


Can Open Large Language Models Catch Vulnerabilities?

Diogo Gaspar Lopes 

Instituto Universitário de Lisboa (ISCTE-IUL), Portugal

Tiago Espinha Gasiba 

Siemens AG, Munich, Germany

Sathwik Amburi 

Siemens AG, Munich, Germany

Maria Pinto-Albuquerque 

Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Portugal

Abstract

As Large Language Models (LLMs) become increasingly integrated into secure software development workflows, a critical question remains unanswered: can these models not only detect insecure code but also reliably classify vulnerabilities according to standardized taxonomies? In this work, we conduct a systematic evaluation of three state-of-the-art LLMs – Llama3, Codestral, and Deepseek R1 – using a carefully filtered subset of the Big-Vul dataset annotated with eight representative Common Weakness Enumeration categories. Adopting a closed-world classification setup, we assess each model's performance in both identifying the presence of vulnerabilities and mapping them to the correct CWE label. Our findings reveal a sharp contrast between high detection rates and markedly poor classification accuracy, with frequent overgeneralization and misclassification. Moreover, we analyze model-specific biases and common failure modes, shedding light on the limitations of current LLMs in performing fine-grained security reasoning. These insights are especially relevant in educational contexts, where LLMs are being adopted as learning aids despite their limitations. A nuanced understanding of their behaviour is essential to prevent the propagation of misconceptions among students. Our results expose key challenges that must be addressed before LLMs can be reliably deployed in security-sensitive environments.

2012 ACM Subject Classification Security and privacy → Software security engineering; Computing methodologies → Machine learning; Software and its engineering → Software testing and debugging

Keywords and phrases Large Language Models (LLMs), Secure Coding, CWE Classification, Machine Learning, Software Vulnerability Detection, Artificial Intelligence, Code Analysis, Big-Vul Dataset

Digital Object Identifier 10.4230/OASICS.ICPEC.2025.4

Funding This work is partially financed by Portuguese national funds through FCT – Fundação para a Ciência e Tecnologia, I.P., under the projects FCT UIDB/04466/2020 and FCT UIDP/04466/2020. Furthermore, Maria Pinto-Albuquerque thanks the Instituto Universitário de Lisboa and ISTAR, for their support.

1 Introduction

In recent years, the integration of Large Language Models (LLMs) into software security workflows has gained increasing attention. Traditional methods for vulnerability detection – such as static and dynamic analysis – often suffer from limited scalability, rigid rule sets, and poor adaptability to novel coding patterns. These limitations have motivated a surge in the use of machine learning techniques, with LLMs emerging as promising candidates due to their ability to reason over both natural language and source code.



© Diogo Gaspar Lopes, Tiago Espinha Gasiba, Sathwik Amburi, and Maria Pinto-Albuquerque; licensed under Creative Commons License CC-BY 4.0

6th International Computer Programming Education Conference (ICPEC 2025).

Editors: Ricardo Queirós, Mário Pinto, Filipe Portela, and Alberto Simões; Article No. 4; pp. 4:1–4:14

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Despite their potential, the application of LLMs to secure software development and their integration into educational contexts for teaching secure coding practices remain topics of ongoing investigation. While existing studies have demonstrated that LLMs can effectively identify insecure code, their performance in vulnerability classification – particularly when mapped to the Common Weakness Enumeration (CWE) [8] taxonomy – is considerably less understood. This task presents a complex challenge, requiring models to distinguish between semantically similar yet distinct categories, often under conditions of sparse contextual information.

In this paper, we present an empirical evaluation of three state-of-the-art LLMs – Llama3, Codestral, and Deepseek R1. Using a filtered subset of the Big-Vul dataset, we assess each model’s capacity to both identify the presence of a vulnerability and classify it into one of eight representative CWE categories. Our experimental setup enforces a closed-world assumption to reduce ambiguity and enable controlled comparison across models. We further analyze typical error patterns, including overprediction and misclassification, to better understand the practical limitations of current-generation LLMs in security-critical tasks.

A critical gap in current research lies in the discrepancy between a model’s ability to detect insecure code and its competence in accurately classifying the nature of the vulnerability. Misclassification can lead to incorrect prioritization, flawed remediation efforts, and an inflated sense of coverage. In this context, our study investigates how well open-source LLMs perform when constrained to a fixed taxonomy, enabling a focused analysis of their classification behaviors, error modes, and practical limitations. By doing so, we offer a more granular assessment of their readiness for integration into security workflows and educational settings, by providing a comprehensive analysis of the benefits, drawbacks, and inherent limitations associated with employing machine learning as a strategy to review and understand secure code.

2 Related Work

The integration of Large Language Models (LLMs) into software security processes has attracted significant attention in recent years. Traditional vulnerability detection methods, such as static and dynamic analysis, often struggle with scalability, noise, and adaptability to evolving codebases. These limitations have motivated the exploration of machine learning and, more recently, the application of LLMs, which offer the ability to reason over natural language and code simultaneously, across different programming languages and patterns.

Recent studies have systematically reviewed the application of LLMs in vulnerability detection tasks. Khare et al. [7] evaluated 16 pre-trained LLMs across diverse datasets and programming languages. Their findings suggest that while LLMs perform well on intra-procedural vulnerabilities, their effectiveness drops when deeper semantic understanding or inter-procedural reasoning is required. Sheng et al. [13] provide a complementary perspective through an extensive analysis of the application of LLMs in vulnerability detection, discussing various model architectures, datasets, and evaluation metrics. Their work highlights the potential of LLMs to outperform traditional methods in certain contexts, especially when dealing with complex or obfuscated code structures.

The classification of vulnerabilities according to the Common Weakness Enumeration (CWE) taxonomy remains particularly challenging. A previous study conducted by Gasiba et al. [4] examined ChatGPT’s capability to aid developers in secure coding, including its classification performance across several real-world vulnerabilities. Using an experimental setup based on serious games and secure coding challenges, the authors observed that while

ChatGPT could often detect the presence of a vulnerability, it frequently misclassified the CWE. This underscores a critical limitation of current LLMs: their inability to consistently differentiate between types of vulnerabilities.

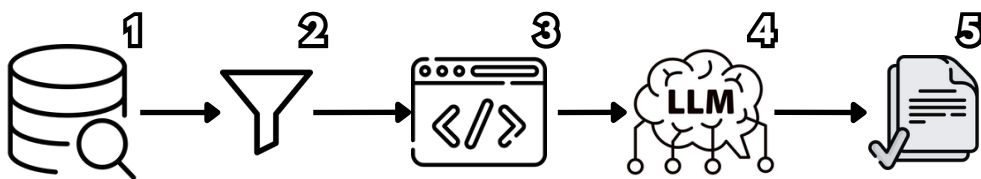
Other studies have explored how LLMs compare to traditional approaches. Zhou et al. [19] conducted a comparative analysis of LLMs and Static Application Security Testing (SAST) tools, finding that LLMs were better at detecting issues in natural language-heavy contexts, while SAST tools excelled in low-level structural patterns. Similarly, Walker et al. [16] highlight the limitations of modern static analysis tools, including the lack of concise overviews, multi-repository support, and standardized integration mechanisms. These challenges hinder developers from effectively utilizing static analysis in their workflows, prompting the need for more adaptable and user-friendly solutions.

Following this idea, an additional crucial aspect is the usability of LLM-powered tools. Vaithilingam et al. [15] found a significant disparity between developer expectations and actual experience when using LLM-based code assistants. Both developers and learners often trust LLM outputs even when they are incorrect, which raises concerns about their reliability and the potential for misinformation in educational settings – a trend supported by recent studies showing that students frequently rely on generative AI tools without fully understanding their limitations: “LLMs are generally better at writing code than answering conceptual or domain-specific questions.” [11].

Taken together, these studies demonstrate both the promise and limitations of using LLMs for vulnerability detection and classification. Despite progress in model capabilities, data accessibility and prompt engineering, significant challenges remain. These challenges encompass classification accuracy, robustness to context omission, model hallucination, and the difficulty of benchmarking LLM performance in realistic software engineering workflows.

3 Experiment

To evaluate the ability of Large Language Models to support secure software development, we conducted an experiment based on real-world C/C++ code snippets that included both vulnerable and non-vulnerable examples. The objective was to assess the extent to which LLMs can (i) detect the presence of a vulnerability and (ii) correctly classify it according to the CWE taxonomy.



■ **Figure 1** Flow diagram illustrating the complete process.

The figure 1 outlines the five sequential phases involved in the process of data selection, data processing, LLM interaction and answer collecting and classification.

We selected and used the publicly available **Big-Vul** dataset [5] -first phase. This data collection consists of over 20,000 real-world functions extracted from open-source GitHub projects, associated with vulnerabilities disclosed in the period from 2010 to 2019. Each entry in the dataset corresponds to either a vulnerable code snippet, paired with its patched version,

or a non-vulnerable snippet. The vulnerable snippets are labelled with their corresponding CWE identifier, making the dataset well-suited for our classification analysis. This makes Big-Vul a comprehensive and representative database for vulnerability detection tasks [5].

The dataset contains code snippets of varying lengths and complexity, which will, in the future, allows us to investigate how snippet size impacts model performance. As highlighted by Ding et al., longer and more complex code snippets can challenge the capabilities of code language models, potentially affecting their accuracy in vulnerability detection tasks [3].

3.1 Dataset Filtering and CWE Selection

During preliminary testing, we observed that LLMs often produced highly diverse and inconsistent CWE predictions for the same code snippet. We hypothesized that this behaviour could stem from the high complexity and overlap between many CWE categories.

To mitigate this, we constrained the classification space by selecting only those CWEs in the dataset that were better represented and more consistently labelled -second phase. Specifically, we selected all CWEs for which the dataset contained at least **100 unique vulnerable snippets**. This filtering process resulted in a total of **8 distinct CWEs**, each with 100 representative examples. This filtering resulted in the following 8 CWEs:

■ **Table 1** List of CWE Categories Included in the Dataset.

CWE Description
CWE-20: Improper Input Validation
CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-125: Out-of-bounds Read
CWE-190: Integer Overflow or Wraparound
CWE-200: Exposure of Sensitive Information to an Unauthorized Actor
CWE-416: Use After Free
CWE-476: NULL Pointer Dereference
CWE-787: Out-of-bounds Write

To complement the set of vulnerable code snippets, we also included **100 safe snippets**, code samples without any known vulnerabilities, also drawn from the same dataset.

The final dataset used for experimentation thus consisted of **900 code snippets**: 800 vulnerable examples (distributed equally across 8 CWEs), and 100 non-vulnerable examples. All code snippets were selected randomly.

3.2 Large Language Models

The experiment was conducted using three state-of-the-art LLMs:

- **Llama 3 (Meta) [12]:** a general-purpose decoder-only model with 70 billion parameters, designed for language understanding and generation.
- **Codestral (Mistral) [14]:** a code-centric model trained on a diverse set of permissively licensed codebases.
- **Deepseek R1 (Deepseek) [2]:** a general-purpose model that employs a “thought process” decoding mechanism to enhance reasoning capabilities and improve performance on complex programming tasks.

With this selection, we aim to cover a representative spectrum of contemporary LLM architectures and specializations, ranging from general-purpose language models to code-specific systems. Importantly, all models included in this study are open source, which ensures transparency, reproducibility, and accessibility for the research community.

3.3 Input Procedure

Here we started our third phase, where to constrain the classification space and better assess model accuracy, we provided each LLM with a fixed list of **8 CWE identifiers**, corresponding to the vulnerabilities present in the dataset. Our prompt included this list and instructed the model to select the most appropriate CWE based on the code snippet provided. This design ensures that the task is framed as a *closed-set classification* problem, rather than free-text generation, enabling a measurable comparison across models.

All models were queried independently using standardised prompts in isolated sessions, with no memory of previous queries.

Each model received the same structured prompt, designed to simulate a realistic developer scenario. The prompt consists of three components:

1. An introduction asking the model to determine whether the code is secure;
2. A list of 8 CWE identifiers from which the model must choose the most appropriate one, if a vulnerability is present.
3. A code snippet (in C or C++);

For non-vulnerable snippets, the model was expected to explicitly state that no vulnerability was present.

3.4 Prompt Engineering

This is our fourth phase where to ensure consistency and minimize variability across models, a carefully designed prompt engineering strategy was adopted. The primary objective was to ensure that all models received the exact same task, with clear and explicit rules to avoid ambiguity.

In this context, the prompt was designed to enforce that models:

- Analyze a code snippet and determine whether it contains a vulnerability.
- If a vulnerability exists, identify the **most appropriate CWE** from a given list.
- Return **only one CWE identifier** from the list or the quote “There isn’t” if no vulnerability is found.
- Never suggest CWEs that are not in the provided list.

After multiple interactions and different results, we arrived at a prompt structured in two parts: a base instruction, common to all examples, and a second block with composed of a CWE list and the code snippet. The complete base prompt used in all tests is shown below:

```
You are a cybersecurity expert specialized in vulnerability
detection.
You are given:
- A list of CWE identifiers (e.g., [CWE-79, CWE-89, CWE-200]).
- A code snippet to analyse.

Your task:
- If the code snippet contains any vulnerability, you must map
  it to the most appropriate CWE from the provided list and
  output only
  that CWE identifier (e.g., CWE-79).
- You must always select a CWE from the provided list if there
  is any vulnerability, even if the match is approximate.
- If the code snippet contains no vulnerabilities at all,
  output exactly: There isn't.
```

4:6 Can Open Large Language Models Catch Vulnerabilities?

```
Rules:
- Do not output any explanation, justification, description, or
  additional text.
- Do not suggest vulnerabilities outside the given CWE list.
- If multiple CWEs could apply, choose the best matching one.
- Output strictly a single CWE identifier from the list or '
  There isn't'.
```

This block was followed by a prompt containing the specific data for each example, such as the list of CWEs and the code to analyse:

```
Analyze the following code snippet for vulnerabilities.

Do not explain your answer. Output only the CWE identifier or '
  There isn't'.

-CWE List:

-Code Snippet:
```

This approach enabled an objective and controlled evaluation of model performance, minimizing interference from external factors such as model verbosity or formatting variation.

3.5 Evaluation Metrics

In this last phase, we collected each answer and evaluated each model using two distinct metrics:

- **Blind Detection:** Whether the model was able to detect the presence of a vulnerability, of any kind, in the code snippet. In this process, we refined the response of each LLM to derive a binary outcome, indicating the detection or absence of vulnerabilities.
- **Specific Detection:** Whether the model was able to correctly classify the vulnerability using the appropriate CWE identifier

These metrics allow us to distinguish between the model's general ability to detect insecurities and its ability to correctly classify known weaknesses.

4 Results

4.1 Detection of Vulnerabilities - Blind Detection

We first assessed each model's ability to detect the presence of vulnerabilities. That is, could the model identify that a given code snippet was problematic? In Figure 2, we present the performance of all three models in identifying *True Positives* – focusing exclusively on vulnerable code snippets.

The three models demonstrated high rates for blind detection. Llama achieved a score of 97.4%, followed by Deepseek with 84.9% and Codestral with 83.2%. These results suggest that LLMs might be highly sensitive to the presence of insecure patterns in code, and capable of flagging problematic input in most cases.

To assess the extent to which the models produce *False Positives* during blind detection, we evaluated them on a set of secure code snippets that had been carefully curated to contain no known vulnerabilities.

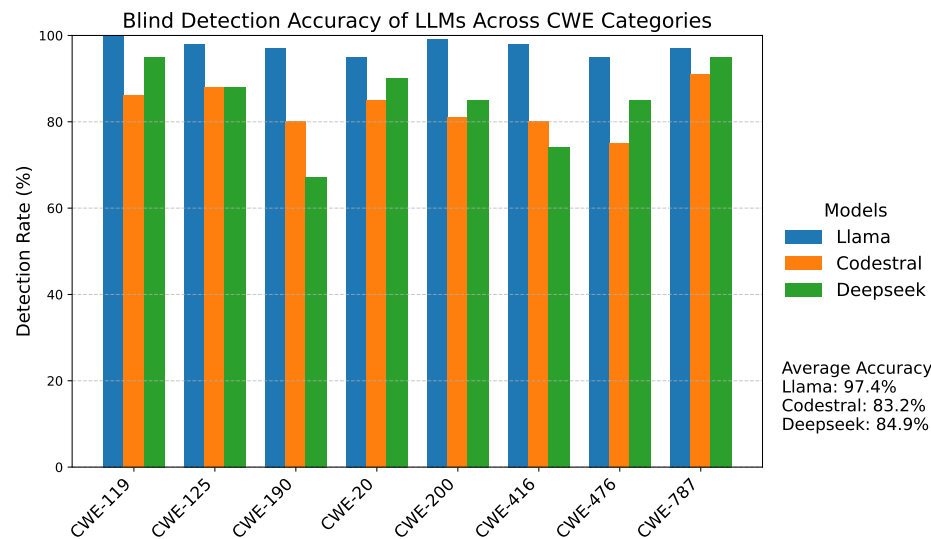


Figure 2 True positive rate across the LLM models, representing the percentage of insecure code snippets correctly identified as vulnerable.

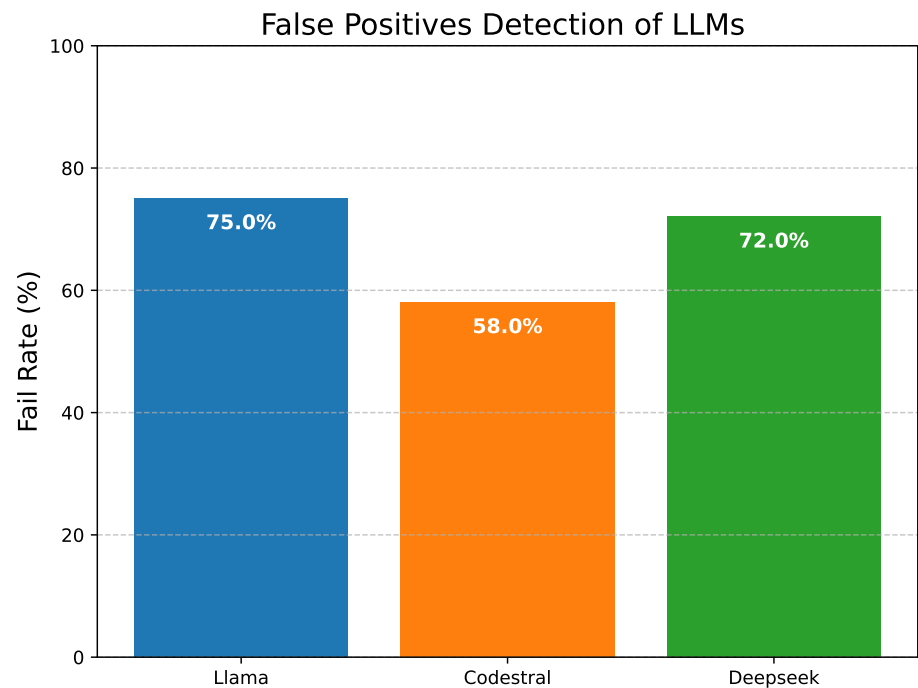


Figure 3 False positive rate across the LLM models, representing the percentage of secure code snippets incorrectly identified as vulnerable.

4:8 Can Open Large Language Models Catch Vulnerabilities?

As shown in Figure 3, the models frequently misclassified safe code as vulnerable. Llama incorrectly flagged 75% of the safe snippets, followed by Deepseek (72%) and Codestral (58%). These results highlight a tendency among LLMs to overpredict vulnerabilities when operating without guidance, raising concerns about their reliability in real-world scenarios where precision is crucial.

These findings suggest that the LLMs are highly cautious – perhaps excessively. Their sensitivity to potential vulnerabilities can lead to false positives, where secure code is mistakenly flagged as problematic. This behaviour, while arguably safer in exploratory analysis, can erode developer trust in the tool, especially when integrated into real-world workflows where the majority of code may not be vulnerable.

High Detection, Low Precision

All models achieved high detection rates for insecure code – up to 97.4% – but also exhibited alarmingly high false positive rates, flagging more than 50% of secure snippets as vulnerable.

4.2 Identification of Vulnerabilities - Specific Detection

Understanding whether language models can not only detect the presence of vulnerabilities but also correctly classify them is critical for evaluating their practical utility in secure software development. In this subsection, we present our results of the models' ability to perform this more fine-grained task.

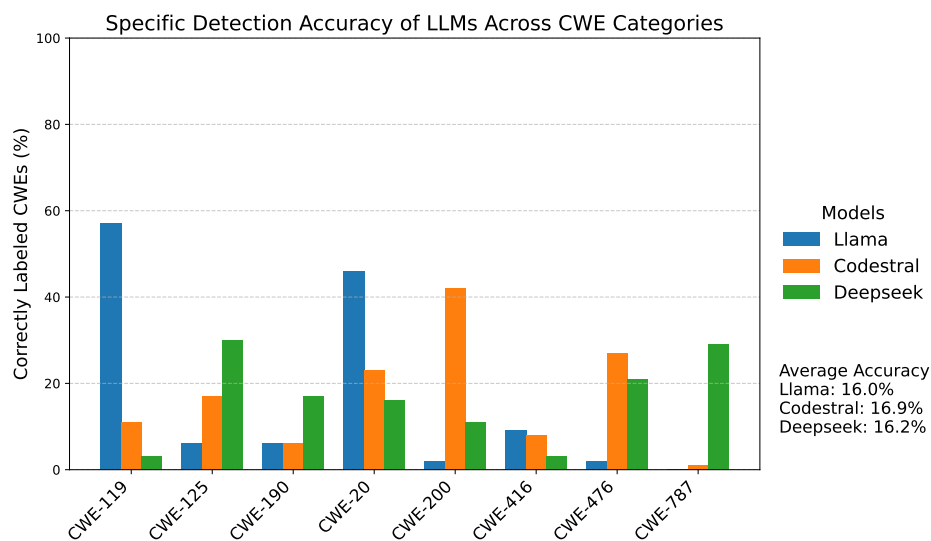


Figure 4 Accuracy of LLMs in correctly labeling vulnerabilities with their corresponding CWE categories.

As shown in Figure 4, the models' ability to perform **specific detection** drops significantly when compared with blind detection. All models performed similarly: Llama with 16.0%, Codestral with 16.9%, and Deepseek with 16.2%. This suggests that while LLMs may be able to recognize the presence of a vulnerability of any kind, they often struggle to accurately map it to its precise classification within the CWE taxonomy.

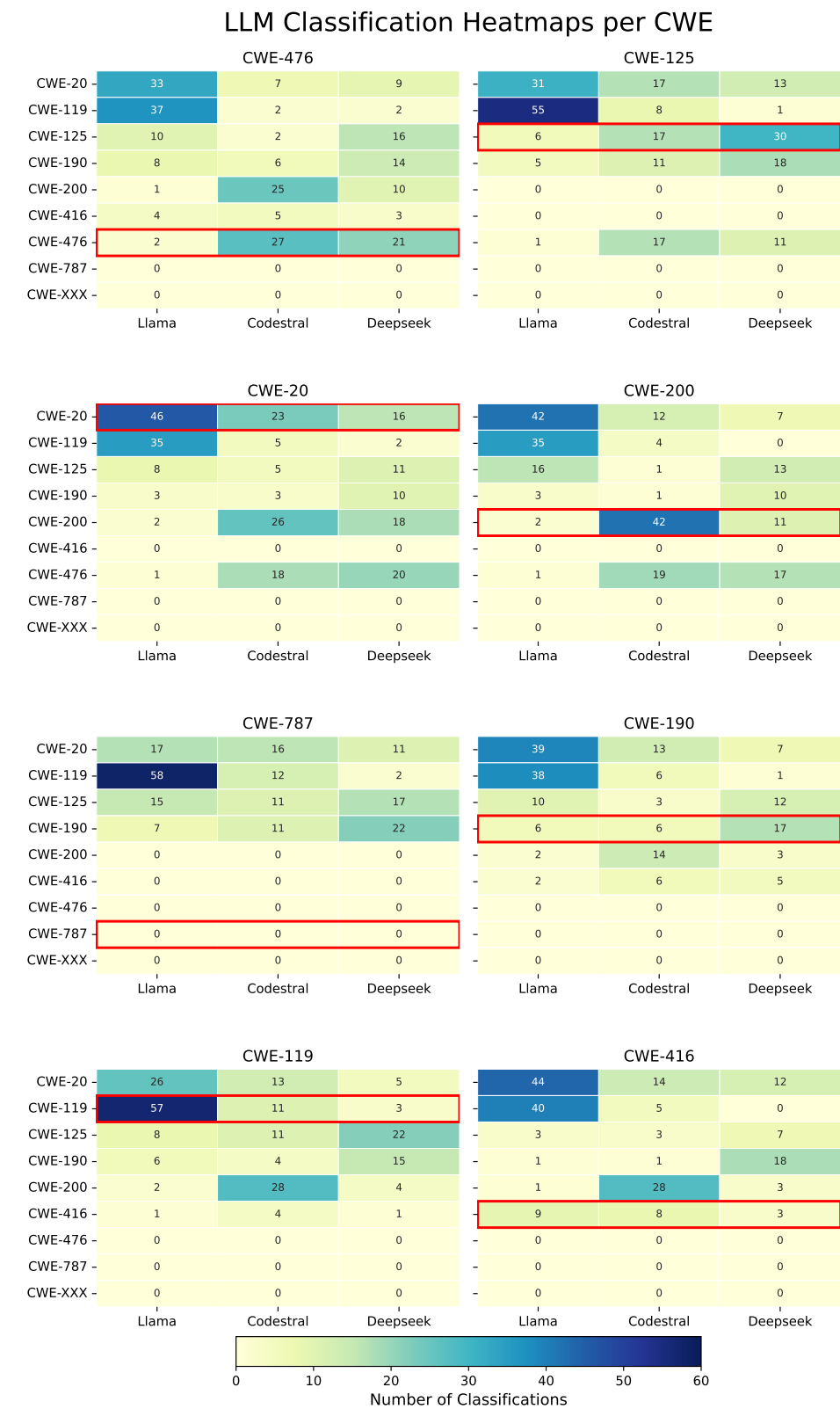


Figure 5 Distribution of the classification made by each LLM, grouped by the CWE ground truth.

4:10 Can Open Large Language Models Catch Vulnerabilities?

To further investigate the classification patterns of each model, we analysed the full distribution of CWE predictions using heatmaps. Each heatmap focuses on one ground truth CWE and shows how frequently each model predicted each CWE as a response. The Y-axis lists all possible CWE predictions, while the X-axis corresponds to the three LLMs. We included an additional row labeled “**CWE-XXX**”, which is intended to capture answers outside the provided list, serving to identify hallucinations or non-compliance with the prompt.

From a high-level perspective, the heatmaps reveal substantial differences in behaviour among the models, as well as important commonalities. A striking observation is that none of the three models ever predicted an invalid CWE label such as **CWE-XXX**. This indicates that, despite their limitations, the models did not hallucinate syntactically incorrect or non-existent CWEs. The absence of such hallucinations suggests that the models exhibit a degree of robustness in adhering to the expected CWE format when prompted explicitly.

At a finer level, the heatmaps show that the Llama model exhibits a pronounced and systematic bias towards predicting CWE-20 (Improper Input Validation) and CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), regardless of the actual ground truth. This behaviour points to overfitting to dominant patterns learned during training or to an insufficient understanding of the distinguishing features required to differentiate between CWE types. While this leads to a high frequency of predictions in those two categories, it also results in extremely poor performance for the remaining CWEs, where the model seldom assigns the correct label. Consequently, beyond CWE-20 and CWE-119, Llama’s classification capability appears substantially limited.

Another relevant insight is that none of the models managed to predict CWE-787 (Out-of-bounds Write) for any example, even when it was the correct label. This consistent omission suggests that this vulnerability type poses a particular challenge to LLMs, either due to a lack of representative training data or due to its subtle distinction from related memory issues such as CWE-119 or CWE-125.

When comparing the models more closely, Codestral and Deepseek emerged as the most similar in terms of their prediction distributions. While not identical, their heatmaps display comparable misclassification patterns and similar concentration zones, implying overlapping decision heuristics or aligned training data characteristics. In contrast, Llama’s distribution stands out as significantly skewed and less nuanced.

Among individual model strengths, Codestral demonstrated a comparatively strong ability to correctly identify CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor), achieving consistent matches in relevant cases. This suggests that Codestral may possess a better contextual understanding of confidentiality-related issues, or may have benefited from more robust examples in pretraining data for that class.

Overall, the heatmap analysis exposes several critical limitations in the CWE prediction capabilities of current LLMs, particularly in terms of overgeneralization and lack of coverage for less common but equally important vulnerabilities.

Classification Accuracy Remains Elusive

Despite successfully detecting vulnerabilities, all models struggled to assign the correct CWE label – achieving only ~16% accuracy – highlighting major limitations in fine-grained vulnerability classification.

5 Discussion

Starting from the blind detection task, our results suggest that large language models exhibit an overly cautious behaviour, or alternatively, that the prompt design may have inadvertently biased the models towards assuming that the input snippets were vulnerable. This interpretation is supported by the exceptionally high detection rates obtained across snippets containing known CWEs. Notably, Llama demonstrated perfect recall for CWE-119, correctly identifying every corresponding snippet as vulnerable. However, this heightened sensitivity comes at a cost: when presented with snippets that do not contain any associated vulnerabilities, all models produced an alarming number of false positives – exceeding 50% in every case. In practical terms, this means that when it comes to classifying secure code, these LLMs, in this conditions, perform worse than random guessing.

These findings cast doubt on the utility of LLMs for automated vulnerability screening in production settings. High false positive rates can lead to wasted developer time and mistrust in the tools, ultimately defeating the purpose of integrating AI-based support into secure development workflows. Similar concerns have been observed in educational contexts, where LLM-generated feedback on programming tasks can be misleading or overly generic, failing to support meaningful learning [9].

Regarding the second task – specific CWE classification – the models performed considerably worse, as expected. This task requires not only detecting the presence of a vulnerability but also reasoning about its type, which poses a much more demanding challenge for current models. As the heatmaps reveal, certain anomalies raise important questions. For example, Code Llama exhibits a strong bias toward predicting CWE-119 and CWE-20. One plausible explanation is that these CWEs appeared early in the provided list, potentially influencing the model’s token-level attention and completion patterns [1]. This raises concerns about prompt-induced biases in multi-label classification settings.

In the other way, both Codestral and DeepSeek performed far below expectations in this task, correctly labelling only 16.9% and 16.2% snippets respectively. These figures suggest a lack of deeper semantic understanding of code on the part of the models, or at the very least, an inability to map such understanding to precise CWE identifiers. This undermines the feasibility of relying on LLMs not only for vulnerability detection but also for accurate categorization – an essential requirement for integrating such models into security tools like SASTs or vulnerability triaging systems [10].

Altogether, our results suggest that while LLMs show promise in recognizing that “something might be wrong” in a code snippet, they struggle considerably with determining exactly *what* is wrong. This limitation, combined with the high rate of false positives, calls into question their practical applicability for precise and trustworthy vulnerability identification, for example, educators should emphasize the exploratory role of LLMs and actively teach students to critically assess their suggestions. Likewise, tool designers should avoid presenting model predictions as final or fully reliable, and instead incorporate contextual cues or scaffolding to help users interpret the results appropriately. Without such caution, there is a risk of reinforcing misconceptions rather than promoting secure coding literacy.

5.1 Threats to Validity

The present study relies on three publicly available LLMs – Llama3, Codestral, and Deepseek R1 – as released and accessible at the time of writing. As these models are still undergoing rapid development and refinement, future iterations may exhibit significantly different behaviours. Consequently, some of the observations and conclusions presented in this work may not generalize to later versions of the same models.

Another threat to validity arises from the ambiguous classification of CWEs and the pre-existing classifications in the dataset, which may contain inaccuracies. While the closed-world classification setting facilitates clearer performance comparisons across models, it oversimplifies the problem space and may not fully capture the complexity of real-world conditions, where multiple or overlapping vulnerabilities can exist.

Additionally, our dataset is composed of vulnerabilities disclosed between 2010 and 2019. Although the Big-Vul dataset remains a valuable resource, it may not accurately represent current programming practices, libraries, or emerging vulnerability patterns. As such, results obtained on this dataset may not extrapolate to newer or more diverse codebases.

An additional concern is that the models may have been trained on the same dataset used in this study, which could lead to biased responses. Such bias arises when the model encounters information that closely resembles what is present in the dataset, possibly reinforcing pre-existing patterns or associations that do not truly capture the diversity of real-world vulnerabilities.

Finally, prompt formulation and evaluation criteria, although carefully designed, may still introduce unintentional biases. Variations in phrasing, ordering of CWE options, or even formatting choices may influence model responses in subtle but impactful ways. While efforts were made to minimize these effects, their complete elimination cannot be guaranteed.

6 Conclusion

Our findings confirm that LLMs are effective at blind detection the presence of insecure code but remain unreliable in specific detection according to the CWE taxonomy. This unreliability, characterized by a high rate of false positives, underscores the need for sustainable and ethical approaches in integrating LLMs into educational tools for teaching secure software development [18].

These results are consistent with recent studies [6, 17] that report similar gaps in semantic understanding and classification accuracy. While LLMs exhibit syntactic robustness, their overgeneralization and high false positive rates limit their current applicability in security-critical workflows. Improving precision and reliability remains essential for integrating LLMs into practical secure development pipelines.

Ultimately, this paper reinforces the need for critical scrutiny when interpreting LLM-generated vulnerability assessments. Their responses must be treated as tentative hypotheses – subject to human verification – rather than definitive judgments. Relying on these models without critical oversight risks propagating errors, fostering misplaced confidence, and undermining both secure development practices and educational goals.

In the future, we will evaluate and further explore the potential causes of the bias exhibited by Code Llama, employing a different ordering of prompts and analyzing whether the resulting outputs follow a similar pattern to those observed in the current findings.

Future efforts must address these shortcomings by refining model training with curated vulnerability datasets, exploring hybrid architectures that combine symbolic and neural reasoning, and establishing standardized benchmarks for secure code classification. Moreover, integrating human-in-the-loop validation mechanisms may help mitigate misclassification risks in both educational and industrial deployments. As LLMs continue to evolve, bridging the gap between surface-level detection and context-aware classification will be essential to realize their full potential in automated vulnerability management pipelines.

References

- 1 Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf.
- 2 DeepSeek-AI et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. [arXiv:2501.12948](https://arxiv.org/abs/2501.12948).
- 3 Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability Detection with Code Language Models: How Far are We? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1729–1741, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi:10.1109/ICSE55347.2025.00038.
- 4 Tiago Espinha Gasiba, Kaan Oguzhan, Ibrahim Kessba, Ulrike Lechner, and Maria Pinto-Albuquerque. I’m Sorry Dave, I’m Afraid I Can’t Fix Your Code: On ChatGPT, CyberSecurity, and Secure Coding. In Ricardo Alexandre Peixoto de Queirós and Mário Paulo Teixeira Pinto, editors, *4th International Computer Programming Education Conference (ICPEC 2023)*, volume 112 of *Open Access Series in Informatics (OASIs)*, pages 2:1–2:12, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIs.ICPEC.2023.2.
- 5 Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 508–512. ACM, 2020. doi:10.1145/3379597.3387501.
- 6 Mohamed Amine Ferrag, Ammar Battah, Norbert Tihanyi, Ridhi Jain, Diana Maimuț, Fatima Alwahedi, Thierry Lestable, Narinderjit Singh Thandi, Abdechakour Mechri, Merouane Debbah, and Lucas C. Cordeiro. Securefalcon: Are we there yet in automated software vulnerability detection with llms?, 2025. doi:10.1109/TSE.2025.3548168.
- 7 Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities, 2024. [arXiv:2311.16169](https://arxiv.org/abs/2311.16169).
- 8 MITRE Corporation. Common Weakness Enumeration (CWE), 2024. URL: <https://cwe.mitre.org>.
- 9 Maciej Pankiewicz and Ryan S. Baker. Large language models (gpt) for automating feedback on programming assignments. 2023. [arXiv:2307.00150](https://arxiv.org/abs/2307.00150).
- 10 Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot’s code contributions. 2021. [arXiv:2108.09293](https://arxiv.org/abs/2108.09293).
- 11 Nishat Raihan, Mohammed Latif Siddiq, Joanna C. S. Santos, and Marcos Zampieri. Large language models in computer science education: A systematic literature review. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education (SIGCSE TS 2025)*, pages 938–944, Pittsburgh, PA, USA, 2025. ACM. doi:10.1145/3641554.3701863.
- 12 Meta AI Research et al. The llama 3 herd of models, 2024. Accessed: 2025-05-09. URL: <https://ai.meta.com/research/publications/the-llama-3-herd-of-models/>.
- 13 Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. Llms in software security: A survey of vulnerability detection techniques and insights. 2025. [arXiv:2502.07049](https://arxiv.org/abs/2502.07049).

- 14 Mistral AI Team. Codestral 25.01: A new sota lightweight and fast coding ai model, 2025. Acesso em: 9 mai. 2025. URL: <https://mistral.ai/news/codestral-2501>.
- 15 Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pages 1–7, New Orleans LA USA, 2022. ACM. doi:10.1145/3491101.3519665.
- 16 Andrew Walker, Michael Coffey, Pavel Tisnovsky, and Tomas Cerny. On limitations of modern static analysis tools. 621:577–586, 2020. doi:10.1007/978-981-15-1465-4_57.
- 17 Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. Beyond correctness: Benchmarking multi-dimensional code generation for large language models. 2024. arXiv:2407.11470.
- 18 Kyrie Zhixuan Zhou, Zachary Kilhoffer, Madelyn Rose Sanfilippo, Ted Underwood, Ece Gumusel, Mengyi Wei, Abhinav Choudhry, and Jinjun Xiong. “the teachers are confused as well”: A multiple-stakeholder ethics discussion on large language models in computing education. *arXiv preprint arXiv:2401.12453*, 2024. URL: <https://arxiv.org/abs/2401.12453>.
- 19 Xin Zhou, Duc-Manh Tran, Le Thanh, Ting Zhang, Ivana Irsan, Joshua Sumarlin, Xuan Bach D Le, and David Lo. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. July 2024. doi:10.48550/arXiv.2407.16235.