

A Study on Vulnerability Explanation Using Large Language Models

Lucas B. Germano^a and Julio Cesar Duarte^b

Military Institute of Engineering, Brazil
{lucas.germano, duarte}@ime.eb.br

Keywords: Large Language Models, LLMs, Vulnerability Explanation, Software Security, CodeLlama, ChatGPT, GPT-4.

Abstract: In the quickly advancing field of software development, addressing vulnerabilities with robust security measures is essential. While much research has focused on vulnerability detection using Large Language Models (LLMs), limited attention has been given to generating actionable explanations. This study explores the capability of LLMs to explain vulnerabilities in Java code, structuring outputs into four dimensions: why the vulnerability exists, its dangers, how it can be exploited, and mitigation recommendations. In this context, smaller LLMs struggled to produce outputs in the required JSON format, with CodeGeeX4 showing high semantic similarity to GPT-4o but generating many incorrect formats. CodeLlama 34B emerged as the best overall performer, balancing output quality and formatting consistency. Despite these findings, comparisons with the GPT-4o baseline revealed no significant differences to rank the models effectively. Human evaluation further revealed that all models, including GPT-4o, struggled to adequately explain complex vulnerabilities, underscoring the challenges in achieving comprehensive explanations.

1 INTRODUCTION

In the continuous software development landscape, security vulnerabilities persist as a challenge, threatening systems' integrity, confidentiality, and availability. Identifying these vulnerabilities has traditionally relied on static and dynamic analysis tools, which, while effective, often lack the ability to contextualize and explain vulnerabilities in a way that is accessible to diverse audiences, including developers, security professionals, and decision-makers. The emergence of Large Language Models (LLMs) offers a transformative approach to this problem, with their capacity to process and generate natural language based on complex patterns in data.


LLMs like GPT and CodeLlama have shown promise in tasks such as vulnerability detection, repair, and explanation. However, most research emphasizes detection, with limited focus on explaining vulnerabilities' root causes, impacts, and remediation. Such explanations are crucial for enhancing developer understanding and promoting proactive security practices.


The study aims to investigate and demonstrate the capability of LLMs to generate contextualized explanations of vulnerabilities in Java code. It focuses

on producing explanations, encompassing critical attributes, enabling a comprehensive understanding and effective remediation of these vulnerabilities. These attributes include: (1) elucidating why the vulnerability exists by identifying its root causes; (2) explaining how the vulnerability can be exploited, outlining potential attack vectors and exploitation scenarios; (3) assessing the danger posed by the vulnerability if successfully exploited, including potential impacts on security, functionality, and data integrity; and (4) providing actionable guidance on how developers can effectively mitigate or fix the identified vulnerabilities. By addressing these objectives, the study seeks to enhance the utility of LLMs in improving software security practices and developer awareness.

The results of this study show that CodeLlama 34B emerged as the best performer, balancing quality and formatting consistency, while smaller models like CodeGeeX4 often struggled with JSON formatting. However, all models, including GPT-4o, faced challenges in providing comprehensive explanations for complex vulnerabilities, highlighting both the potential of LLMs in generating structured explanations and their limitations in addressing intricate scenarios.

The paper is structured as follows: Section 2 reviews related work on LLM-based vulnerability detection and explanation. Section 3 details the methodology for model training and evaluation. Section 4

^a  <https://orcid.org/0009-0007-1607-4863>

^b  <https://orcid.org/0000-0001-6656-1247>

presents the results, and Section 5 discusses insights and future research directions.

2 RELATED WORK

An extensive literature review revealed no studies directly addressing the task of explaining vulnerabilities. The most closely related work is GPTLens (Hu et al., 2023), which investigates the application of LLMs for vulnerability detection in smart contracts.

GPTLENS introduces an “Auditor” and “Critic” framework to enhance detection accuracy, balancing diverse predictions and reduced false positives. The “Auditor” identifies vulnerabilities and generates reasoning, while the “Critic” evaluates correctness, severity, and profitability. While the framework includes an explanation mechanism, it is limited, providing only brief reasoning on why vulnerabilities exist without detailing dangers, exploitation methods, or mitigation steps. Additionally, the reliance on GPT-4 for both generation and evaluation raises concerns about bias, highlighting the need for human validation to ensure trustworthiness.

While only one study directly addressing vulnerability explanation was identified, numerous works focus on vulnerability detection, particularly employing LLMs. An example is LineVul (Fu and Tantithamthavorn, 2022), a Transformer-based approach for fine-grained vulnerability prediction in C/C++ code. LineVul utilizes BERT’s self-attention mechanism to achieve line-level predictions, significantly improving the accuracy of locating vulnerable code compared to coarse-grained methods.

Similarly, other studies such as (Chen et al., 2023; Hin et al., 2022; Nguyen et al., 2022) also focus on the task of detecting vulnerabilities using advanced techniques and models.

In addition, works like (Fu et al., 2022; Wu et al., 2023; Zhang et al., 2024) focus on vulnerability repair, introducing methods for automated fixes. However, these studies also lack emphasis on explaining the repaired vulnerabilities, leaving developers with a limited understanding of the changes made or their implications.

To contextualize this study within the scope of existing research, Table 1 provides a comparative analysis of these studies, highlighting their primary contributions to vulnerability detection, repair, and explanation tasks. It demonstrates the current emphasis on detection and repair while revealing a lack of focus on explanation across most works.

This gap underscores the need for approaches that extend beyond identifying or fixing vulnerabilities to

delivering in-depth insights into their root causes, associated risks, exploitation strategies, and effective mitigation techniques. This study addresses that gap by proposing a framework dedicated to delivering comprehensive and actionable explanations for software vulnerabilities.

3 EXPERIMENTAL WORKFLOW

This study’s experimental workflow systematically evaluates the effectiveness of LLMs in explaining vulnerabilities in Java source code. The process starts with creating a refined dataset tailored for the experiment. Prompt engineering techniques then ensure consistent and structured input formats for generating clear and concise explanations. A Retrieval-Augmented Generation (RAG) approach enriches the input dynamically with accurate and up-to-date vulnerability context. Lastly, LLM selection is based on a detailed performance benchmark analysis, prioritizing state-of-the-art models that meet computational constraints and research objectives. These interconnected steps, outlined in Figure 1, provide a structured framework for exploring LLM capabilities in software vulnerability analysis. Each step is detailed in the subsections below.

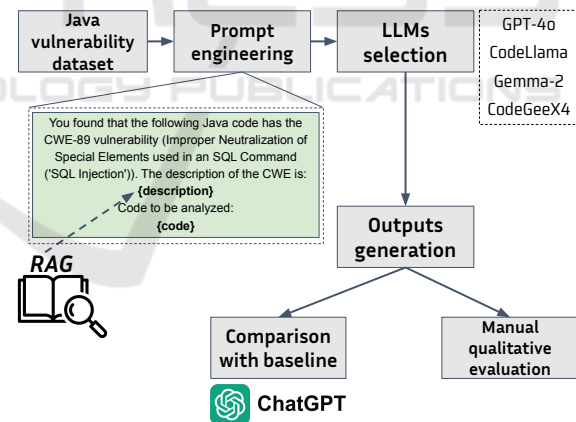


Figure 1: Methodology employed on the experiment.

3.1 Java Vulnerability Dataset

The proposed dataset for this study is the ReposVul (Wang et al., 2024) dataset, which includes around 1,000 Java test cases. A filtering process was applied to refine the dataset based on the following criteria:

- The test case must explicitly include the CWE (Common Weakness Enumeration) number of the vulnerability.

Table 1: Primary focus of cited studies on vulnerability detection, repair, and explanation. Full circle (●) = main focus, half circle (◐) = partial focus, empty circle (○) = not addressed.

Study	Detection	Repair	Explanation
GPTLENS (Hu et al., 2023)	●	○	◐
LineVul (Fu and Tantithamthavorn, 2022)	●	○	○
DiverseVul (Chen et al., 2023)	●	○	○
LineVD (Hin et al., 2022)	●	○	○
ReGVD (Nguyen et al., 2022)	●	○	○
VulRepair (Fu et al., 2022)	○	●	○
Wu et al. (Wu et al., 2023)	○	●	○
Zhang et al. (Zhang et al., 2024)	○	●	○
This Work	○	○	●

- The patch for the vulnerability should change at most one file.

These criteria were selected to address two key challenges. First, since the experiments involve evaluating seven different LLM models, detailed in subsection 3.4, running all 1,000 test cases would be prohibitively time-consuming. Filtering reduces the dataset to a more manageable size without compromising representativeness. Second, multi-file vulnerabilities are naturally more complex to analyze and are prone to false positives, as developers may modify unrelated code during patching. Limiting the dataset to cases where the patch affects at most one file helps minimize this risk, ensuring that the selected test cases are more straightforward for the models to analyze and explain.

After applying this filtering process, the dataset was reduced to 170 cases, resulting in a balanced and representative sample for the study.

3.2 Prompt Engineering

This study employed prompt engineering to ensure coherence and structure in the LLM outputs. The output format was strictly defined as JSON, with four keys: why, danger, how, and fix. Each key corresponded to a specific aspect of the vulnerability, as detailed in Box 1. This approach ensured clear and consistent representation of explanations.

Prompts explicitly instructed LLMs to avoid generating code fixes, focusing instead on contextually accurate explanations. They directed models to reference specific variables and functions from the provided code, ensuring explanations remained grounded in context.

Additionally, prompts required the LLMs to focus exclusively on the specified CWE vulnerability. For instance, when analyzing CWE-89 (SQL Injection), the LLMs were instructed to address only this issue, ignoring unrelated vulnerabilities.

The system prompt sets the rules, behavior, and response format, ensuring the model adheres to constraints like focusing on the specified CWE and using JSON formatting. The user prompt provides task-specific input, such as the Java code and vulnerability description. An example user prompt for CWE-89 (SQL Injection) is shown in Box 2.

Box 1 | System Prompt

You are a software security specialist and will be asked to provide a JSON response about vulnerabilities found in Java source code. Do not write any code in your response, but you may cite variables and functions. The JSON must contain four specific keys: why, danger, how, and fix. Each key should correspond to a short and concise paragraph as instructed:

- **why:** Explain why the vulnerability happens.
- **danger:** Describe the danger the vulnerability may cause if exploited.
- **how:** Explain how the vulnerability could be exploited.
- **fix:** Provide directions to fix the vulnerability, but do not write any code.

Do not include any other keys or write responses outside the JSON format. If any part of the response cannot be completed, explicitly state "No information available" for that key. Concentrate solely on vulnerabilities related to the given CWE, ignoring all other types of vulnerabilities.

A key aspect of prompt creation is focusing on the affected code segment identified in the patch. The prompt includes the old, vulnerable code while excluding the corrected code. Although full file context would be ideal, many files exceed the hardware memory limits, so the prompts prioritize the impacted segment to balance context and resource constraints.

Box 2 | User Prompt

You found that the following Java code has the CWE-89 vulnerability (Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')). The description of the CWE is: The product constructs all or part of an SQL command using externally influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data. Code to be analyzed: {code}

3.3 RAG

A Retrieval-Augmented Generation (RAG) approach is used to provide the LLMs with accurate and relevant information about vulnerabilities. This method dynamically enriches the input with real-time context fetched via web scraping. Specifically, the mechanism retrieves the full name and description of the targeted vulnerability from the CWE website¹ on demand, ensuring the prompts include the most up-to-date details. This integration enhances the relevance and accuracy of the explanations while maintaining efficiency. Figure 2 illustrates this process.

3.4 LLMs Selection

Selecting appropriate LLMs is crucial due to the rapid evolution of this field. Peer-reviewed benchmarks often become outdated by the time they are published, making online benchmarks a practical alternative for evaluating the latest models. This study consulted several benchmarks, including Can AI Code Results², BigCode Models Leaderboard³, Aider Chat Leaderboards⁴, ProLLM Coding Assistant Leaderboard⁵, and BigCode Bench⁶.

From these benchmarks, six models were selected for the experiments in this study: CodeLlama (7B, 13B, 34B, and 70B versions) (Rozière et al., 2024), Gemma 2 27B (Team et al., 2024), and CodeGeeX4

¹<https://cwe.mitre.org/>

²<https://huggingface.co/spaces/mike-ravkine/can-ai-code-results>

³<https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard>

⁴<https://aider.chat/docs/leaderboards/>

⁵<https://prollm.toqan.ai/leaderboard/coding-assistant>

⁶<https://bigcode-bench.github.io/>

9B (Zheng et al., 2023), where B stands for billion parameters. The primary selection criteria were:

- **Hardware Compatibility.** Each selected model must fit within the memory constraints of a 64GB VRAM GPU, as detailed in subsection 4.1. This ensures that the models can be evaluated without additional infrastructure constraints.
- **Performance Ranking.** The selected models represent the best-performing LLMs across the consulted benchmarks, prioritizing their relevance and effectiveness in code-related tasks.

This approach allows the study to utilize state-of-the-art LLMs that are both feasible to deploy and aligned with the objectives of vulnerability explanation in Java. The selected models are evaluated extensively in the subsequent sections.

4 RESULTS

This section presents the findings of the study, encompassing both quantitative and qualitative analyses of the performance of the selected LLMs. The section starts by describing the hardware and model parameters utilized for the experiments; it then evaluates the adherence of the model outputs to the requested JSON format, a critical aspect of generating structured and usable results. Next, the outputs of the LLMs are quantitatively compared against a baseline (GPT-4o) using metrics such as BERTScore, BLEU, and ROUGE. Finally, a manual qualitative analysis is conducted to assess the models' ability to generate meaningful, accurate, and contextualized explanations for vulnerabilities of varying complexities.

The filtered ReposVul dataset, as well as the outputs from all LLMs used, are available online⁷.

4.1 Hardware and Model Parameters

The experiments were conducted on a virtual machine equipped with four NVIDIA A16 GPUs (16GB each, totaling 64GB VRAM), 32GB RAM, and an 8-core 2GHz CPU. The transformers library version 4.46 was used throughout the experiments.

For the CodeLlama 70B model, QLoRa 4-bit quantization was employed to accommodate the hardware limitations. Key parameters were adjusted, including the quantization type (*bnb_4bit_quant_type* set to *nf4*), compute data type (*bnb_4bit_compute_dtype* set

⁷<https://github.com/lucasg1/vulnerabilities-explanation-with-llms>

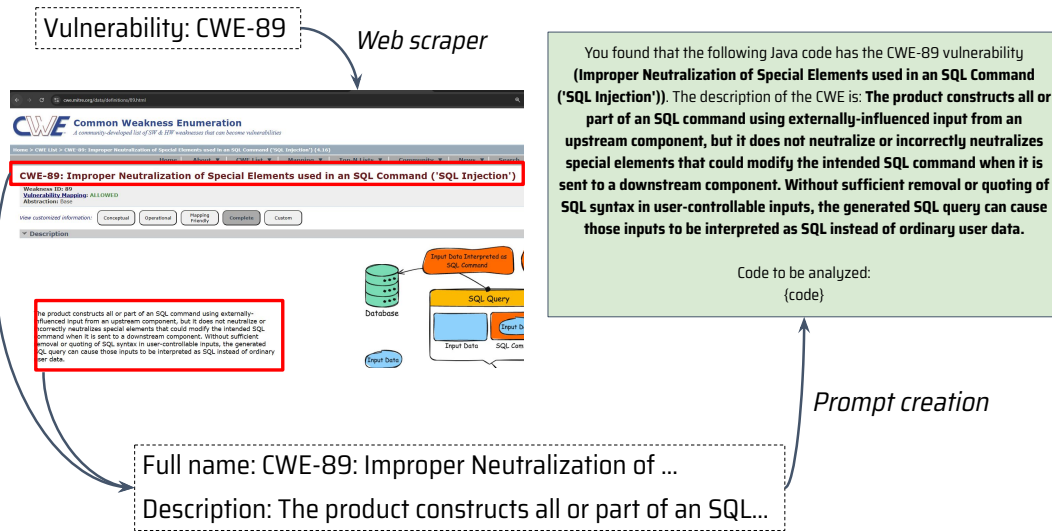


Figure 2: RAG procedure.

to *bfloat16*), and enabling double quantization (*bnb_4bit_use_double_quant=True*). Gradient checkpointing was activated to optimize memory usage while preserving precision.

For the other LLMs used in this study, the parameters were maintained at their default settings, except for the quantization configuration, which was adjusted to 8-bit (*load_in_8bit=True*). All models utilized the *bfloat16* compute data type throughout the experiments.

4.2 Outputs not Conforming to the JSON Standard

The study also evaluated the ability of LLMs to produce outputs in the requested JSON format across the 170 test cases. Smaller models, such as CodeLlama 7B and CodeGeeX4 9B, frequently deviated from the standard, likely due to limited capacity to handle strict formatting requirements. Larger models, like CodeLlama 70B, also showed higher error rates, potentially due to excessive quantization affecting output accuracy. In contrast, mid-sized models, such as CodeLlama 13B and CodeLlama 34B, demonstrated better adherence to the JSON standard, with error rates of 7.6% and 2.9%, respectively. These findings, illustrated in Figure 3, highlight the effectiveness of these models in producing structured outputs within the constraints of the study.

4.3 Comparison with Baseline (GPT-4o)

The outputs of the models were compared against GPT-4o, chosen as the baseline for its reputation

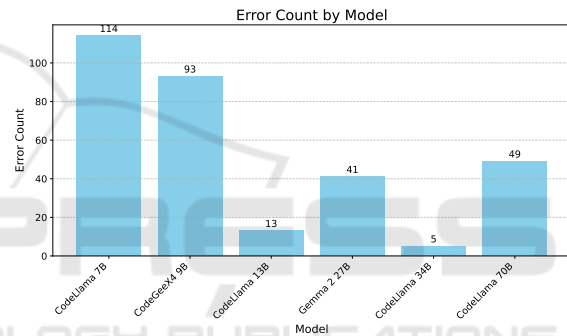


Figure 3: Number of errors for requested output format by model.

as a reliable and advanced large language model. Renowned for its superior natural language understanding and generation, GPT-4o serves as a high-quality benchmark widely used in research and industry.

The comparison employed the following metrics:

- **BERTScore** (Zhang* et al., 2020). Captures semantic similarity by embedding texts and measuring scores. BERTScore, as the main metric, reveals semantic relationships beyond word matching.
- **BLEU**. Evaluates n-gram matches between generated and reference responses, focusing on exact word sequence matches without accounting for semantics.
- **ROUGE-X**. Measures lexical overlap via unigrams (ROUGE-1), bigrams (ROUGE-2), and longest common subsequences (ROUGE-L), emphasizing word-level similarity over semantic fidelity.

While BLEU and ROUGE provide insights into lexical overlap, they fail to account for semantically equivalent expressions with different wording, a frequent occurrence in complex explanations. For instance, synonyms or rephrased structures conveying the same meaning are often overlooked by these metrics. In contrast, BERTScore, which evaluates contextual embeddings, is better suited for assessing explanations where semantic accuracy is critical.

The results of the comparison with the baseline GPT-4o are presented in Figure 4. This figure summarizes the aggregated performance of the models across the four JSON keys: *why*, *danger*, *how*, and *fix*. For each key, the metrics BLEU, ROUGE-1, ROUGE-L, and BERTScore are used to evaluate the quality of the generated outputs. The full results, including additional graphs, are available in the provided GitHub repository.

The selected metrics proved insufficient to draw meaningful comparisons between the models. The main metric, BERTScore, showed a maximum difference of only 2% across all models, highlighting the limitations of the metric in capturing nuanced differences in model performance for this task.

CodeGeeX4 performed best in semantic similarity to GPT-4o, achieving the highest overall scores. However, it exhibited significant formatting issues, with 93 responses (55%) failing to conform to the JSON standard. In contrast, CodeLlama-34B had only 5 improperly formatted outputs (3%), demonstrating superior reliability in adhering to the requested format.

These results suggest that while CodeGeeX4 excels in generating responses semantically close to GPT-4o, its high rate of formatting errors limits its practical utility. CodeLlama-34B, with its significantly lower error rate, presents itself as a more reliable option for applications requiring strict adherence to output formatting.

Due to space restrictions, an example of a CWE-400 (Uncontrolled Resource Consumption) vulnerability, CVE-2022-24839, is provided on the main page of the GitHub repository: <https://github.com/lucasg1/vulnerabilities-explanation-with-llms>. The corresponding explanation provided by two models, GPT-4o and CodeLlama 34B, is shown at the page.

4.4 Qualitative Manual Analysis

The comparison with the baseline using automated metrics proved insufficient in evaluating the models in terms of the quality of their explanations. As a result, a manual evaluation of the vulnerabilities was conducted to provide a more comprehensive analysis of the models' performance in generating meaningful

and actionable explanations. The vulnerabilities were analyzed based on the following criteria:

- **Contextualization.** Assesses how well the model connects its explanation to the provided code, including references to variables, functions, or code snippets used.
- **Clarity.** Evaluates the clarity of each explanation in terms of language and structure.
- **Identification of the Cause (why?).** Examines whether the model correctly identifies the root cause of the vulnerability in the provided code.
- **Risk Assessment (danger?).** Measures if the model accurately describes the potential dangers of the vulnerability.
- **Explanation of Exploitation (how?).** Evaluates if the model clearly and correctly explains how the vulnerability can be exploited.
- **Fix Direction (fix?).** Assesses whether the model provides practical and clear directions to fix the vulnerability.

To accomplish this, five vulnerabilities were selected for analysis, varying in complexity:

- **CWE-521.** Weak password requirements (low complexity)
- **CWE-611.** Improper restriction of XML external entity reference (medium complexity)
- **CWE-668.** Exposure of resource to wrong sphere (medium complexity)
- **CWE-79.** Improper neutralization of input during web page generation (XSS) (medium complexity)
- **CWE-203.** Observable discrepancy (high complexity)

The evaluations were conducted using the following scale:

- 1: Poor (does not meet expectations)
- 3: Average (partially meets expectations)
- 5: Excellent (fully meets expectations)

This analysis enhances understanding of the models' qualitative performance and their ability to address vulnerabilities with structured and actionable explanations.

Table 2 shows GPT-4o achieved the highest average score of 3.9, as expected, given its status as the baseline and a state-of-the-art model. CodeLlama 34B and Gemma 2 27B followed with scores of 3.3 and 3.6, demonstrating comparable performance and alignment with evaluation criteria.

Smaller models like CodeLlama 13B (2.7), CodeLlama 7B (3.0), and CodeGeeX4 9B (3.2)

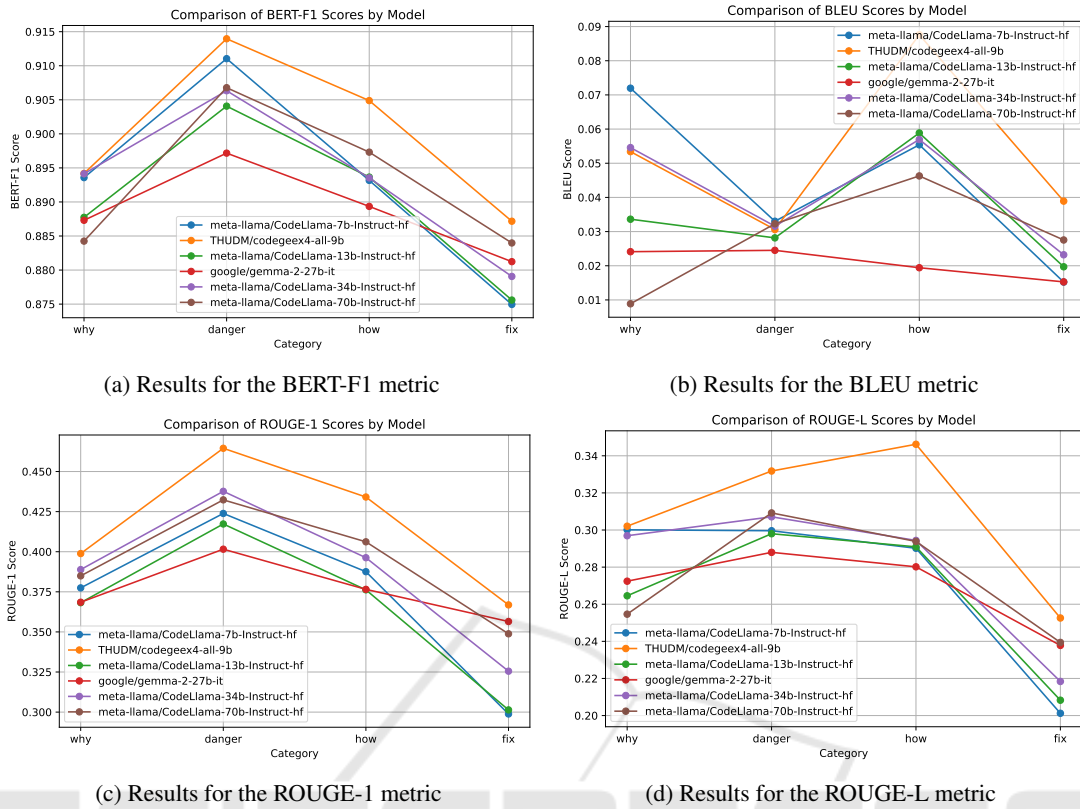


Figure 4: Results for the comparison with the baseline GPT-4o. Each image in the grid is the aggregated result for one of the JSON keys of the generated outputs with the metrics BLEU, ROUGE-X, and BERTScore.

Table 2: Qualitative manual analysis results. The average score represents the mean value of the manual evaluation scores across all criteria and all five analyzed cases for each model.

Model	Average score
GPT-4o	3.9
CodeLlama 7B	3.0
CodeLlama 13B	2.7
CodeLlama 34B	3.3
CodeLlama 70B	3.0
Gemma 2 27B	3.6
CodeGeeX4 9B	3.2

scored lower. Although differences between top performers and smaller models are evident, they are not significant enough to make these smaller models completely unsuitable. Medium-sized models, such as CodeLlama 34B and Gemma 2 27B, performed better, likely due to reduced quantization and fewer memory limitations.

These results show that model size does not solely determine performance. CodeLlama 70B did not surpass mid-sized models like CodeLlama 34B using the provided metrics, raising questions about the impact of quantization. Better hardware resources are needed

to assess whether hardware limitations affected the larger model’s potential.

5 CONCLUSIONS

As reliance on secure software systems grows, addressing vulnerabilities is crucial to safeguarding infrastructure. Traditional static and dynamic analysis tools, while effective for detection, often lack contextualized explanations needed by developers and security practitioners. This study demonstrates how LLMs can bridge this gap by generating structured, contextualized explanations for Java code vulnerabilities. By addressing the why, danger, how, and fix dimensions, the research highlights both the potential and limitations of LLMs in this domain.

Among the evaluated models, CodeLlama 34B performed best, particularly in generating structured outputs with minimal formatting errors. However, all models, including GPT-4o, struggled with providing comprehensive explanations for complex vulnerabilities, often failing to contextualize vulnerabilities within the provided code.

A key limitation of this study lies in the inadequacy of BLEU, ROUGE, and BERTScore metrics for evaluating vulnerability explanations. These metrics, standard in NLP tasks, fail to capture nuances relevant to this domain. Additionally, hardware challenges, such as VRAM limitations, posed significant obstacles during model execution, highlighting the need for robust memory resources.

Problems Found. Various operational challenges were observed. The Gemma-2 27B model frequently encountered memory issues, requiring manual restarts. Similarly, the CodeGeeX4 9B model exhibited inconsistent response times, often taking excessively long to generate outputs, leading to the implementation of a 150-second execution time limit. Many models also occasionally produced unusable outputs, such as repeated line breaks or redundant phrases.

Future Works. Future research should focus on fine-tuning models, though larger models will require more memory for this process. Exploring alternative RAG techniques and their impact on the quality of explanations is another promising direction. Additionally, explanations, particularly the `why?` and `fix?` components, could support tasks aimed at automating vulnerability repair. Investigating collaborative reasoning techniques, where multiple LLMs interact to produce more contextualized explanations, is another avenue. Finally, surveying experienced programmers for their opinions on model performance could provide valuable insights into practical applications and user preferences.

REFERENCES

- Chen, Y., Ding, Z., Alowain, L., Chen, X., and Wagner, D. (2023). DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *PROCEEDINGS OF THE 26TH INTERNATIONAL SYMPOSIUM ON RESEARCH IN ATTACKS, INTRUSIONS AND DEFENSES, RAID 2023*, pages 654–668, New York. Assoc Computing Machinery.
- Fu, M. and Tantithamthavorn, C. (2022). LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620.
- Fu, M., Tantithamthavorn, C., Le, T., Nguyen, V., and Phung, D. (2022). VulRepair: A T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 935–947, New York, NY, USA. Association for Computing Machinery.
- Hin, D., Kan, A., Chen, H., and Babar, M. A. (2022). LineVD: Statement-level Vulnerability Detection using Graph Neural Networks. In *2022 MINING SOFTWARE REPOSITORIES CONFERENCE (MSR 2022)*, MSR '22, pages 596–607, Los Alamitos. IEEE Computer Soc.
- Hu, S., Huang, T., Ilhan, F., Tekin, S. F., and Liu, L. (2023). Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. In *2023 5TH IEEE INTERNATIONAL CONFERENCE ON TRUST, PRIVACY AND SECURITY IN INTELLIGENT SYSTEMS AND APPLICATIONS, TPS-ISA*, pages 297–306, New York. IEEE.
- Nguyen, V.-A., Nguyen, D. Q., Nguyen, V., Le, T., Tran, Q. H., and Phung, D. (2022). ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 178–182.
- Rozière, B. et al. (2024). Code llama: Open foundation models for code.
- Team, G. et al. (2024). Gemma 2: Improving open language models at a practical size.
- Wang, X., Hu, R., Gao, C., Wen, X.-C., Chen, Y., and Liao, Q. (2024). Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE-Companion '24*, page 472–483, New York, NY, USA. Association for Computing Machinery.
- Wu, Y., Jiang, N., Pham, H. V., Lutellier, T., Davis, J., Tan, L., Babkin, P., and Shah, S. (2023). How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*, pages 1282–1294, New York, NY, USA. Association for Computing Machinery.
- Zhang, Q., Fang, C., Yu, B., Sun, W., Zhang, T., and Chen, Z. (2024). Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We? *IEEE Transactions on Dependable and Secure Computing*, 21(4):2507–2525.
- Zhang*, T., Kishore*, V., Wu*, F., Weinberger, K. Q., and Artzi, Y. (2020). Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.
- Zheng, Q. et al. (2023). Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.