250	450	203.78
mistral:v0.2	155	45	248	448	204.60
vicuna:13b	153	35	260	448	252.45
vicuna:33b	133	29	267	429	434.82
wizardcoder:7b-python	103	48	254	405	156.62
llama2:7b	140	34	216	390	146.14
HiTyper	**163**	**27**	**179**	**369**	**268.40**
wizardcoder:34b-python	140	43	178	361	463.05
orca2:7b	117	27	184	328	215.53
vicuna:7b	131	17	172	320	154.28
orca2:13b	113	19	166	298	397.66
tinyllama	3	0	23	26	232.67
phind-codellama:34b-python	5	0	15	20	407.20
codellama:13b-python	0	0	0	0	147.21
codellama:34b-python	0	0	0	0	305.74
codellama:7b-python	0	0	0	0	243.01

that the fine-tuned version of GPT 3.5 Turbo is the second best-performing model with 730 correctly inferred type annotations and an inference speed 4 times faster than that of GPT 4. Considering open-source LLMs, with 699 correctly inferred annotations CodeLlama (13B-instruct) has comparable performance to GPT-4 and the fine-tuned GPT 3.5. LLMs specialized in code-related tasks like CodeLLaMA outperform general-purpose LLMs such as vanilla LLaMA. Another observation is that TinyLlama, a 1.1B parameter model, performs poorly: it only infers 26 annotations correctly. It seems that models smaller than seven billion parameters, like TinyLlama, are insufficiently capable of the type inference task.

6 DISCUSSION

Similar to findings in previous work [12, 19], we observe that the construction of callgraphs does not yet significantly benefit from the use of LLMs. In comparison to LLMs, for this task traditional SA methods remain more efficient. However, fine-tuning GPT models showed promising improvements in callgraph analysis results, paving the way for future research in this direction.

In the type-inference tasks, LLMs such as gpt-4 and gpt-3.5, have demonstrated promising results, as evidenced in our study involving the TypeEvalPy framework. Nonetheless, in extensive Python projects using LLMs for type inference can be resource-intensive. Moreover, employing OpenAI's services incurs monetary costs and lacks privacy for proprietary projects. Open-source LLMs like CodeLLaMA avoid these problems as they are free and also offer the advantage of local deployment.

The LLMs tested in this study are predominantly large, having over seven billion parameters. This renders them unsuitable for deployment on standard machines equipped with a single GPU. In contrast, PyCG and HeaderGen, both traditional SA methods, are capable of operating well within such hardware constraints. Consequently, for SA tasks, traditional SA methods still yield the best trade-off between accuracy and speed. Nonetheless, as indicated by our findings related to type inference, where accuracy is paramount, LLMs can be effectively used, especially with fine-tuning.

7 THREATS TO VALIDITY

We list limitations and threats to the validity of our study as follows: (1) We only analyzed the source code of the main program, excluding the code of the imported modules in the prompt. This decision was due to the complexities of constructing a prompt that accounts for the diverse import statement variations. This particularly affects programs in the "`imports`" category of the TypeEvalPy, HeaderGen, and PyCG benchmarks. Despite this, the affected portion is relatively small (5.6% of the total facts), so the overall results are only insignificantly altered. For a more comprehensive analysis, future work should include imported files. (2) We used the same prompt for all models, which may not have extracted the best possible performance from each. (3) Open-source models often deviate from the required output formats. We addressed this by manually identifying response patterns and adding a preprocessing step for format standardization. However, this does not cover all possibilities. This issue further highlights the LLMs' inability to produce structured data consistently. (4) If the micro-benchmarks were included in the training set of the LLMs, data leakage could have skewed the results.

8 CONCLUSION

In this paper, we used micro-benchmarks to evaluate the application of LLMs in SA tasks on Python programs. Our findings reveal that LLMs demonstrate promising capabilities in type inference, often surpassing traditional techniques. GPT-4 stood out as the most effective model without fine-tuning, while fine-tuning GPT-3.5 Turbo yielded significant improvements. However, in the area of callgraph analysis, traditional methods still outperform LLMs, indicating a need for more focused fine-tuning and task-specific model adaptation. Notably, these advancements come with substantial computational and monetary requirements. To reduce LLM size and enhance inference speeds, future research should explore model compression techniques, such as quantization [24]. Further avenues of research include applying explainability methods to understand the challenges faced by LLMs in SA, expanding the scope to cover various SA tasks and programming languages, and evaluating the performance of fine-tuned open-source models.

ACKNOWLEDGMENTS

Funding for this study was provided by the Ministry of Culture and Science of the State of North Rhine-Westphalia under the SAIL project with the grand no NW21-059D.

REFERENCES

[1] [n. d.]. Hugging Face – The AI Community Building the Future. https://huggingface.co/.
[2] [n. d.]. Langchain-Ai/Langchain: Building Applications with LLMs through Composability. https://github.com/langchain-ai/langchain.
[3] [n. d.]. Ollama. https://ollama.ai.
[4] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3385412.3385997
[5] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. 2023. Unleashing the Potential of Prompt Engineering in Large Language Models: A Comprehensive Review. arXiv:2310.14735 [cs]
[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs]
[7] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3540250.3549114
[8] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. https://arxiv.org/abs/2310.03533v4.
[9] 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. https://doi.org/10.48550/arXiv.2308.10620 arXiv:2308.10620 [cs]
[10] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. 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* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2107–2111. https://doi.org/10.1145/3611643.3613078
[11] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. https://doi.org/10.48550/arXiv.2308.00245 arXiv:2308.00245 [cs]
[12] 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.
[13] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. https://doi.org/10.1145/3510003.3510124
[14] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2019–2030. https://doi.org/10.1145/3510003.3510038
[15] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. [n. d.]. Language Models Are Unsupervised Multitask Learners. ([n. d.]).
[16] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. https://doi.org/10.48550/arXiv.1910.10683 arXiv:1910.10683 [cs, stat]
[17] 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
[18] 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. arXiv:2310.00673 [cs.LG]
[19] 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? arXiv:2305.12865 [cs.SE]
[20] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Jiawei Wang, Amir M. Mir, Li Li, and Eric Bodden. 2023. TypeEvalPy: A Micro-benchmarking Framework for Python Type Inference Tools. https://doi.org/10.48550/arXiv.2312.16882 arXiv:2312.16882 [cs]
[21] Ashwin Prasad Shivarpatna Venkatesh, Jiawei Wang, Li Li, and Eric Bodden. 2023. Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 391–401. https://doi.org/10.1109/SANER56733.2023.00044
[22] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A Survey on Large Language Models for Software Engineering. https://doi.org/10.48550/arXiv.2312.15223 arXiv:2312.15223 [cs]
[23] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an Understanding of Large Language Models in Software Engineering Tasks. https://doi.org/10.48550/arXiv.2308.11396 arXiv:2308.11396 [cs]
[24] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. 2023. A Survey on Model Compression for Large Language Models. https://doi.org/10.48550/arXiv.2308.07633 arXiv:2308.07633 [cs]