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.
Large language models have recently seen widespread use in different areas.

Keywords: Security testing · Static analysis · Large language models.

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 life cycle, 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 behavior, like dynamic memory allocation. Moreover, they usually generate many false positives, which require a lot of effort for the developers to manually go through [15].

In light of this, the possibility of combining the output from static analyzers with LLMs, as a way to enhance their reasoning and mitigate some of these flaws, is a promising approach being researched.

Recent studies have investigated the use of LLMs for taint analysis [7], combined SAST output with a retrieval augmented generation (RAG) system [3, 5], and compared the effectiveness of both tools [16]. Despite of highlighting their potential in security testing, significant challenges still remain in the way of practical use [2]. One such challenge is the common necessity of providing context for the vulnerable functions, be it in the form of RAG systems or the actual relations between the targeted functions and the other ones from the project they are inserted in, as this often proves to be decisive when analyzing software security [12].

This paper further investigates the usability of LLMs when combined with SAST tools and extra context. By leveraging a human-curated dataset containing known vulnerabilities in open-source software (OSS) [6], SAST tools [4, 9], and code context, specifically callers and callees of the analyzed functions, 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 reviews relevant studies and highlights how this project extends or differs from them, focusing on four key areas: static application security testing, the use of LLMs for vulnerability detection, the integration of SAST tools with LLMs, and the importance of providing code context for security analysis.

Static application security testing. Lipp et al. [8] conducted a comprehensive evaluation of the effectiveness of different SAST tools, testing their detection capabilities across various vulnerability classes. The study involved five static analyzers and proposed different voting systems based on the combination of different tools and examined how they fared in comparison to the use of single analyzers. It was 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. [2] analyzed the usefulness of code language models (LMs) in real-world vulnerability detection. They highlighted the ineffectiveness 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 ineffective in detecting vulnerabilities and highlight the need for more code context. However, their study did not investigate how providing additional code might affect detection performance, a gap this study aims to address.

Combining SAST tools with LLMs. Li et al. [7] introduced IRIS, a novel framework based on combining LLMs with static analyzers. The study demonstrated the potential of combining GPT-4 with CodeQL to enhance vulnerability detection. However, their focus was on taint analysis in Java vulnerabilities, whereas this study explores the combination of 2 static analyzers and examines C/C++ code.

Providing code context for security testing with LLMs. Risse and Böhme [12] thoroughly analyzed recent top publications in the field of using machine learning for vulnerability detection and argued that their treatment of vulnerability detection as an isolated function-level problem does not reflect real-world vulnerabilities. They highlighted the significance of incorporating calling and code context for effective security analysis. Their results inspired this study to provide the LLM with extra context in the form of caller and callee relationships for flagged functions.

Keltek et al. [5] 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 relied on synthetic datasets, which may not represent real-world software vulnerabilities.

Du et al. [3] 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 the static analyzer Cppcheck. Their study, however, did not include real code context or static analysis results as a knowledge source. Moreover, the benchmark dataset consisted of pairs of functions, where one was vulnerable and the other was a similar, but correct version, which does not correspond to real-world challenges in vulnerability detection.

Sun et al. [13] created a RAG framework based on similar vulnerability reports and the callees for all analyzed functions. Interestingly, they found that additional code context did not necessarily improve the performance results and was even detrimental in some cases. Their study did not examine the effect of integrating static analysis results as contextual information on the LLM detection capability.

3 Approach

This study investigates the potential of combining static analysis tools with large language models to enhance the detection of software vulnerabilities while addressing the issue of false positives. The methodology involves using two static analyzers, CodeQL and Infer, to flag vulnerabilities, enriching their output with additional code context obtained through a code graph of the project, and leveraging an LLM to refine the results. Figure 1 provides an overview of the workflow.

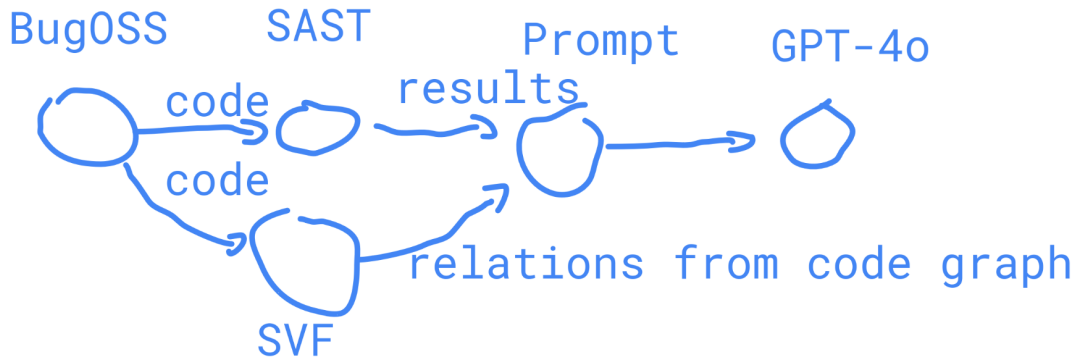

Fig. 1. This diagram denotes the flow of information throughout the project.

In each of the following subsections, detailed explanations of the tools used will be provided, with the goal of clarifying the diagram illustrated above step by step.

3.1 Dataset and Ground Truths

BugOSS [6] is a dataset containing vulnerabilities manually detected in various open-source software projects. It comprises a total of 21 different real-world projects, ranging from lesser known projects like Poppler [10], a library used for rendering PDFs, to widely used software tools, such as Curl [1], a command-line tool that enables data transfer with URLs.

Despite being originally crafted for examinations through fuzz testing, BugOSS contains, for each of the listed projects, information regarding the bug-inducing commit (BIC), as well as the bug-fixing commit (BFC). Together with corresponding Dockerfiles and build scripts for each one of the projects, this dataset proves very useful for the reproducibility of the detected vulnerabilities.

3.2 Static Analysis with CodeQL and Infer

For the static analysis, two widely used tools were chosen: CodeQL and Infer.

3.3 Contextual Analysis with SVF

To attain contextual information about the flagged vulnerable functions, the chosen method was to find out which other functions were callers and callees of the ones analyzed. For this to be possible, it was necessary to generate and analyze the call graph of the code. A call graph is a different representation of a program, in which the program is depicted as a control-flow

graph. In this form, each function becomes a node, and the relationships between functions, as in how they call one another, are shown as edges between the nodes. Thus, this form facilitates the finding out of contextual information for the vulnerable functions.

To generate this call graph, SVF (Static Value-Flow Analysis Framework for Source Code) [14] was used. SVF is a code analysis tool enables interprocedural dependence analysis for LLVM-based languages. By providing SVF with a bytecode file generated with the LLVM compiler, this framework generates a .dot file describing the call graph.

LLVM is an open-source compiler ...

An example of a .dot file is formatted and its translation into a graph illustration can be seen in Figure 2

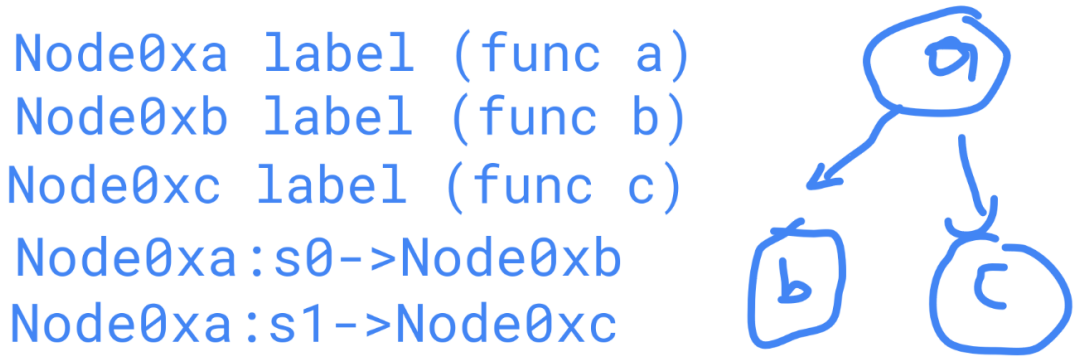

Fig. 2. This images show the side-by-side comparison of a node and its edge into dot-formatting with their graphical representation.

After generating the call graph, it was necessary to filter out the vulnerable functions obtained as explained in Subsection 3.2. This required at first a Python script to match the names of the functions with the corresponding labels of the nodes, as the nodes are not named directly after the functions, as depicted in Figure 2. Some manual filtering was also required due to name mismatches.

The navigation of the graph for finding out callers and callees was done with the pydot [11] library, which easily handled the lookup of the relevant edges and connected functions.

3.4 LLM Augmentation

After attaining SAST results and the contextual information explained in the previous subsections, this data was then put together and formatted into a prompt.

The prompt was formatted as show in Figure 2:

3.5 Summary

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. curl Developers: curl - a command-line tool and library for transferring data with urls. `https://curl.se/`, accessed: 2025-01-25
2. 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`
3. 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`
4. GitHub: Codeql. `https://codeql.github.com/`, accessed: 2025-01-25
5. 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`
6. 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)
7. Li, Z., Dutta, S., Naik, M.: Llm-assisted static analysis for detecting security vulnerabilities (2024), `https://arxiv.org/abs/2405.17238`
8. 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`
9. Meta: Infer: A tool to detect bugs in java and c/c++/objective-c code. `https://fbinfer.com/`, accessed: 2025-01-25
10. Poppler Developers: Poppler - a pdf rendering library. `https://poppler.freedesktop.org/`, accessed: 2025-01-25
11. Pydot Developers: Pydot. `https://pypi.org/project/pydot/`, accessed: 2025-01-25
12. 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`
13. 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`
14. SVF Developers: Svf - source code analysis with static value-flow. `http://svf-tools.github.io/SVF/`, accessed: 2025-01-25
15. 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
16. 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`