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.

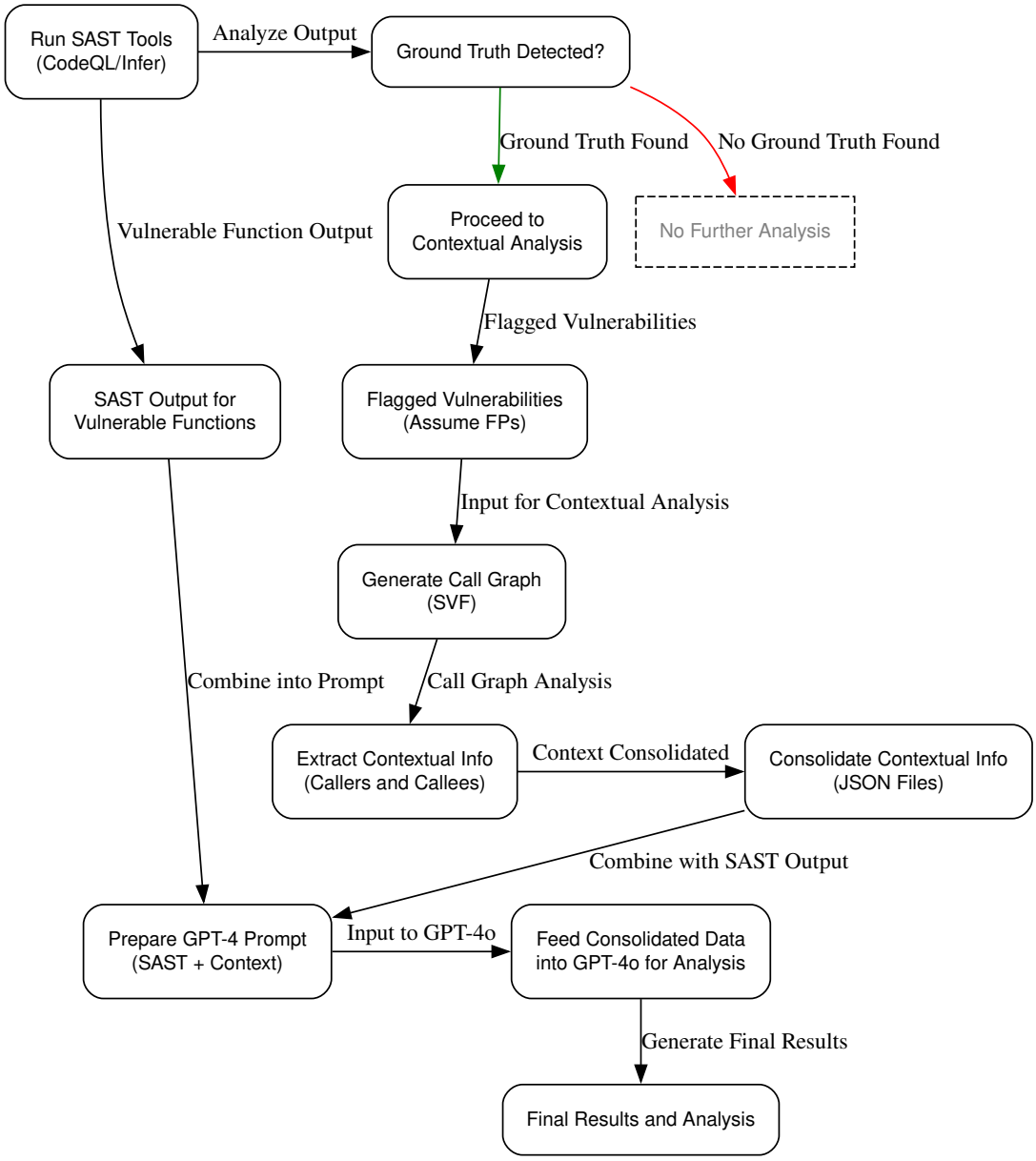


Fig. 1. Overview of the workflow combining SAST tools, contextual analysis, and LLM augmentation.

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. Listing 1.1 provides a clear overview of how information is structured in the dataset.

Listing 1.1. Hierarchy of the BugOSS dataset

```
BugOSS Dataset
+-- Projects (21 Total)
|   \-- Project A (e.g., Poppler)
|   |   +-- Bug-Inducing Commit (BIC)
|   |   +-- Bug-Fixing Commit (BFC)
|   |   +-- Information about Failure Type
|   |   +-- Dockerfile
|   |   +-- Build Script
|   |   \-- Supporting Artifacts
|   \-- Project B (e.g., Curl)
|   |   +-- Bug-Inducing Commit (BIC)
|   |   +-- Bug-Fixing Commit (BFC)
|   |   +-- Information about Failure Type
|   |   +-- Dockerfile
|   |   +-- Build Script
|   |   \-- Supporting Artifacts
\-- Fuzz Testing Metadata
```

For this study, the selection and treatment of ground truths followed a systematic process:

1. **Selection Criteria:** Out of the 21 projects in the dataset, only those that could be successfully built and reproduced using the provided Dockerfiles and build scripts were included in the analysis. While the dataset aims to ensure reproducibility, certain projects encountered build errors due to syntax errors in the code or the BIC not being in a branch. Ultimately, 16 projects were successfully built and used as the basis for further analysis.
2. **Use of Ground Truths:** The validated vulnerabilities served as the baseline for evaluating the performance of static analysis tools. For each successfully built project, the outputs of the tools were analyzed to determine whether they could detect the known vulnerabilities. This process is discussed in greater detail in the next section.

3.2 Static Analysis with CodeQL and Infer

To evaluate the potential of static analysis tools in detecting vulnerabilities, this study utilized two widely known tools: CodeQL and Infer. These tools were selected for their distinct methodologies—CodeQL's query-based analysis and Infer's focus on memory-related issues—and their relevance in the field of software security. Together, they represent complementary approaches to vulnerability detection in real-world software projects.

CodeQL Developed by GitHub, CodeQL is a query-based static analysis tool that generates a database representation of a codebase, enabling developers to query their code as though it were a database. CodeQL excels at identifying vulnerabilities such as:

- SQL injection,
- Cross-site scripting (XSS), and
- Insecure deserialization.

For this study, the standard library of 60 predefined queries provided by CodeQL's base installation was used without customization. While highly effective for analyzing web applications and languages like JavaScript and Python, CodeQL's performance in detecting vulnerabilities in C/C++ codebases was limited. This analysis found that CodeQL struggled to detect vulnerabilities relevant to C/C++ projects, such as memory-related issues, and often produced very few or no results for the BugOSS projects.

To run CodeQL, the provided build scripts from the BugOSS dataset were adapted to work with CodeQL's database creation process. Instead of running the `make` command (or its corresponding equivalent depending on the build setup) directly, the build command was passed to CodeQL's CLI, which intercepted the build process and generated the required database for analysis.

Infer Developed by Meta, Infer is a static analysis tool designed to identify specific classes of vulnerabilities commonly found in C and C++ codebases. Its strengths include detecting:

- Null pointer dereferences,
- Memory leaks, and
- Thread safety violations.

Infer uses symbolic execution to analyze paths through the program's control flow, identifying potential bugs. Due to its specialization in memory-related bugs, Infer proved highly effective for analyzing the C/C++ projects in the BugOSS dataset. However, limitations in build system support were observed: the Harfbuzz project could not be analyzed because its Ninja build system is not supported by Infer. This highlights a key challenge when dealing with non-standard or less commonly supported build configurations.

Similar to CodeQL, Infer required modifications to the BugOSS build scripts. Instead of executing the build command directly, the command was passed through Infer's CLI, allowing it to hook into the build process and analyze the code during compilation.

Comparison and Application in This Study Both tools were run on the successfully built projects from the BugOSS dataset. The application process differed:

- **CodeQL**: Required the creation of a database for each project using its build process.
- **Infer**: Analyzed the source code directly without requiring additional preprocessing.

The effectiveness of each tool was heavily influenced by the types of vulnerabilities they were designed to detect. Given that the BugOSS dataset consists largely of C/C++ projects:

- **Infer's specialization in memory-related issues** resulted in a significantly higher number of detected vulnerabilities across all projects.
- **CodeQL's default query set** was not well-suited for C/C++ projects and detected very few or no bugs in many cases.

This stark difference highlights the importance of tool selection and configuration in vulnerability analysis, especially when analyzing language-specific codebases. The results underline

the need to consider the characteristics of the codebase and vulnerabilities when choosing and setting up static analysis tools.

Given that only one ground truth vulnerability was detected across all analyzed projects - specifically, in the PcapPlusPlus project using Infer - this study followed through with PcapPlusPlus for deeper contextual analysis. In addition to the ground truth vulnerability, flagged vulnerabilities detected by Infer in PcapPlusPlus, which were assumed to include false positives, were also analyzed further. The project thus provided a suitable basis for evaluating both the limitations of static analysis tools and the potential to gather contextual information for reducing false positives among flagged vulnerabilities.

3.3 Contextual Analysis with SVF

To gather contextual information about the flagged vulnerable functions, the chosen method was to identify which other functions were their callers and callees. This required generating and analyzing the call graph of the code. A call graph is a representation of a program in the form of a control-flow graph. In this structure:

- Each function is depicted as a node.
- The relationships between functions, such as function calls, are represented as edges connecting the nodes.

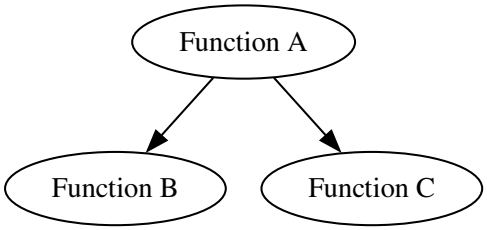
This representation reveals how vulnerable functions interact with other parts of the code, facilitating the extraction of contextual information.

For this analysis, the focus was on the PcapPlusPlus project, as it was the only project for which a ground truth vulnerability was detected by a static analysis tool (Infer). To generate the call graph, the **Static Value-Flow Analysis Framework for Source Code (SVF)** [14] was used. SVF is a code analysis tool that enables interprocedural dependence analysis for LLVM-based languages. By providing SVF with a bytecode file generated using the **LLVM compiler**, the framework produces a .dot file that describes the call graph.

LLVM is an open-source compiler framework widely used for program analysis. It compiles source code into an intermediate representation called LLVM bytecode, which can then be analyzed by tools like SVF. The `.dot` file generated by SVF serves as a textual representation of the call graph. Figure 2 illustrates how the `.dot` formatting translates into a visual representation of a call graph, with nodes representing functions and edges representing calls between them.

```
digraph G {
    a [label="Function A"];
    b [label="Function B"];
    c [label="Function C"];
    a -> b;
    a -> c;
}
```

Call graph in .dot file format.



Graphical representation of the call graph.

Fig. 2. Comparison of the textual and graphical representation of the call graph.

After generating the call graph, the next step was to filter out the vulnerable functions obtained as described in Subsection 3.2. This involved:

1. Using a Python script to match the function names to the corresponding node labels in the graph. This step was necessary because the nodes in the `.dot` file are not directly named after the functions, as shown in Figure 2.

2. Performing some manual filtering to address mismatches and inconsistencies in the node labeling.

To navigate the graph and extract callers and callees of the vulnerable functions, the **pydot** library [11] was used. This Python library efficiently handled the parsing of the `.dot` file and allowed for the lookup of relevant edges and connected functions, making it easier to analyze the relationships between the nodes.

Once the relevant functions and their contexts were extracted, the results were stored in `.json` files for further analysis. These files contained details about the vulnerable functions and their relationships with other functions in the graph (e.g., callers and callees).

Despite this focus, only 11 out of the 51 vulnerabilities detected in PcapPlusPlus appeared in the call graph generated by SVF. This discrepancy highlights a limitation in the representation or the analysis process, as certain vulnerabilities may correspond to code that was not captured or connected in the generated call graph.

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.

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