LLM-Augmented Static Analysis Security Testing

Murilo Escher Pagotto Ronchi

Seminar: Software Quality
Advisor: Kohei Dozono
Technical University of Munich
murilo.escher@tum.de

Abstract. The abstract should briefly summarize the contents of the paper in 15–250 words.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

Large language models (LLMs) have recently gained a lot of popularity, be it in academia, business or for personal use. Among their many use cases, one currently much researched possibility is their application in static application security testing (SAST).

Static application security testing is a widely used methodology for identifying software vulnerabilities by analyzing the source code, bytecode or binaries of an application. In contrast to dynamic testing methods, which require the code to be executed, SAST operates statically. This characteristic proves useful for integration into the software development lifecycle, for example in continuous integration (CI) pipelines. However, despite their widespread adoption, SAST tools face notable limitations. They are often not capable of identifying all existing vulnerabilities, especially those caused by runtime behaviour, like dynamic memory allocation. Moreover, they usually generate many false positives, which require a lot of effort for the developers to manually go through [11].

In light of this, the possibility of combining SAST tools' output with LLMs, as a way to enhance their reasoning and mitigate some of these flaws, is being much researched.

Recent studies have investigated the use of LLMs for taint analysis [6], combined SAST output with a retrieval augmented generation (RAG) system [2, 4], and compared the effectiveness of both tools [12]. Despite of highlighting their potential in security testing, significant challenges still remain in the way of practical use [1]

This paper further investigates the usability of LLMs when combined with SAST tools. By leveraging a human-curated dataset containing known vulnerabilities in open-source software (OSS) [5], SAST tools [3, 8], and code context acquired through different program representations, this study evaluates the extent to which LLMs can enhance the accuracy and utility of static analysis results. In this way, it aims to address the challenges presented by false positives in vulnerability detection, contributing to the development of better security systems.

2 Related Work

This section covers related studies and in which ways this project differs from them and provides additional value.

Static application security testing. Lipp et. al [7] did a comprehensive analysis of the effectiveness of different SAST tools. They chose 5 relevant tools and examined their detection rate on different vulnerability classes. Aditionally, they also proposed different voting systems

based on the combination of different static analyzers and examined how they fared in comparison to the use of single tools. The study concluded that the used analyzers were mostly not capable of detecting real-world vulnerabilities, while combining different analyzers proves very useful for increasing the detection rate, despite also marking more functions as vulnerable. Their findings on combining various tools motivated the use of more than one analyzer on this study.

Use of LLMs in vulnerability detection. Ding et. al [1] analyzed the usefulness of code language models (LMs) in real-world vulnerability detection. They highlighted the innefectiveness of the current evaluation metrics and created a new dataset, PrimeVul, to combat the limitations they found in existing benchmark datasets. Their findings reinforce the belief that code LMs are inneffective in detecting vulnerabilities and highlight the need for more code context. Their study did not investigate the difference in performance when this added context is present.

Combining SAST tools with LLMs. Li et. al [6] propose a new approach, IRIS, to combining LLMs with static analyzers. They focused their study on taint analysis in Java vulnerabilities using GPT-4 and CodeQL. This study uses 2 static analyzers and examines C/C++ code.

Providing code context for security testing with LLMs. Risse and Böhme [9] thoroughly analyzed recent top publications in the field of using machine learning for vulnerability detection and argued that their treatment of vulnerability detection as a function-level problem does not represent real-world vulnerabilities. They discussed and highlighted the importance of the calling and/or code context in security analysis. Their results inspired this study to provide the LLM with extra context in the form of callers and callees of the flagged functions.

Keltek et. al [4] combined LLMs with SAST tools and a knowledge retrieval system based on HackerOne vulnerability reports. They proposed different methods for retrieving similar code and improving their RAG system. However, their study did not provide the actual code context from the examined functions and also focused on synthetic datasets, which may not represent real-world software vulnerabilities.

Du et. al [2] created Vul-RAG, a RAG framework for use in vulnerability detection. They focused on Linux kernel Common Vulnerabilities and Exposures (CVE) reports and compared the metrics from different LLMs and Cppcheck, a static analyzer. Their study did not provide the LLM with real code context nor did it use static analysis results as a knowledge source. Moreover, the benchmark dataset was based on pairs of functions, of which one was vulnerable and the other was a similar, but correct version.

Sun et. al [10] created a RAG framework based on similar vulnerability reports and the callees for all analyzed functions. They found that providing code context did not necessarily improve the performance results and was even detrimental in some cases. Their study did not examine the effect of static analysis results as added context on the LLM detection capability.

3 Approach

How the tools were chosen, flow of information, etc

4 Evaluation

How the results are going to be interpreted. For ex. the voting system when using multiple SAST Tools. The research questions we are going to analyse also come here.

5 Conclusion

You can also reference other parts of the document, e.g., sections or subsections. In Section 1 we briefly introduced something, whereas in Subsection **??**, we motivated something else.

Make sure to capitalize chapters, sections or subsections when referencing them.

A Appendix

Anything additional goes here ...

Maybe extra information as to how the code words could be provided, if needed.

References

1. Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., Chen, Y.: Vulnerability Detection with Code Language Models: How Far Are We? . In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). pp. 469–481. IEEE Computer Society, Los Alamitos, CA, USA (May 2025). https://doi.org/10.1109/ICSE55347.2025.00038, `https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00038`
2. Du, X., Zheng, G., Wang, K., Feng, J., Deng, W., Liu, M., Chen, B., Peng, X., Ma, T., Lou, Y.: Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag (2024), `https://arxiv.org/abs/2406.11147`
3. GitHub: Codeql. `https://codeql.github.com/`, accessed: 2025-01-25
4. Keltek, M., Hu, R., Sani, M.F., Li, Z.: Boosting cybersecurity vulnerability scanning based on llm-supported static application security testing (2024), `https://arxiv.org/abs/2409.15735`
5. Kim, J., Hong, S.: Poster: BugOSS: A regression bug benchmark for evaluating fuzzing techniques. In: IEEE International Conference on Software Testing, Verification, and Validation (ICST) (2023)
6. Li, Z., Dutta, S., Naik, M.: Llm-assisted static analysis for detecting security vulnerabilities (2024), `https://arxiv.org/abs/2405.17238`
7. Lipp, S., Banescu, S., Pretschner, A.: An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 544–555. ISSTA 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3533767.3534380, `https://doi.org/10.1145/3533767.3534380`
8. Meta: Infer: A tool to detect bugs in java and c/c++/objective-c code. `https://fbinfer.com/`, accessed: 2025-01-25
9. Risse, N., Böhme, M.: Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection (2024), `https://arxiv.org/abs/2408.12986`
10. Sun, Y., Wu, D., Xue, Y., Liu, H., Ma, W., Zhang, L., Liu, Y., Li, Y.: Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning (2025), `https://arxiv.org/abs/2401.16185`
11. Yang, J., Tan, L., Peyton, J., A Duer, K.: Towards better utilizing static application security testing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 51–60 (2019). https://doi.org/10.1109/ICSE-SEIP.2019.00014
12. Zhou, X., Tran, D.M., Le-Cong, T., Zhang, T., Irsan, I.C., Sumarlin, J., Le, B., Lo, D.: Comparison of static application security testing tools and large language models for repo-level vulnerability detection (2024), `https://arxiv.org/abs/2407.16235`