Explaining Software Vulnerabilities with Large Language Models

Oshando Johnson
Fraunhofer IEM
Paderborn, Germany
oshando.johnson@iem.fraunhofer.de

Alexandra Fomina
Chapman University
California, United States
fomina@chapman.edu

Ranjith Krishnamurthy
Fraunhofer IEM
Paderborn, Germany
ranjith.krishnamurthy@iem.fraunhofer.de

Vaibhav Chaudhari
Paderborn University
Paderborn, Germany
vaibhav.chaudhari@uni-paderborn.de

Rohith Kumar Shanmuganathan
University of Oldenburg
Oldenburg, Germany
rohith.shanmuganathan@uol.de

Eric Bodden
Paderborn University and Fraunhofer IEM
Paderborn, Germany
eric.bodden@uni-paderborn.de

***Abstract*—The prevalence of security vulnerabilities has prompted companies to adopt static application security testing (SAST) tools for vulnerability detection. Nevertheless, these tools frequently exhibit usability limitations, as their generic warning messages do not sufficiently communicate important information to developers, resulting in misunderstandings or oversight of critical findings. In light of recent developments in Large Language Models (LLMs) and their text generation capabilities, our work investigates a hybrid approach that uses LLMs to tackle the SAST explainability challenges. In this paper, we present *SAFE*, an Integrated Development Environment (IDE) plugin that leverages GPT-4o to explain the causes, impacts, and mitigation strategies of vulnerabilities detected by SAST tools. Our expert user study findings indicate that the explanations generated by *SAFE* can significantly assist beginner to intermediate developers in understanding and addressing security vulnerabilities, thereby improving the overall usability of SAST tools.**

***Index Terms*—vulnerability explanation, static analysis, large language models, explainability, vulnerability detection**

I. Introduction

With the rise in software security vulnerabilities such as those in the Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Weaknesses list [1], many companies resort to static application security testing (SAST) tools for the detection of software vulnerabilities. Given that many SAST tools provide generic warning messages and also lack sufficient information about the detected vulnerabilities, developers often misunderstand or ignore the tool findings [2]. As an alternative, developers have expressed the need for improved warning messages similar to the explanations found on blogs and online forums [2].

To address the explainability challenges of SAST tools, recent developments in Large Language Models (LLMs) have provided new possibilities for security-oriented tasks such as vulnerability detection, given the capability of LLMs to understand code and convey information in natural language [3]. However, these approaches often present LLMs as possible alternatives for established static application security testing approaches. Despite advancements in vulnerability detection using LLMs, previous studies have unequivocally demonstrated that for critical systems, SAST is advisable due to its ability to deliver the requisite reliability and precision [4]. This underscores the necessity of investigating complementary roles of LLMs in vulnerability detection rather than viewing them only as substitutes. However, there is little research exploring hybrid approaches that integrate SAST tools with LLMs to leverage the strengths of both methodologies. To address this research gap, we explore the effectiveness of using LLMs to explain the cause, impact, and mitigation for vulnerabilities detected by SAST tools [2].

In this paper, we present *SAFE* (Static Analysis Findings Explainer), an IntelliJ IDEA plugin that leverages LLMs to explain vulnerabilities for developers with limited software security experience. To identify the most suitable LLM for *SAFE*, we benchmarked widely used open and closed-source models on vulnerability detection tasks using various prompt engineering techniques. Using one of the top performing model, GPT-4o, we evaluated *SAFE* in an expert study, which revealed that the explanations are helpful for developers with beginner to intermediate experience in software security.

We present *SAFE* in Section II, evaluate it in Section III, review related work in Section IV, and conclude in Section VI. The plugin source code, demo and evaluation results are available online [5].

II. Approach

In this section, we describe the *SAFE* plugin, which explains software vulnerabilities detected by SAST Tools. Figure 2 shows an overview of the plugin's architecture. The *parser* processes SAST result files and extends the vulnerability findings with code snippets sourced from the *Integrated Development Environment (IDE)*. Additionally, the results are enriched with supplementary information from *security catalogs* [6] through the *annotator*. The prompt engine utilizes the annotated results to construct a prompt, which is sent to the LLM to produce explanations. The plugin's user interface

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00045

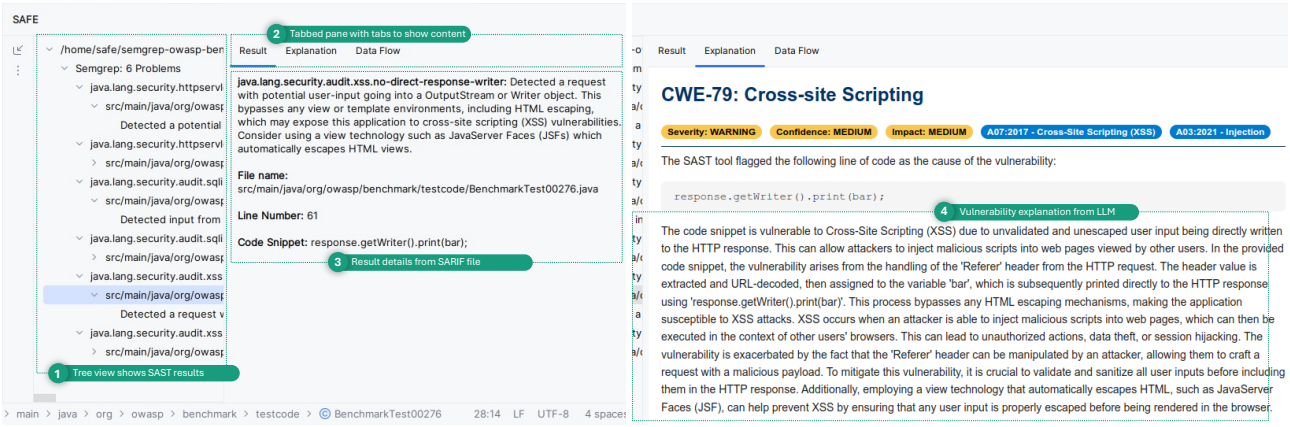

Fig. 1. *SAFE*'s tool window screenshot showing the tree view (1) and tabbed pane (2) containing tabs for result details, explanations, and data-flow. The result details (3) and explanation (4) for a sample cross-site scripting vulnerability are shown.

(UI) displays both the identified vulnerabilities and their corresponding explanations. In the following sections, we describe the modules for parsing and annotating the SAST results, generating explanations, and the user interface.

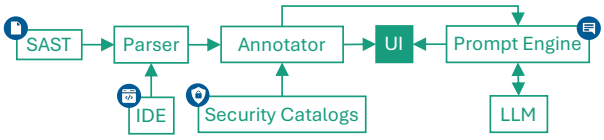

Fig. 2. Architecture of the *SAFE* Integrated Development Environment plugin for explaining static analysis tool results with large language models.

A. Parsing and Annotating SAST Vulnerability Results

The parser processes the output of static analysis tools provided in the industry standard *Static Analysis Results Interchange Format* (SARIF) [7]. From the SARIF file, the parser extracts a property named *results*, an array containing single results detected by the SAST tool [7]. The property contains objects for the result's type, severity, message, and location. Using the *location* property, the source code of the method containing the vulnerability is obtained using application programming interfaces (APIs) provided by the IntelliJ IDEA platform. When available, the parser extracts the *threadFlows* property, which is an array storing the code location for a specific single thread execution path through the program [7]. From this property, the parser extracts data-flow information relevant for taint analysis, such as the source, intermediate elements, and sink for the vulnerability [6]. The results are annotated with information from two security catalogs: the CWE list of software weaknesses [1] and a list of critical methods [6] in programs that influence security.

B. Explaining SAST Results with Large Language Models

To explain the results, SAFE employs a zero-shot prompting strategy [8], in which the explanation task is presented to the model without any prior examples. Using role-playing [8], we assign the role of a security expert to the LLM in the system prompt, while the user prompt outlines the specific explanation task. Using the data extracted in Sub-Section II-A from the SARIF file, we populate the below zero-shot prompt template.

> ***System***: You are an assistant with expertise in explaining software security vulnerabilities in code snippets. You will be given a code snippet and the result from a static analysis security testing tool for the code snippet. Your task is to explain the static analysis result to a software developer based on their software security experience level. Provide information about the underlying cause, consequences, and mitigation strategies for the reported vulnerability.
> When providing a response, follow these guidelines:
> {*Formatting and output Guidelines*}.

> ***User***: Explain the vulnerability detected in the code snippet to a developer who has {*level*} experience in software security.
> Detected Vulnerability: {*rule name*}:{*rule message*}
> Code Snippet: {*location*}
> Line with vulnerability: {*location-line*}
> Data-flow: {*data-flow*}

The *level* field is replaced with the user's experience level (beginner, intermediate, or advanced) with software security and impacts the level of detail of the explanations. The *rule-name* and *rule-message* fields are replaced with the vulnerability rule name and warning message reported by the SAST tool. The *location* and *location-line* placeholders are replaced with the source code of the vulnerable method and line, respectively. Finally, the *data-flow* placeholder is replaced with the taint source, intermediate, and sink lines of codes.

C. User Interface

The plugin's user interface uses a tool window (child windows in IntelliJ Idea for displaying information) and contains

a toolbar with tool window buttons. Figure 1 displays a screenshot of the plugin's tool window with the result message and explanation for a cross-site scripting vulnerability detected by the static analysis tool Semgrep v1.119.0. A screencast of the plugin is available on GitHub [5]. The tool window is divided into two parts: a custom tree view to hierarchically display the results from the SARIF file (❶) and a tabbed pane to show content (❷). The Results tab (❸) shows the result details (*results* property), the Explanation tab shows the LLM explanation (❹), and the data-flow tab shows data-flow information from the *threadflow* property. The explanation tab additionally contains the vulnerability type, severity, and impact, as well as the SAST tool's confidence, general mitigation strategies, and thumbs up/down feedback buttons.

III. Evaluation

In this section, we evaluate *SAFE*'s main objective of explaining vulnerabilities detected by static analysis tools with LLMs using the following research questions:

- ***RQ1***: Which large language models and prompt engineering techniques achieve the best performance in detecting security vulnerabilities?
- ***RQ2***: To what extent can large language models explain vulnerabilities detected by static analysis tools for users with different software security experience?

For the evaluation, we use the OWASP Benchmark [9], a comprehensive Java vulnerability test suite containing more than 2000 test cases across 11 vulnerability types. In the next subsections we describe the experiments and results.

A. RQ1: Vulnerability Detection with Large Language Models

Before evaluating the quality of explanations provided by LLMs (addressed in RQ2), it is essential to first determine which model and prompting strategy are most suitable for the task of vulnerability detection. To this end, we developed a benchmarking framework, VuLLMBench [10], that systematically compares multiple LLMs and different prompting techniques for vulnerability detection. The rationale is that if a model employing a specific prompting strategy excels in vulnerability detection, it is likely to exhibit internal reasoning capabilities, which may enable it to produce reliable explanations of the identified vulnerabilities.

We evaluated role-based prompts without examples (zero-shot), with code examples (few-shot), and also with a series of intermediate reasoning steps (chain-of-thought). Using these prompting techniques, we tested 22 open- and closed-source LLMs that have been recommended for vulnerability detection tasks. Our experiments confirmed previous research [11], which reported that zero-shot prompts are more effective for vulnerability detection tasks.

To further evaluate the robustness, generalization, and real-world applicability of the models, we experimented with basic name-based obfuscation techniques in which variables, methods, and classes were renamed. Experimenting with obfuscated code provides insights into how well models can identify vulnerabilities when the code is altered, thereby testing their ability to discern underlying logic and security flaws. Table I reports the performance of the top 10 LLMs using a zero-shot prompt on the OWASP Benchmark for the original and obfuscated test cases. The complete results are available in the project repository [10].

TABLE I
Precision (P), recall (R) and F1-Score (F1) for vulnerability detection with LLMs on the OWASP Benchmark.

	Original			*Obfuscated*		
Model	**P**	**R**	**F1**	**P**	**R**	**F1**
GPT-5	0.78	0.98	0.87	0.75	0.96	0.84
o3-mini	0.82	0.90	0.86	0.84	0.88	0.86
GPT-5-mini	0.72	0.96	0.83	0.75	0.95	0.84
GPT-4o	0.59	1.00	0.74	0.53	0.99	0.69
Deepcoder (14B)	0.70	0.75	0.72	0.65	0.68	0.66
Llama3.1 (70B)	0.57	0.96	0.72	0.56	0.96	0.70
GPT-4o-mini	0.54	0.98	0.70	0.55	0.97	0.70
Gemma2 (9B)	0.53	0.99	0.69	0.53	0.88	0.66
CodeGemma (7B)	0.57	0.86	0.69	0.52	0.88	0.65
CodeLlama (70B)	0.53	0.96	0.68	0.52	0.94	0.67

The reasoning models (GPT-5 and o3) outperformed the other models on the OWASP Benchmark with F1-Scores above 0.83, given their increased ability to make more reliable and accurate decisions, work through ambiguity that may exist in the code snippets, and solve problems [12]. The performance of these models could be possibly improved by designing prompts that following the guidelines for reasoning models such as using developer instead of system messages [12]. The general models (GPT-4o, Llama3.1, and GPT-4o-mini) as well as the code models (Deepcoder, CodeGemma and CodeLlama) also achieved relatively good F1-Scores due to high recall, however, the models have relatively low precision. These findings suggest that the models produce a high rate of false positives and are unable to effectively filter noise—a known challenge for developers using static analysis tools [2].

The performance of most of the models was negatively impacted due to the minor lexical changes from the name-based obfuscation. Although simply renaming identifiers does not impact the presence of the vulnerabilities and most SAST tools would be robust against such changes, some of the models seem to misled by such alterations in the code.

Although GPT-5 achieved the strongest overall performance in our benchmarks, its August 2025 release postdated our user study (see RQ2), for which GPT-4o had already been selected; consequently, GPT-5 was not included. We added the reasoning model o3-mini to the benchmark alongside the GPT-5 reasoning models. Although o3-mini outperformed GPT-4o on the metrics in Table I, it required approximately twice the runtime in our benchmarks and was more expensive. Accordingly, we selected GPT-4o, consistent with OpenAI's guidance for coding and agentic tasks; moreover, GPT models remain well suited for well-defined tasks, with lower latency and cost [12].

B. RQ2: LLM-generated Explanations Evaluation

To evaluate the explanations, we analyzed the OWASP Benchmark with the open-source static analysis tool Semgrep (version 1.119.0). From the benchmark results, we selected two random samples of the top 3 most dangerous vulnerabilities [1], namely *CWE22 Path Traversal*, *CWE89 SQL Injection*, and *CWE79 Cross-site Scripting*. Figure 1 shows the message reported by Semgrep as well as SAFE's generated explanation. Semgrep's message for the "no-direct-response-writer" rule is intentionally generic, as it is reused across all detected instances. In contrast, *SAFE*'s explanation provides a step-by-step walkthrough tailored to the specific finding, referencing the variables and lines of code that may contribute to the vulnerability. *SAFE*'s explanations align with research-recommended warning practices: they are detailed and descriptive, outline the analysis steps, and adapt to the coding context in which the vulnerability occurs [13].

To further assess the quality of *SAFE*'s explanations, we conducted an expert study with four experienced software security trainers to evaluate the accuracy, correctness, readability, and helpfulness of the generated messages. Three of the trainers were certified scientific trainers. On average, they had four years of training experience and had delivered eight training sessions, primarily to participants with beginner-to-intermediate software security backgrounds. The trainings focused on secure software engineering and information technology security. We opted for a human evaluation study [14] with experts to verify the explanations against the source code and the SAST tool's results.

The one-hour study comprised a survey to capture participants' training experience, a plugin demonstration, an expert usability test, and an interview. Participants evaluated the LLM-generated explanations using five criteria or attributes:

1) *Relevant*: relates to vulnerability, appropriate for user's experience, fitting vocabulary, and essential details
2) *Faithful*: free from hallucination (information that is not supported by the source text and vulnerability)
3) *Concise*: information-dense, does not repeat the same point multiple times, and is not unnecessarily verbose
4) *Coherent*: well-structured, easy to follow, not just a jumble of facts, grammatically and syntactically sound
5) *Accuracy*: captures the original warning message and code meaning

Relevance and (coherency) cohesiveness are commonly used in human evaluations of generated text [14]. Because effective warning messages should clearly identify the detected issue, explain why it matters, and describe how to fix it [2], we emphasized conciseness. For vulnerability detection tasks, it is critical to convey accurate information; accordingly, we assessed faithfulness and accuracy. These criteria are evaluated using a 5-point Likert scale with options ranging from *very poor* to *very good*.

Figure 3 shows the expert evaluations of the explanations for the vulnerabilities detected in the OWASP Benchmark with Semgrep. For all of the criteria, the trainers considered 64% of the explanations to be at least *acceptable* and none of them were considered to be *very poor*. The *relevant* criteria had the highest distribution (35%) of *poor* evaluations, arising from an overlap in the content of explanations for the different levels and explanations being more relevant for another level. Although the trainers mentioned that beginner and intermediate explanations were usually fitting, more general explanations would be helpful for beginners while intermediate explanations could explain the detected vulnerability. For the *faithful* category, all of the explanations evaluated were considered to be at least good and therefore free from hallucinations. This was also confirmed in an initial assessment by four of the authors: the explanations contained no hallucinations and correctly referenced the given the prompt-provided context. The distribution among the responses for the *concise* criteria arises because the explanations contained information that was not fitting, repeated or unnecessary, too wordy, too technical, and often a mere rephrasing of the SAST tool output. A similar evaluation of the *coherent* criteria is observed given the even split across *acceptable, good and very good*. The trainers cited that the structure of the explanation was not always logical and missed important information such as mitigation strategies. 90% of the explanations were considered to be *accurate* (*good and very good*) given that the code was often correctly explained by the LLM. However, in the initial assessment by the authors, it was observed that even for false positives, the explanations still implied a vulnerability in the source code, indicating that the language model tended to assume all findings were true positives. This issue can be mitigated by instructing the large language model to validate each reported vulnerability or by applying other false-positive detection methods before generating explanations.

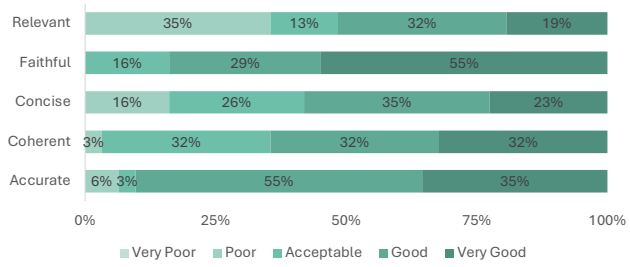

Fig. 3. Stacked bar chart showing the evaluation of the vulnerability explanations using a Likert scale with options from very poor to very good.

Qualitatively, the participants found the generated explanations useful for novice and intermediate users and more effective than the SAST tool messages for explaining the vulnerability causes, impact, and mitigation strategies. They recommended improvements to explanation utility, automatic skill-level detection, and overall plugin usability improvements. Explanation quality (relevance, concision, coherence) could be enhanced by refining the prompt template with clearer instructions and sufficient task context. They also advocated for a clearer distinction between novice and intermediate levels

and level-specific prompts, rather than a single prompt for all users.

IV. Related Work

Prior work on explainability in static analysis has introduced contextual information, visualizations, and standardized reporting formats. For example, rule graphs encode the rules and data flows underlying vulnerability detection, thereby visualizing the internal reasoning of analysis tools [15]. While effective, this approach assumes domain knowledge of taint analysis to interpret the depicted flows. Our approach addresses this barrier by using AI-generated natural-language explanations of data flows, making them accessible to developers with limited software security expertise.

In the domain of artificial intelligence, prior work has examined large language models for vulnerability detection and explanation, often positioning them as replacements for state-of-the-art AI techniques and static application security testing (SAST). LLM-based approaches such as LLMVulExp [3] focus on using various prompt engineering and fine tuning techniques to enhance vulnerability detection, explanation and repair. LLM4SA [16] builds on these approaches but also considers the static analysis warnings in order to filter out false positives. Using LLMs to explain vulnerabilities detected by static analysis tools has not been well-researched and *SAFE* bridges this gap by providing explanations using the calling context in which the vulnerability was detected.

V. Limitations and Threats to validity

We acknowledge several threats to validity that may affect the interpretation and generalizability of our results, which we outline below.

a) External validity: Our evaluation relied on security experts rather than the target user population (beginner/intermediate developers), which may limit generalizability. We plan to refine the plugin based on expert feedback and conduct a large-scale user study with developers at these levels.

b) Construct/conclusion validity: : RQ1 underpins RQ2, but we did not analyze the relationship between vulnerability detection performance and explanation quality in LLMs. This warrants further investigation through experiments that compare multiple models, analyze their outputs to identify similarities and differences, and quantify the correlation between detection accuracy and explanatory quality. We plan to undertake a more comprehensive investigation of the relationship between LLMs' vulnerability detection performance and explanatory capabilities.

VI. Conclusion

In this paper, we address some of the usability/explainability limitations of static analysis tools by developing an approach *SAFE* that uses LLMs to explain security vulnerabilities. We bench-marked various models and prompt engineering techniques and evaluated the explanations generated with GPT-4o in a study with software security trainers. Our results showed that a hybrid approach combining LLMs with SAST can help to improve the limitations of both approaches for vulnerability detection tasks, especially for developers with beginner to intermediate software security experience.

Acknowledgment

We acknowledge support by the German state of North Rhine-Westphalia for the CyberResilience.nrw Project funded through the NEXT.IN.NRW innovation competition.

References

[1] MITRE Corporation, "2024 cwe top 25 most dangerous software weaknesses," 2025. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

[2] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. South Korea: ACM, Jul. 2022, p. 532–543.

[3] Q. Mao, Z. Li, X. Hu, K. Liu, X. Xia, and J. Sun, "Towards effectively detecting and explaining vulnerabilities using large language models," *arXiv preprint arXiv:2406.09701*, 2024.

[4] D. Gnieciak and T. Szandala, "Large language models versus static code analysis tools: A systematic benchmark for vulnerability detection," *arXiv preprint arXiv:2508.04448*, 2025.

[5] Fraunhofer IEM, "Safe - static analysis findings explainer," 8 2025. [Online]. Available: https://github.com/fraunhofer-iem/safe

[6] O. Johnson, G. Piskachev, R. Krishnamurthy, and E. Bodden, "Detecting security-relevant methods using multi-label machine learning," in *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, ser. IDE '24. New York: ACM, 2024, p. 101–106.

[7] E. by Michael C. Fanning and L. J. Golding. (2023, Aug.) Static analysis results interchange format (sarif) version 2.1.0 plus errata 01. [Online]. Available: https://docs.oasis-open.org/sarif/sarif/v2.1.0/errata01/os/sarif-v2.1.0-errata01-os-complete.html.Lateststage:https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[8] A. Kong, S. Zhao, H. Chen, Q. Li, Y. Qin, R. Sun, X. Zhou, E. Wang, and X. Dong, "Better zero-shot reasoning with role-play prompting," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2024, pp. 4099–4113.

[9] OWASP, "Owasp benchmark," https://owasp.org/www-project-benchmark/, 2020.

[10] V. Chaudhari, "Vullmbench," 8 2025. [Online]. Available: https://github.com/vaibhavc99/VuLLMBench

[11] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, "Unleashing the potential of prompt engineering for large language models," *Patterns*, vol. 6, no. 6, p. 101260, 2025.

[12] OpenAI, "Reasoning best practices," https://platform.openai.com/docs/guides/reasoning-best-practices, 2025.

[13] M. Nachtigall, L. Nguyen Quang Do, and E. Bodden, "Explaining static analysis - a perspective," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2019, pp. 29–32.

[14] C.-H. Chiang and H.-y. Lee, "Can large language models be an alternative to human evaluations?" in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 15 607–15 631. [Online]. Available: https://aclanthology.org/2023.acl-long.870/

[15] L. N. Q. Do and E. Bodden, "Explaining static analysis with rule graphs," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 678–690, 2022.

[16] Wen, Cheng et al., "Automatically inspecting thousands of static bug warnings with large language model: How far are we?" *ACM Trans. Knowl. Discov. Data*, vol. 18, no. 7, jun 2024.