From Large to Mammoth: A Comparative Evaluation of Large Language Models in Vulnerability Detection

Jie Lin
University of Central Florida
ji132432@ucf.edu

David Mohaisen
University of Central Florida
mohaisen@ucf.edu

***Abstract*—Large Language Models (LLMs) have demonstrated strong potential in tasks such as code understanding and generation. This study evaluates several advanced LLMs—such as `LLaMA-2`, `CodeLLaMA`, `LLaMA-3`, `Mistral`, `Mixtral`, `Gemma`, `CodeGemma`, `Phi-2`, `Phi-3`, and `GPT-4`—for vulnerability detection, primarily in Java, with additional tests in C/C++ to assess generalization. We transition from basic positive sample detection to a more challenging task involving both positive and negative samples and evaluate the LLMs' ability to identify specific vulnerability types. Performance is analyzed using runtime and detection accuracy in zero-shot and few-shot settings with custom and generic metrics. Key insights include the strong performance of models like `Gemma` and `LLaMA-2` in identifying vulnerabilities, though this success varies, with some configurations performing no better than random guessing. Performance also fluctuates significantly across programming languages and learning modes (zero- vs. few-shot). We further investigate the impact of model parameters, quantization methods, context window (CW) sizes, and architectural choices on vulnerability detection. While CW consistently enhances performance, benefits from other parameters, such as quantization, are more limited. Overall, our findings underscore the potential of LLMs in automated vulnerability detection, the complex interplay of model parameters, and the current limitations in varied scenarios and configurations.**

I. Introduction

Vulnerability detection research has witnessed significant progress with the advent of deep learning (DL) and natural language processing (NLP) techniques [1], [2]. These techniques have enabled more efficient identification of vulnerabilities in software systems, enhancing traditional approaches, such as static and dynamic analysis, in terms of scalability, accuracy, and adaptability to evolving coding practices. Moreover, DL techniques have contributed to the field of vulnerability detection, achieving high accuracy in predicting exploitable vulnerabilities [3], [4]. Coupled with NLP techniques that leverage transformer-based models, DL has shown promise in automating vulnerability detection and improving accuracy [5].

The rise of Large Language Models (LLMs) signals promising progress in vulnerability analysis and detection. For instance, models such as SecureFalcon have achieved notable accuracy in classifying vulnerabilities [6], though their focus remains on classification rather than direct detection. Additionally, studies have examined LLM adaptation across various fields [7], yet their capabilities for vulnerability detection are neither well understood nor thoroughly investigated. Recent research showcases LLMs' strengths in code understanding and generation, suggesting they could be especially valuable for vulnerability detection in Java and C/C++ code, where grasping complex syntactic and semantic structures is essential. However, understanding the underlying factors affecting the performance of such techniques is underdeveloped.

Existing research has advanced the field, but limitations persist, notably in the narrow scope of evaluations regarding models and their underlying performance. For example, Thapa *et al.* [5] and Steenhoek *et al.* [8] focus only on C/C++ code, and studies by Purba *et al.* [9] and Khare *et al.* [10] cover a limited set of LLMs. Broader studies, like those by Liu *et al.* [11] and Mathews *et al.* [1], concentrate on specific models (`GPT-3.5` and `GPT-4`), while Ullah *et al.* [12] examine eight LLMs but with limited parameter analysis. Additionally, research has not deeply explored factors like model size, context window, and quantization on LLM vulnerability detection accuracy, underscoring the need for more comprehensive studies to understand these influences.

Using five LLM families, ten architectures, 21 LLMs by varying the architectures' corresponding model size, and 38 by varying configurations, we address this gap by presenting a comprehensive investigation into how various LLM characteristics affect vulnerability detection performance in Java and C/C++ codes. We focus specifically on file-level vulnerability detection, determining whether a given code file contains vulnerabilities. This scope is deliberately chosen to establish a clear baseline for understanding LLMs' capabilities in security analysis. Unlike previous approaches that rely on external tools or knowledge bases, our baseline considers a zero-shot approach to evaluate the models' inherent capabilities, examining how model families, parameter sizes, quantization methods, and context window sizes influence detection accuracy.

With a carefully curated dataset of Java code samples, we conduct extensive experiments with state-of-the-art LLMs, including `LLaMA-2`, `CodeLLaMA`, `LLaMA-3`, `Mistral`, `Mixtral`, `Gemma`, `CodeGemma`, `Phi-2`, `Phi-3`, and `GPT-4`. Our analysis begins with positive samples (known vulnerable code) and extends to include negative samples, providing a comprehensive assessment of the models' discrimination capabilities. To ensure robustness, we further validate our findings on C/C++ source code, exploring the generalizability of our observations across programming languages.

Our investigation goes beyond basic vulnerability detection to explore whether few-shot learning improves accuracy and whether models can identify specific vulnerability types and

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241491
www.ndss-symposium.org

TABLE I: Tested models. ⋆ is the mode's mini-version.

Model	Parameters	Version	Type	Quantization	CW
LLaMA-2	7B	-	Chat	q5_K_M	4096
LLaMA-2	13B	-	Chat	q5_K_M	4096
LLaMA-2	70B	-	Chat	q5_K_M	4096
CodeLLaMA	7B	-	Instruct	q5_K_M	16384
CodeLLaMA	34B	-	Instruct	q5_K_M	16384
CodeLLaMA	70B	-	Instruct	q5_K_M	2048
LLaMA-3	8B	-	Instruct	q5_K_M	8192
LLaMA-3	70B	-	Instruct	q5_K_M	8192
Mistral	7B	v0.2	Instruct	q5_K_M	32768
Mixtral	8x7B	v0.1	Instruct	q5_K_M	32768
Gemma	2B	v1.1	Instruct	q5_K_M	8192
Gemma	2B	v1.1	Instruct	fp16	8192
Gemma	7B	v1.1	Instruct	q5_K_M	8192
Gemma	7B	v1.1	Instruct	fp16	8192
CodeGemma	7B	v1.1	Instruct	q5_K_M	8192
CodeGemma	7B	v1.1	Instruct	fp16	8192
Phi-2	2.7B	v2	Chat	q5_K_M	2048
Phi-2	2.7B	v2	Chat	fp16	2048
Phi-3	3.8B⋆	-	Instruct	q5_K_M	4096
Phi-3	3.8B⋆	-	Instruct	fp16	4096
GPT-4	-	-	Chat	-	-

associate them with CVE IDs or descriptions, all without external knowledge bases. This comprehensive evaluation systematically analyzes how various factors influence vulnerability detection performance, addressing a critical research gap.

Except GPT-4, the majority of the used LLMs in this study are open-source. A key reason for focusing on open-source models for vulnerability detection is the security risk associated with third-party services, since relying on closed-source models, such as GPT-4 accessed via OpenAI's APIs, poses risks like data privacy breaches and potential exposure of sensitive information to external servers. Open-source models enable local processing, ensuring data remains secure within the organization. Additionally, open-source models offer transparency, allowing to audit and modify code as needed, enhancing trust in deployment. Fine-tuning local models for vulnerability detection is a promising area, offering specialized and effective solutions compared to closed-source models. The emphasis on open-source models is a commitment to secure and transparent solutions given their competitive performance.

Research Questions. We structure our study as an investigation around several critical aspects of using LLMs for vulnerability detection in Java and C/C++ code samples. Based on a diverse set of models (Table I), we formulated the following questions to guide our experiments and analysis:

RQ1. *Can LLMs be used for vulnerability detection and type identification across language?* By evaluating various models on Java and C/C++ codes, we attempt to determine the overall efficacy of LLMs across various metrics for both vulnerability detection and vulnerability type identification.

RQ2. *Does the context window (CW) affect the overall performance of LLMs?* CW, the maximum token capacity a model can handle at once, is critical for understanding large programs. We hypothesize a smaller CW may reduce the accuracy. By comparing models with CW sizes of 2^{11}–2^{15} tokens, we aim to quantify CW effects on model performance.

RQ3. *Does quantization matter for vulnerability detection* Quantization reduces model weight precision, decreasing memory usage and boosting inference speed. By comparing various models under different quantizations, we identify the trade-offs between efficiency and accuracy in our task.

RQ4. *Do advanced architectures affect detection?* We examine the impact of LLMs architectural enhancements on detection by assessing whether these improvements indeed translate to better vulnerability detection compared to predecessors.

RQ5. *Does few-shot learning improve the vulnerability detection accuracy?* In realistic and constrained settings, we examine whether exposing the models to some examples pertaining to the detected vulnerability type affect the performance.

By addressing these questions, we aim to comprehensively analyze the capabilities and limitations of various LLMs in vulnerability detection. This study will highlight the strengths and weaknesses of different models and offer insights into the practical implications of model selection, CW size, quantization, and architectural advancements for a real-world application.

Key findings from our study reveal that while context window size consistently improves performance, the impact of other parameters varies significantly across model families. Some models, particularly in the Gemma and LLaMA-2 families, demonstrate strong capabilities in identifying vulnerabilities, though this performance is not universal. These insights have important implications for both the theoretical understanding of LLMs in security analysis and their practical application in development tools.

Contributions. This paper presents the following contributions: (1) An empirical analysis of the impact of model architecture, parameters, quantization, and context window size on vulnerability detection performance in both Java and C/C++ code; (2) A systematic evaluation framework to assess the inherent vulnerability detection capabilities of LLMs without external aids; (3) Insights into factors influencing the success of LLMs in vulnerability detection, including cross-language performance and prompting strategies; and (4) Evidence-based recommendations for optimizing LLM configurations for vulnerability detection tasks.

Organization. § II provides background, followed by related work in § III, methodology in § IV, results in § V, discussion in § VI, and conclusion in § VII. The limitations are deferred to appendix J due to the lack of space.

II. Background

LLMs have witnessed significant advancements in recent years, driven by innovations in neural networks, Transformer-based architectures, and training methods [13]. In particular, the self-attention mechanism's efficiency and ability to handle long-range dependencies have made the Transformer a cornerstone for subsequent LLM developments. In the following, we provide the background of the LLMs used in this study.

The LLaMA Family. Meta's contributions to LLMs began with the LLaMA models family. Touvron *et al.* [14] introduced LLaMA, with 7B to 65B parameters, trained on publicly available datasets with compute-optimal approaches. Key improvements included pre-normalization, SwiGLU activation, and rotary positional embeddings. LLaMA-2, presented by Touvron *et al.* [15], expanded the pretraining corpus of LLaMA by 40%, doubled the context length, and incorporated grouped-query attention for scalability. Roziére *et al.* [16] extended this line with CodeLLaMA, optimized for code generation and handling sequences up to 100,000 tokens. The LLaMA-3 models introduced enhanced tokenizers for long-context tasks [17].

The Gemma Family. Google developed the Gemma and CodeGemma families of transformer-based models. Mesnard *et al.* [18] presented Gemma using a decoder-only architecture with Multi-Query Attention, RoPE Embeddings, GeGLU Activations, and RMSNorm. These models, trained on diverse datasets, including web documents and code, showed strong language understanding and reasoning performance. Cuenca *et al.* [19] built on this foundation with CodeGemma, specifically tailored for code-related tasks and maintaining high performance on sequences up to 8192 tokens.

The Mistral Family. Jiang *et al.* [20] introduced Mistral 7B, leveraging grouped-query and sliding window attention for efficient inference, outperforming larger models like LLaMA-2 13B. The Mistral architecture was further extended with Mixtral 8x7B, employing a Sparse Mixture of Experts (SMoE) approach to balance the performance and efficiency by activating only a subset of its parameters during inference [21]. These innovations significantly reduced computational costs while maintaining high performance.

The Phi Family. Microsoft's contributions are exemplified by the Phi family of models. Gunasekar *et al.* [22] introduced Phi-1, a Transformer-based model designed for code generation, with 24 layers and 32 heads, trained on high-quality datasets. Li *et al.* [23] expanded this with Phi-1.5, enhancing training data with synthetic datasets for improved performance in common sense reasoning and general knowledge tasks. Abdin *et al.* [24] scaled the model up to 2.7 billion parameters in Phi-2, emphasizing data quality and innovative scaling techniques. The Phi-3-mini model presented by Abdin *et al.* [25] further demonstrated the potential of smaller, efficiently trained models to rival much larger counterparts, showcasing strategic data curation and training techniques.

These advances in architecture and training underscore the rapid evolution and increasing sophistication of LLMs, setting the stage for future breakthroughs in the field. The continued innovation in architectural designs and strategic data curation has significantly enhanced the performance and applicability of LLMs across various domains.

III. Related Work

Code Generation. Tipirneni *et al.* [26] introduced StructCoder, a Transformer-based model for code generation that integrates Abstract Syntax Tree (AST) and Data Flow Graph (DFG) into its encoder and decoder, enhancing its understanding of syntactic and semantic relationships in code. StructCoder also utilizes a structure-aware self-attention and is pretrained with a structure-based denoising autoencoding task. It excels in code translation and text-to-code generation tasks, surpassing baselines like CodeT5 in BLEU, exact match, and CodeBLEU metrics on the CodeXGLUE benchmark.

Vulnerability Detection. Thapa *et al.* [5] investigated transformer-based models for detecting vulnerabilities in C/C++ code. They developed a framework using "code gadgets," which are groups of semantically related code slices with data or control dependencies. This framework involved normalizing code, extracting function and variable definitions, and identifying relevant code lines based on data dependencies. Each code gadget was treated as a unit for binary and multi-class classification to detect vulnerabilities. Results indicated that transformer models, especially GPT-2 Large and GPT-2 XL, outperformed the traditional RNN models.

Purba *et al.* [9] evaluated LLMs for multi-label vulnerabilities classification in C/C++ code, using the code gadgets (focused on memory management and buffer overflow) and CVEfixes (including SQL injection and buffer overflow) datasets. They compared four LLMs—GPT-3.5-Turbo, GPT-3.5-Turbo-0613, Codegen-2B-multi, and Davinci—through fine-tuning and zero-shot approaches. Although LLMs showed high false-positive rates compared to traditional tools, fine-tuned models demonstrated an ability to recognize common vulnerability patterns.

Khare *et al.* [10] evaluated the capabilities of pre-trained LLMs, such as GPT-4, CodeLLaMA-7B, and CodeLLaMA-13B, for vulnerability detection across five benchmarks in Java and C/C++. In their study, vulnerability detection was defined as identifying code patterns associated with specific Common Weakness Enumeration (CWE) classes within isolated code snippets, rather than complete source files or projects. They found that CWE-specific prompts and data flow analysis-based prompts notably enhanced LLM performance, with GPT-4 achieving high F1 scores on synthetic datasets like OWASP and Juliet.

Mathews *et al.* [1] applied LLMs to Android vulnerability detection using the Ghera dataset, refining prompts to address high false-positive rates. Using a retrieval-augmented generation approach, which allowed models to access key files (e.g., AndroidManifest.xml, MainActivity.java), they improved vulnerability identification despite noise from irrelevant files.

Steenhoek *et al.* [8] provided an evaluation of eleven LLMs, including GPT-4 and CodeLLaMA, for vulnerability detection in isolated code functions, rather than entire source code. They define vulnerability detection as the model's capacity to identify security flaws within functions by reasoning through code structure, boundary checks, and logical flow. The authors introduced three prompting techniques: in-context prompting, chain-of-thought prompting from CVE descriptions, and from static analysis. However, the LLMs struggled, achieving Balanced Accuracy scores between 0.5 and 0.63 and failing to differentiate between buggy and fixed function versions 76% of the time. Human participants significantly outperformed LLMs in bug localization, where LLMs correctly located only 6 of 27 bugs in DbgBench.

Liu *et al.* [11] evaluated ChatGPT's performance on six vulnerability management tasks, such as bug report summarization and vulnerability repair, using a dataset of 70,346 samples and comparing GPT-3.5 and GPT-4 to 11 state-of-the-art approaches. They found ChatGPT effective in tasks like bug report summarization but noted challenges in more complex tasks, highlighting the critical role of prompt design and model selection for vulnerability management.

Ullah *et al.* [12] introduced SecLLMHolmes, a framework for evaluating LLMs on vulnerability identification and reasoning across 228 code scenarios, including CVEs and augmented examples. Testing eight LLMs with varied prompting techniques, they found limitations in LLMs, including non-deterministic outputs and poor performance on real-world scenarios, concluding that current LLMs are unreliable for automated vulnerability detection.

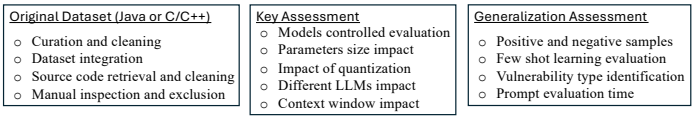

Fig. 1: Overview of our evaluation.

This Work. We present a comprehensive evaluation of LLMs' capabilities for real-world vulnerability detection scenarios, addressing limitations in existing work through several key aspects: (1) evaluating complete source code files rather than isolated snippets, better reflecting actual development environments, (2) employing zero-shot learning as our primary strategy to assess out-of-box performance without relying on fine-tuning or external information (e.g., CVE/CWE descriptions), simulating realistic scenarios where vulnerability detectors lack prior knowledge, (3) testing with both positive and negative samples to simulate real deployment conditions, and (4) examining cross-language generalization between Java and C/C++. By evaluating multiple open-source LLMs, including `LLaMA-2`, `CodeLLaMA`, `LLaMA-3`, `Mistral`, `Mixtral`, `Gemma`, `CodeGemma`, `Phi-2`, and `Phi-3`, we provide insights into their practical applicability for security analysis. Unlike previous studies that focused on code snippets or relied on external knowledge bases, our work offers a holistic assessment of LLMs' inherent capabilities in vulnerability detection under realistic constraints, contributing to a deeper understanding of their potential in enhancing software security.

IV. Methodology

As Our evaluation, per Figure 1, begins with the automated retrieval of relevant data, followed by a cleaning and preprocessing for both Java and C/C++ datasets. Next, the dataset undergoes vulnerability detection testing, employing a systematic approach to address key evaluation questions regarding LLM performance in isolation and the influence of factors like model size, quantization, context window, and architecture. To broaden the scope of our evaluation, we also investigate vulnerability detection using negative samples, prompt settings such as few-shot learning, vulnerability type identification, and prompt evaluation time.

A. Datasets

Java Dataset. Our primary Java dataset is sourced from the `Vul4J` dataset developed by Bui *et al.* [27]. The `Vul4J` dataset was created by analyzing 1,803 fix commits addressing 912 Java vulnerabilities, resulting in 79 reproducible vulnerabilities from 51 open-source projects across 25 CWE types, with 28 vulnerabilities corresponding to OWASP's Top 10 risks. The dataset includes PoV tests, patches, and build information for evaluating APR tools.

The dataset curation process involved automated retrieval of CVE and CWE descriptions using the OpenCVE API, followed by data cleaning to ensure consistency by removing whitespaces and newlines. We integrated these descriptions into the dataset and retrieved source code using the GitHub API, tracking changes between pre-patched and post-patched versions. Source codes no longer available on GitHub were excluded. After removing comments to reduce context token length, we manually inspected and filtered the data, excluding non-Java files such as XML to ensure accuracy and relevance. Details are deferred to appendix A for the lack of space.

Final Java Dataset. The final dataset comprises 280 Java files, consisting of 140 pairs of vulnerable and non-vulnerable versions covering 74 different vulnerabilities.

C/C++ Dataset. To evaluate the generalizability of our findings, we extended our analysis to assess LLMs' ability to detect vulnerabilities in C/C++ code. Utilizing the same data curation pipeline described in section IV-A, we processed data from the `Big-Vul` dataset [28]. The `Big-Vul` dataset initially includes 348 projects associated with 4,432 unique code commits, addressing 3,754 vulnerabilities across 91 CWE types. After processing, the dataset comprised 8,314 code files, representing 4,157 pairs of pre-patched (vulnerable) and post-patched (non-vulnerable) code.

Final C/C++ Dataset. To ensure consistency and facilitate a robust validation process, we randomly sampled 200 code files (100 vulnerable and 100 non-vulnerable) from the processed `Big-Vul` dataset. This sample size mirrors our `Vul4J` dataset, allowing for direct comparison and reinforcing the reliability of our generalizability assessments.

B. Model Selection and Configurations

The selection of models for our work was based on their performance in the field of LLMs and their capabilities in various NLP tasks. We included models from several prominent families, namely `LLaMA-2`, `CodeLLaMA`, `LLaMA-3`, `Mistral`, `Mixtral`, `Gemma`, `CodeGemma`, `Phi-2`, `Phi-3`, and `GPT-4`. The primary reasons for these selections stem from their architectural innovations, performance benchmarks, and relevance in current LLMs research.

We selected a diverse range of models with varying architectures, parameter sizes, and training objectives, focusing solely on fine-tuned models optimized for instruction-following and dialogue. These "chat" or "instruct" models are designed for conversational understanding and response, aligning our evaluation with real-world applications where LLMs are used interactively, such as vulnerability detection via dialogue. This approach allows us to assess the effectiveness of fine-tuned models under conditions relevant to security analysis. Model details are provided in Table I.

Context Window (CW). CW refers to the maximum number of tokens a model can process simultaneously. For example, most `CodeLLaMA` models have a CW of 16,384 tokens, except for the 70B model, which has a notably smaller CW of 2048 tokens, as verified in our tests. This discrepancy may reflect specific architectural choices. The CW sizes for other models align with their descriptions in the respective research papers.

Reference Model. `GPT-4`, developed by OpenAI, was chosen as a benchmark to evaluate open-source models against a top closed-source model. While the exact parameter count of `GPT-4` is not officially disclosed[1], it is known for its strong performance. Released in different versions with varying CWs, such as `GPT-4-0613` with an 8192-token CW and `GPT-4-turbo-0125-preview` with up to 128,000 tokens,

[1]Although details are largely undisclosed, some sources suggest `GPT-4` consists of 8 models totaling 1.76 trillion parameters.

its quantization specifics remain undisclosed but likely use full precision (fp32). Given `GPT-4`'s evolving specs, details like CW and quantization may change; thus, they were excluded from Table I to avoid inaccuracies and reflect potential updates.

Quantization. Quantization reduces the precision of a model's weights to lower memory usage and boost inference speed with minimal performance impact. Our experiments mainly employed the `Q5_K_M` method, which strikes a balance between performance and efficiency by applying `Q6_K` to half of the attention and feed-forward tensors while using `Q5_K` for the rest, retaining most of the model's performance. Other methods include `Q5_K_S`, which applies `Q5_K` across all tensors for reduced resource usage at a slight performance cost, and `Q6_K`, which utilizes `Q8_K` for all tensors, providing high accuracy but demanding substantial resources. Additionally, `Q8_0` closely resembles `fp16` (16-bit floating point) in performance but requires more resources, making it less practical.

Justification. The choice of `Q5_K_M` is driven by its ability to preserve most models' performance while optimizing resources making it ideal for large-scale evaluations. Models with `fp16` precision were included as a baseline, helping to evaluate the effectiveness of the quantization applied to some models.

C. Experimental Pipeline

Our pipeline is designed to evaluate the effectiveness of LLMs in performing vulnerability detection on Java code. We highlight our pipeline in more detail in the following.

System Prompt. In our pipeline, the system prompt guides the LLMs in executing intended task and is crafted to simulate an expert Java programmer, instructing the model to carefully analyze the provided Java code and determine its vulnerability status. Our approach employs a *zero-shot* prompting strategy, deliberately chosen to evaluate the inherent capabilities of LLMs in vulnerability detection without task-specific fine-tuning or examples. This approach provides crucial insights into the models' baseline performance and generalization abilities in a challenging security context. The prompt leverages the generative capabilities of LLMs by not only requesting a binary yes/no response but also requiring them to provide reasoning for their decision. The used prompt is as follows:

> *You are an expert Java programmer who can carefully analyze the provided Java code. The goal is to judge if the provided code is vulnerable or not. Your answer should be concise, with a yes or no to represent the code's type. If it is vulnerable, then yes; otherwise, no. Also, please explain concisely why you made the decision.*

This prompt sets a clear context, describing the specific task and constraining the response format to ensure consistency. The model is expected to respond with a simple "yes" or "no" followed by a concise explanation of the decision, mimicking the behavior of a human expert (examples are in appendix C).

Pre-patched Code. The user prompt also provides the pre-patched Java code for each tested file, formatted to ensure clarity. The user prompt template is structured as follows to include the following comma-separated lines: `template = f""", ```java, {pre_code}```, """`. This straightforward format ensures that the Java code is presented clearly to the LLMs for analysis, for effective task execution.

Model Configuration. Our pipeline implements the chosen model configurations to ensure reproducibility, consistency, and fair comparison across LLMs. For open-source models, we standardized several parameters to maintain rigor.

We set the temperature to 0.5 for all open-source models, balancing between deterministic outputs and maintaining necessary flexibility in response generation. This setting is important for vulnerability detection tasks where we need focused, precise answers while allowing the model to explain its reasoning. The choice of 0.5 aligns with the standard 0-1 temperature scale used by open-source LLMs, providing better comparability across models than OpenAI's 0-2 range.

To enhance reproducibility, we fixed the random seed to 42 for all open-source models. This ensures that given the same pre-trained weights and input, the models generate consistent outputs across different runs. The fixed seed is fundamental for scientific validation and benchmarking purposes, allowing other researchers to replicate our results reliably.

For output generation, we implemented a 2048-token limit across all models, serving multiple purposes: to prevent excessive generation, avoid potential memory issues, and ensures efficient post-processing and manual analysis of results. The limit was empirically chosen to provide sufficient space for detailed explanations while addressing large-scale evaluation.

For GPT-4, we maintained its default configuration without modifications. This decision acknowledges GPT-4's non-deterministic nature and allows it to serve as a baseline for comparing open-source model performance. While this means GPT-4's responses may show slight variations across runs, it represents real-world usage scenarios more accurately.

CW Configurations. We designed two distinct configurations by varying the CW size: the restricted and the maximum CW. ① *Restricted CW.* We restrict all LLMs to an input CW of 2048 tokens to ensure fair comparison across all models, as some models like `Phi-2` have a maximum input token length of 2048. By standardizing to this length, we create a level playing field for evaluation while maintaining the 2048-token output limit and temperature of 0.5 for consistent responses. ② *Maximal CW.* We remove the input token restriction, allowing each LLM to operate at its maximum supported CW capacity to evaluate the true capabilities of open-source LLMs in their out-of-box settings while maintaining the 2048-token output limit. For instance, `CodeLLaMA-7B` can process inputs up to 16,384 tokens in this setting, potentially allowing for more comprehensive code analysis.

Preventing Leakage. A key aspect of our pipeline is preventing data leakage between evaluations. For each new user prompt, we fully offload and reload the LLM to ensure that evaluations remain independent and free from influence by prior interactions. This model reload strategy preserves the zero-shot nature of the assessment, ensuring each evaluation is based solely on the current input. Although computationally intensive, this process guarantees the integrity of our results. This comprehensive configuration of our experimental pipeline ensures reproducible, fair, and unbiased evaluation of LLM performance in vulnerability detection. The careful balance of parameters and strict data leakage prevention measures enable us to systematically address the research questions outlined in Table I while maintaining scientific rigor.

D. Evaluation Metrics

The evaluation metrics are the accuracy and explicitness of the model's response in determining whether the provided Java code files are vulnerable. Given that all Java code files in our curated dataset are indeed vulnerable, the evaluation focuses on the explicitness and correctness of the LLMs' responses.

Custom Metrics. Manual evaluation is conducted to ensure accuracy due to the varied freestyle nature of LLM responses, making automatic assessments difficult. Responses are categorized as (1) Correct if marked as vulnerable (weighted 1), (2) Incorrect if marked as not vulnerable (weighted 0), and (3) Irrelevant if they do not address vulnerability (weighted -1). The accurate response as a percentage (AP) represents the proportion of correct responses marked as vulnerable, while the explicit response as a percentage (EP) reflects responses that clearly state the vulnerability status. Vulnerability-level aggregation groups code files by vulnerability ID (VID), with correct VID assessment requiring at least one file marked as vulnerable and explicit VID response requiring at least one explicit statement of vulnerability status. As such, vulnerability-level accurate responses (VAP) and vulnerability-level explicit responses (VEP) assess the proportion of VIDs correctly and explicitly evaluated, respectively. Details of these metrics are deferred to appendix B for the lack of space.

Other Standard Metrics. We evaluated our models using three standard metrics: precision (P), recall (R), and F1 score (F1), all expressed as percentages. These metrics are defined as follows. (1) **Precision (P)**: The ratio of true positive predictions to the total number of positive predictions made by the model reflecting the model's ability to correctly identify actual vulnerabilities without generating false alarms. (2) **Recall (R)**: The ratio of true positive predictions to the total number of actual positive cases, indicating the model's effectiveness in detecting all existing vulnerabilities. (3) **F1 Score (F1)**: The harmonic mean of P and R, providing a single metric that balances both false positives and false negatives.

V. Evaluation and Results

We present the detailed results of our evaluation based on the metrics defined above. The results highlight the performance of different LLMs for best performant in isolation, best performant variants, both with Java codes, extended results for C/C++, testing with positive and negative samples, few-shot learning, vulnerability identification, and time overhead.

A. Controlled Model Comparison

We focus on a controlled model comparison by fixing the parameters for the best-performing models and highlighting their performance. More details are delegated to Appendix F.

AP/VAP. Table II shows the overall performance.

Best Performance. `Gemma` achieves the highest performance in both metrics, with an AP of 78.57% and a VAP of 93.24%, indicating its superior ability to identify vulnerabilities and correctly assess specific VIDs accurately. `LLaMA-2` also exhibits strong performance with an AP of 70.00% and a VAP of 82.43%, significantly (and surprisingly) higher than its successor, `LLaMA-3`, which records an AP of 38.57% and a VAP of 52.70%. This result suggests that `LLaMA-3`, despite being a newer version that performs better on a range of tasks, does not necessarily outperform its predecessor, `LLaMA-2`, in these specific tasks. This finding underscores the need to understand the different model variants for the specified task.

Contradictory Results. `CodeLLaMA`, a variant of `LLaMA-2` specifically trained for code understanding and generation, shows an AP of 39.29% and a VAP of 58.11%, which are significantly and surprisingly lower than `LLaMA-2` but higher slightly than `LLaMA-3`. Similarly, `CodeGemma`, another model tailored for code-related tasks, achieves an AP of 48.57% and a VAP of 59.46%, slightly outperforming `CodeLLaMA` but still falling short compared to its base model, `Gemma`, which is supposed to improve for this specific task.

Moreover, `Mixtral`, claimed to be an advanced version of `Mistral`, performs poorly with an AP of 12.86% and a VAP of 21.62%, compared to `Mistral`'s AP of 20.71% and VAP of 39.19%. This indicates that `Mixtral` enhancements did not translate to better performance in these specific tasks. Similarly, we found that `Phi-2` outperforms its successor, `Phi-3`, with APs of 40.71% vs. 30.00%, and VAPs of 55.41% vs. 45.95%, respectively. This demonstrates that `Phi-3`, although a more recent version, does not improve on `Phi-2` in accurately identifying vulnerabilities. Finally, `GPT-4`, known for its performance in various tasks, achieves a disappointing AP of 37.86% and a VAP of 51.35%, ranking 7th overall.

Open vs. Closed Source. When comparing open-source models to `GPT-4`, it is evident that certain open-source models, such as `Gemma` and `LLaMA-2`, outperform `GPT-4` in both AP and VAP metrics. These results show that performant open-source models can rival or even exceed the capabilities of proprietary models in specific vulnerability detection tasks. The availability of these open-source models provides significant value to the research community by offering accessible alternatives that do not compromise performance.

Takeaway. Our results illustrate that specialized models have strengths but do not consistently outperform their general purpose or earlier versions across all metrics. Moreover, the comprehensive evaluation of both AP and VAP provides valuable insights into each model's capabilities in detection.

EP/VEP Performance. The right-hand side of Table II shows the overall performance of the different models.

Best in-class Models. As observed, `GPT-4` and `Mixtral` achieve the highest scores of 100% in both EP and VEP, indicating that these models provide clear and explicit assessments of vulnerability status in all instances. `CodeLLaMA` and `LLaMA-3` also demonstrate strong performance with high EP and VEP scores, showcasing their ability to make explicit vulnerability assessments effectively.

Consistent Results. `CodeLLaMA` outperforms `LLaMA-2`, with an EP/VEP of 71.43%/83.78%. This underscores the effectiveness of domain-specific training in improving the model's ability to provide explicit responses. Similarly, `CodeGemma` shows marked improvements over `Gemma` in both metrics.

Similarly, `LLaMA-3` performs better than `LLaMA-2`, indicating improved model architecture and training methods. `Mixtral` surpasses `Mistral`, and `Phi-3` outperforms `Phi-2`, following the trend that newer versions and specifically trained models achieve better results in making explicit

TABLE II: File- and VID-level performance of the different models.

Model	Param	Quant	CW	AP	VAP	EP	VEP
LLaMA-2	7B	q5_K_M	2048	45.71	62.16	45.71	62.16
LLaMA-2	7B	q5_K_M	4096	68.57	82.43	70.71	82.43
LLaMA-2	13B	q5_K_M	2048	17.86	28.38	45.71	62.16
LLaMA-2	13B	q5_K_M	4096	45.71	63.51	71.43	83.78
LLaMA-2	70B	q5_K_M	2048	45.00	62.16	45.71	62.16
LLaMA-2	70B	q5_K_M	4096	70.00	82.43	70.71	82.43
CodeLLaMA	7B	q5_K_M	2048	2.14	4.05	45.0	60.81
CodeLLaMA	7B	q5_K_M	16384	22.86	39.19	92.14	98.65
CodeLLaMA	34B	q5_K_M	2048	9.29	16.22	44.29	59.46
CodeLLaMA	34B	q5_K_M	16384	13.57	25.68	82.14	93.24
CodeLLaMA	70B	q5_K_M	2048	39.29	58.11	45.0	62.16
LLaMA-3	8B	q5_K_M	2048	12.86	21.62	47.86	62.16
LLaMA-3	8B	q5_K_M	8192	38.57	52.70	90.71	97.3
LLaMA-3	70B	q5_K_M	2048	2.86	4.05	47.86	62.16
LLaMA-3	70B	q5_K_M	8192	2.86	4.05	92.86	98.65
Mistral	7B	q5_K_M	2048	9.29	17.57	45.0	62.16
Mistral	7B	q5_K_M	32768	20.71	39.19	97.86	97.3
Mixtral	8x7B	q5_K_M	2048	7.14	10.81	45.0	60.81
Mixtral	8x7B	q5_K_M	32768	12.86	21.62	100.0	100.0
Gemma	2B	q5_K_M	2048	52.14	64.86	52.14	64.86
Gemma	2B	q5_K_M	8192	68.57	85.14	70.0	85.14
Gemma	2B	fp16	2048	53.57	66.22	53.57	66.22
Gemma	2B	fp16	8192	78.57	93.24	78.57	93.24
Gemma	7B	q5_K_M	2048	32.86	51.35	53.57	66.22
Gemma	7B	q5_K_M	8192	69.29	87.84	94.29	98.65
Gemma	7B	fp16	2048	37.86	54.05	53.57	66.22
Gemma	7B	fp16	8192	77.86	90.54	94.29	98.65
CodeGemma	7B	q5_K_M	2048	13.57	22.97	54.29	66.22
CodeGemma	7B	q5_K_M	8192	48.57	59.46	95.0	98.65
CodeGemma	7B	fp16	2048	10.00	17.57	53.57	66.22
CodeGemma	7B	fp16	8192	42.86	59.46	89.29	98.65
Phi-2	2.7B	q5_K_M	2048	40.71	55.41	42.14	58.11
Phi-2	2.7B	fp16	2048	40.00	55.41	40.71	56.76
Phi-3	3.8B[1]	q5_K_M	2048	22.14	37.84	45.71	62.16
Phi-3	3.8B[1]	q5_K_M	4096	30.00	45.95	52.14	66.22
Phi-3	3.8B[1]	fp16	2048	16.43	28.38	45.71	62.16
Phi-3	3.8B[1]	fp16	4096	23.57	36.49	52.14	66.22
GPT-4	-	-	-	37.86	51.35	100.0	100.0

vulnerability assessments. Phi-2, with the lowest scores in both EP and VEP, demonstrates the least effective performance, whereas Phi-3 shows considerable improvement.

Takeaway. While GPT-4 sets a high benchmark with perfect performance, open-source models like CodeLLaMA, LLaMA-3, and Mixtral exhibit strong performance with explicit and accurate responses. These results highlight the importance of domain-specific training in enhancing the effectiveness of LLMs for vulnerability detection.

B. Performance Analysis of Model Variants

1) AP/VAP Performance: Table II presents the AP and VAP across different models by altering the model size (i.e., the number of parameters in the model), CW size, and quantization method. In both tables, the distinct colors highlight the best-performing models within each family (i.e., best-in-class).

Both tables show a comprehensive analysis of the impact of different model parameters on the performance of LLMs in identifying vulnerabilities in Java codes. The analysis of these tables reveals similar trends, with the number of parameters, quantization methods, CW sizes, and model advancements showing varied impacts on performance across models.

Model Size and Performance. Our analysis reveals that the number of parameters does not consistently correlate with better performance across different model families. LLaMA-2 models show inconsistent improvements with parameter increases, while CodeLLaMA models generally benefit from more parameters, albeit inconsistently. Surprisingly, the Gemma and CodeGemma models with fewer parameters often outperform larger models, highlighting the importance of other parameters in determining performance.

Quantization and Model Performance. Quantization methods, such as q5_K_M and fp16, play a significant role in model performance. In the Gemma family, fp16 quantization generally enhances performance compared to q5_K_M. However, this trend is not universal, as demonstrated by the Phi family, where the effectiveness of quantization varies significantly between configurations.

CW and Performance. Increasing the CW size generally enhances model performance, though results vary across different model families. The LLaMA-2, CodeLLaMA, Gemma, and Mistral model families consistently benefit from larger CWs. However, inconsistencies are observed in the LLaMA-3 family, where the 70B model does not improve with a larger CW. These findings indicate that while larger CWs are beneficial, their effectiveness is influenced by model-specific factors like architecture and quantization methods, necessitating further investigation to optimize performance.

Advanced Variants and Performance. Advancements in model architecture and specialized training datasets result in varied performance improvements. CodeLLaMA models, specialized for code understanding, show mixed results, often underperforming compared to LLaMA-2 models. LLaMA-3 models exhibit improved performance with increased CWs in some cases but do not consistently outperform LLaMA-2. Mixtral models, intended as improved versions of Mistral, display inconsistent performance gains. Similarly, CodeGemma models, fine-tuned for code tasks, do not always surpass the original Gemma models. Phi-3 models do not improve over Phi-2 models, indicating that the architectural advancements and improved training datasets in Phi-3 do not result in better performance. While some advancements offer benefits, they do not universally guarantee better performance across all configurations.

Takeaway. File-level and VID-level analyses highlight the complex interplay between model parameters, quantization, CW, and architectures in determining the performance of LLMs in vulnerability detection. While specific trends emerge, such as the general benefits of larger CWs and the effectiveness of fp16 quantization in some cases, the inconsistencies observed across models highlights the need for further research to optimize these models for specific tasks and domains.

2) EP/VEP Performance: Table II presents the performance of the different models and their variants in terms of EP and VEP. The data in the table allows for a comprehensive analysis of the impact of different model parameters on the file-level and VID-level performance evaluation of LLMs identifying vulnerabilities. Overall, the analysis of these tables reveals similar trends as those in section V-B1, and the detailed analysis is delegated to appendix G for the lack of space.

C. Positive vs. Negative Samples

We extend our analysis by incorporating both vulnerable (positive) and non-vulnerable (negative) Java code samples into our testing pipeline. This setting assesses the LLMs' ability to accurately distinguish between vulnerable and non-vulnerable code, providing a more realistic evaluation of their performance in practical detection scenarios.

TABLE III: Performance comparison of various LLM models for Java and C/C++ vulnerability detection with positive-negative samples using precision, recall, and F1 score. The total samples for Java were 280 and for C/C++ were 200 samples. [1] mini-4k-instruct.

				Java			C/C++		
MF	**Param**	**Quant**	**CW**	**P**	**R**	**F1**	**P**	**R**	**F1**
LLaMA-2	7B	q5_K_M	4096	41.38	68.57	51.61	20.16	25.00	22.32
LLaMA-2	13B	q5_K_M	4096	37.43	45.71	41.16	8.08	8.00	8.04
LLaMA-2	70B	q5_K_M	4096	41.18	70.00	51.85	20.80	26.00	23.11
CodeLLaMA	7B	q5_K_M	16384	51.61	22.86	31.68	28.57	32.00	30.19
CodeLLaMA	34B	q5_K_M	16384	24.68	13.57	17.51	8.05	7.00	7.49
CodeLLaMA	70B	q5_K_M	2048	28.95	39.29	33.33	9.09	10.00	9.52
LLaMA-3	8B	q5_K_M	8192	42.86	38.57	40.60	20.56	22.00	21.26
LLaMA-3	70B	q5_K_M	8192	23.53	2.86	5.10	0.00	0.00	0.00
Mistral	7B	q5_K_M	32768	50.88	20.71	29.44	20.51	8.00	11.51
Mixtral	8x7B	q5_K_M	32768	50.00	12.86	20.45	13.64	3.00	4.92
Gemma	2B	q5_K_M	8192	40.68	68.57	51.06	20.63	26.00	23.01
Gemma	2B	fp16	8192	44.35	78.57	56.70	21.43	27.00	23.89
Gemma	7B	q5_K_M	8192	44.50	69.29	54.19	29.29	41.00	34.17
Gemma	7B	fp16	8192	46.19	77.86	57.98	29.58	42.00	34.71
CodeGemma	7B	q5_K_M	8192	65.38	48.57	55.74	23.71	23.00	23.35
CodeGemma	7B	fp16	8192	54.05	42.86	47.81	20.79	21.00	20.90
Phi-2	2.7B	q5_K_M	2048	29.08	40.71	33.93	9.91	11.00	10.43
Phi-2	2.7B	fp16	2048	28.72	40.00	33.43	9.91	11.00	10.43
Phi-3	3.8B[1]	q5_K_M	4096	26.58	30.00	28.19	6.93	7.00	6.97
Phi-3	3.8B[1]	fp16	4096	23.40	23.57	23.49	4.12	4.00	4.06

Experimental Setup. For this evaluation, we used the complete Vul4J benchmark dataset, comprising 280 Java source files with paired vulnerable (pre-patched) and non-vulnerable (post-patched) versions (examples are in appendix D). This setup allowed us to evaluate the LLMs' ability to detect vulnerabilities. Each code sample was analyzed individually under a zero-shot learning setting, providing insights into the models' generalization abilities. All models were tested with their largest available CW sizes, without adapters or extensions, as prior work suggested larger CWs enhance performance. Models limited to a 2048-token CW were excluded to ensure sufficient capacity for processing full code samples.

System Prompts. We maintained the same system prompt used in our main experiments shown in section IV-C.

Irrelevant Responses. Irrelevant responses—those that fail to address the task—are classified as incorrect predictions: they are treated as false positives when the ground truth is non-vulnerable and as false negatives when it is vulnerable. This ensures that our metrics accurately reflect the models' capability to deliver relevant and correct vulnerability assessments.

Analysis Results. Table III presents the performance metrics of various LLMs on the Java vulnerability detection task. Analyzing the results, we observe the following. (1) Precision: The highest precision is achieved by the CodeGemma 7B model with q5_K_M quantization, attaining 65.38%. This indicates that when this model predicts a vulnerability, it is correct 65.38% of the time, suggesting a lower rate of false positives. Conversely, models like Phi-3 3.8B fp16 have the lowest precision at 23.40%, indicating more false positives. (2) Recall: The highest recall is observed in the Gemma 2B model with fp16 quantization, achieving 78.57%, closely followed by the Gemma 7B fp16 with 77.86%. This demonstrates these models' effectiveness in identifying a majority of the actual vulnerabilities, minimizing false negatives. In contrast, the LLaMA-3 70B model has the lowest with 2.86%. (3) F1 Score: The highest F1 score is achieved by the Gemma 7B model with fp16 quantization, at 57.98%, indicating the best balance between precision and recall among the models tested. The LLaMA-3 70B q5_K_M have a score of 5.1%. More details are in appendix H for the lack of space.

Impact of Parameter Size. Examining the effect of parameter size on performance: (1) LLaMA-2 *Family.* Increasing the parameter size from 7B to 13B results in a decrease in Precision (from 41.38% to 37.43%) and F1 Score (from 51.61% to 41.16%), while Recall decreases from 68.57% to 45.71%. However, further increasing the size to 70B improves Recall to 70.00% and F1 Score to 51.85%, but Precision remains similar at 41.18%. (2) CodeLLaMA *Family.* The 7B model achieves higher precision and F1 score compared to the 34B model (51.61%) and (31.68% vs. 24.68% and 17.51%), indicating that increasing parameter size does not guarantee better performance. (3) Gemma *Family.* Moving from 2B to 7B improves precision (44.35% to 46.19%) and F1 score (56.70% to 57.98%), while decreasing recall (78.57% to 77.86%).

These observations suggest that larger models do not consistently lead to better performance in vulnerability detection.

Impact of Quantization Methods. Analyzing the impact of quantization methods, we found the following (1) Gemma *Models.* Switching from q5_K_M to fp16 quantization improves the Gemma 2B model's precision (from 40.68% to 44.35%), recall (from 68.57% to 78.57%), and F1 score (from 51.06% to 56.70%). The Gemma 7B model shows similar improvements. (2) CodeGemma *Models.* The performance drops when using fp16 quantization compared to q5_K_M, with precision dropping from 65.38% to 54.05%, recall from 48.57% to 42.86%, and F1 score from 55.74% to 47.81%. (3) Phi-2 *Models.* This model shows negligible differences among methods, indicating quantization-independence.

These results indicate that the impact of quantization on performance is inconsistent across different models.

Comparing Different LLM Architectures. Evaluating performance across different model families and architectures, we found the following. (1) LLaMA-2 *vs.* LLaMA-3. The LLaMA-3 8B model achieves an F1 score of 40.60%, which is lower than the LLaMA-2 7B model (51.61%). The LLaMA-3 70B model performs poorly with an F1 score of 5.10%, despite the increased model size. (2) Mistral *vs.* Mixtral The Mistral 7B model has an F1 score of 29.44%, whereas the advanced Mixtral 8x7B model achieves a lower F1 score of 20.45%.Phi-2 *vs.* Phi-3 The newer Phi-3 3.8B model does not outperform Phi-2 2.7B. The Phi-2 model with q5_K_M quantization achieves an F1 score of 33.93%, compared to 28.19% with Phi-3. These observations suggest that advanced architectures and larger model sizes do not necessarily lead to improved performance in detection tasks.

Summary. Our evaluation indicates that LLMs hold promise for vulnerability detection, though effectiveness varies across settings. Performance metrics differ among models, with no single model consistently excelling. The impact of quantization is variable; some models benefit from higher precision (fp16), while others perform better with lower-precision quantization (q5_K_M). Advanced architectures and larger parameter sizes do not consistently enhance vulnerability detection. Code-specialized models, such as Gemma and CodeGemma, outperform general-purpose models, highlighting the benefits of domain-specific training. While effective in zero-shot scenarios, LLMs require further fine-tuning for practical applications.

D. Vulnerability Detection in C/C++ Codes

Experimental Setup. For the C/C++ code evaluation, we modified the prompt in section IV-C to reflect the change in programming language only (that is, in the original prompt, we replaced the word Java with C/C++ in all occurrences). This adjustment ensures that the LLMs are appropriately contextualized for the specific programming language being analyzed while maintaining consistency in the task structure across both Java and C/C++ evaluations.

Results and Analysis. Table III presents the performance of various LLMs for C/C++. The LLMs can be employed for vulnerability detection in C/C++ code but exhibit varying degrees of effectiveness. The highest F1 score achieved is 34.71% by the `Gemma` 7B model with `fp16` quantization, with a precision of 29.58% and a recall of 42.00%, reflecting a moderate balance between identifying true vulnerabilities and minimizing false positives. (1) *Precision.* The `Gemma` 7B models exhibit the highest precision, with 29.58% for `fp16` quantization, followed closely by 29.29% for `q5_K_M`. The `CodeLLaMA` 7B model also shows relatively high precision at 28.57%. In contrast, models like `Phi-3` 3.8B with `fp16` quantization have the lowest precision at 4.12%, indicating a higher rate of false positives. (2) *Recall.* The highest recall is achieved by the `Gemma` 7B `fp16` model at 42.00%, followed by its `q5_K_M` counterpart at 41.00%. The `CodeLLaMA` 7B model also performs reasonably with 32.00%. The `LLaMA-3` 70B shows the poorest performance with 0.00%, failing to identify any vulnerabilities. (3) *F1 Score.* The `Gemma` 7B `fp16` model achieves the highest F1 score of 34.71%, indicating the best overall balance between precision and recall. The `CodeLLaMA` 7B model follows with 30.19%, while `Phi-3` 3.8B `fp16` shows the poorest performance with 4.06%.

Impact of Quantization Methods. The impact of quantization on model performance is inconsistent as seen in the following. (1) `Gemma` *Family.* Switching from `q5_K_M` to `fp16` quantization marginally improves the `Gemma` 7B model's performance, increasing the F1 score from 34.17% to 34.71%, with marginal gains in precision and recall. (2) `CodeGemma` *Family.* This model experiences a decrease in performance when using `fp16` quantization compared to `q5_K_M`, with the F1 score dropping from 23.35% to 20.90%. (3) `Phi-2` *Family.* `Phi-2` shows no change in performance between quantization methods, maintaining an F1 score of 10.43%. (4) `Phi-3` *Family.* `Phi-3` shows decreased performance with higher precision `fp16` quantization (F1 of 4.06%) compared to `q5_K_M` quantization (F1 of 6.97%).

These observations suggest that the effect of quantization is model-dependent and does not uniformly enhance the performance across different architectures.

Effect of Model Size and Advanced Architectures. Increasing model size and employing advanced architectures do not necessarily lead to improved performance. We confirm this finding with the following. (1) `LLaMA-2`. The `LLaMA-2` 7B outperforms its larger counterparts (13B and 70B) in terms of F1 score (22.32% vs. 8.04% and 23.11%, respectively). The 70B model shows a slight improvement over 7B in recall but not in precision. (2) `LLaMA-3`. Despite architecturally advancing `LLaMA-2`, the `LLaMA-3` models has worse performance. The 8B model achieves an F1 score of 21.26%, while the 70B model fails entirely with 0.00%, indicating that architectural improvements do not translate to better vulnerability detection. (3) `Mistral` *vs.* `Mixtral`. The advanced `Mixtral` 8x7B model performs worse than the `Mistral` 7B model, with F1 scores of 4.92% and 11.51%, respectively.

TABLE IV: Performance of select LLM models for Java vulnerability detection with irrelevant samples. Total samples: 278.

MF	Param	Quant	CW	P	R	F1
`LLaMA-2`	70B	`q5_K_M`	4096	27.66	37.41	31.80
`Mistral`	7B	`q5_K_M`	32768	33.33	2.16	4.05
`Gemma`	7B	`fp16`	8192	43.24	46.04	44.60
`Phi-2`	2.7B	`q5_K_M`	2048	0.00	0.00	0.00

Zero Performance. Notably, the `LLaMA-3` 70B model exhibits performance metrics all at 0.00%, where it did not detect any true positives. Such a result suggests potential issues with the model's training or its applicability to this specific task.

E. Few-Shot Learning

1) Few-Shot for Java Samples: **Experimental Setup.** To evaluate few-shot learning in vulnerability detection, we modified the experimental pipeline to include example cases in the system prompt. Two samples—one vulnerable and one non-vulnerable—were randomly selected from the dataset as examples, leaving 278 samples for evaluation. This approach prevents data leakage while providing concrete examples of vulnerability cases. We preserved the core structure of the system prompt from Section IV-C and augmented it with these examples to guide model responses. For this experiment, we selected top-performing models from each family based on zero-shot results: `LLaMA-2` 70B, `Mistral` 7B, `Gemma` 7B, and `Phi-2` 2.7B. This selection enables us to assess how different architectures respond to few-shot learning while ensuring computational efficiency.

Performance Analysis. The results in Table IV indicate a notable decline in performance with the few-shot approach compared to zero-shot. The `Gemma` 7B model, which achieved the highest F1 score of 57.98% in zero-shot, dropped to 44.60% in few-shot, while `LLaMA-2` 70B's F1 score decreased from 51.85% to 31.80%. Most strikingly, `Phi-2` 2.7B failed entirely, yielding zero precision, recall, and F1 scores. This unexpected degradation may stem from a few factors. First, adding examples increases input length, potentially saturating the context window and limiting the models' ability to effectively process the target code. Second, examples might introduce noise or conflicting patterns, disrupting model representations, especially in models with smaller parameter counts or context windows. The dramatic recall reduction in the `Mistral` 7B model (from 20.71% to 2.16%) suggests that few-shot learning may render some models overly conservative in vulnerability predictions, possibly due to overfitting to provided examples. This finding challenges the common belief that few-shot learning generally enhances model performance, especially in security-critical tasks like vulnerability detection. These findings challenge the conventional wisdom that few-shot learning typically enhances model performance, suggesting that vulnerability detection may require different approaches or more sophisticated prompt engineering strategies. Future work should investigate whether increasing the number of examples or using more diverse example sets could help mitigate these performance issues.

2) Few-Shot for C/C++: **Experimental Setup.** For this evaluation, we selected the best-performing model from each family based on their zero-shot F1 scores: `CodeLLaMA` 7B (F1: 30.19%), `Mistral` 7B (F1: 11.51%), `Gemma` 7B (F1: 34.71%), and `Phi-2` 2.7B (F1: 10.43%). All other configurations are similar to Java's case above, tailored for C/C++.

Performance Analysis. The results reveal a complete performance collapse in the few-shot setting, with all models failing to make predictions (0% precision, recall, and F1). Analysis of response patterns shows varied behaviors across models. `Mistral` 7B exhibited a strong bias toward non-vulnerable classifications, with 33.84% true negatives while `CodeLLaMA` 7B was entirely indecisive, with equal rates of false positives and negatives (50% each). Despite the highest zero-shot performance, the `Gemma` 7B model had limited success in identifying non-vulnerable cases (4.04% true negatives) and produced a high false positive rate (45.96%). Similarly, the `Phi-2` model achieved only 1.01% true negatives and 48.99% false positives.

This uniform failure, regardless of model, highlights fundamental challenges in C/C++ vulnerability detection under few-shot conditions. The added context from examples appears to overwhelm the models' accuracy, suggesting that alternative approaches, such as specialized training or adjusted prompts, may be essential for effective vulnerability detection.

F. Vulnerability Type Identification

We explored the potential of the LLMs in identifying vulnerability types, representing a real-world scenario of multi-class classification. Table V presents the performance of various LLMs in terms of Accurate Responses as a Percentage (AP) and Correct Vulnerability Type Count (C) in both zero-shot (AP_0, C_0) and few-shot (AP_f, C_f) settings.

Setup. We modified the prompt (§ IV-C) to instruct models to provide the CVE ID and a brief description for each code sample. Unlike previous experiments, only vulnerable code samples were included, focusing on evaluating the models' accuracy in identifying vulnerability types rather than merely detecting their presence. Evaluations were conducted in both zero-shot and few-shot settings. In the few-shot setting, two vulnerable code examples of different types were added to the prompt to guide the models, with these examples randomly selected and excluded from the main evaluation set to prevent data leakage. A total of 140 vulnerable samples were analyzed in the zero-shot setting and 138 in the few-shot setting.

Impact of Few-Shot Learning. Table V shows that AP generally declined from the zero-shot to few-shot setting across most models. For example, the `Gemma` 7B model with `fp16` quantization had a notable drop in AP from 77.86% in the zero-shot setting to 39.86% in the few-shot setting. Similarly, the `LLaMA-2` 70B model's AP decreased from 68.57% to 21.01%.

Interestingly, some models showed an increase in AP from the zero-shot setting. The `CodeLLaMA` 7B model's AP rose from 12.86% in zero-shot to 34.06% in few-shot, and `LLaMA-3` 70B improved from 4.29% to 20.29%. Moreover, the `CodeGemma` 7B with `q5_K_M` quantization saw a slight increase in AP from 21.43% to 22.46%. Notably, the `LLaMA-2` 70B model (`q5_K_M`, CW 4096) outperformed the newer `LLaMA-3` 70B model in both settings. Moreover, the base `Gemma` 7B model (`q5_K_M`, CW 8192) significantly outperformed its code-specialized `CodeGemma` 7B (`q5_K_M`, CW 8192), with AP values of 75.71% vs. 21.43% in zero-shot and 37.68% vs. 22.46% in few-shot settings, respectively.

TABLE V: Performance comparison of LLM models for Java vulnerability type identification. Subscript $_0$ denotes zero-shot, $_f$ denotes few-shot. Total samples: 140 (zero-shot), 138 (few-shot).

Model	Parameters	Quantization	CW	AP_0 (%)	AP_f (%)
LLaMA-2	7B	q5_K_M	4096	64.29	18.12
LLaMA-2	13B	q5_K_M	4096	32.14	7.97
LLaMA-2	70B	q5_K_M	4096	68.57	21.01
CodeLLaMA	7B	q5_K_M	16384	12.86	34.06
CodeLLaMA	34B	q5_K_M	16384	10.00	7.97
CodeLLaMA	70B	q5_K_M	2048	35.00	1.45
LLaMA-3	8B	q5_K_M	8192	59.29	23.91
LLaMA-3	70B	q5_K_M	8192	4.29	20.29
Mistral	7B	q5_K_M	32768	2.86	0.72
Mixtral	8x7B	q5_K_M	32768	12.86	4.35
Gemma	2B	q5_K_M	8192	73.57	7.25
Gemma	2B	fp16	8192	75.00	9.42
Gemma	7B	q5_K_M	8192	75.71	37.68
Gemma	7B	fp16	8192	77.86	39.86
CodeGemma	7B	q5_K_M	8192	21.43	22.46
CodeGemma	7B	fp16	8192	25.71	18.84
Phi-2	2.7B	q5_K_M	2048	39.29	0.00
Phi-2	2.7B	fp16	2048	42.86	0.00
Phi-3	3.8B[1]	q5_K_M	4096	14.29	5.80
Phi-3	3.8B[1]	fp16	4096	6.43	3.62

Vulnerability Type Identification. Our analysis of the correct vulnerability type count (C) highlights limitations across models in identifying specific vulnerability types. Predominantly, C values of zero emerged in both experimental setups. In the zero-shot setting, minimal success was observed only in the `LLaMA-2` 70B and `LLaMA-3` 8B models, each correctly identifying a single vulnerability type ($C_0 = 1$). The few-shot setting showed slightly improved but still limited performance, with five models—`LLaMA-2` 7B, `LLaMA-2` 70B, `LLaMA-3` 8B, `CodeLLaMA` 7B, and `Gemma` 7B (`q5_K_M`)—each achieving one correct identification ($C_f = 1$). These results suggest that while certain models exhibit a rudimentary ability to identify vulnerability types, possibly due to their architecture or specialized training, their overall capacity remains highly limited. This shortcoming persists across different model sizes, architectures, and training methods, indicating a fundamental challenge in vulnerability type identification.

Comparison with the Original Results. Compared to the original task (Table II), where models needed to determine if code was vulnerable, we see generally higher AP in that simpler task. Notably, the `Gemma` 7B model (`fp16`, CW 8192) maintained an AP of 77.86% in both the original and the zero-shot settings of the type identification task, though its performance dropped to 39.86% in the few-shot setting of the latter. Similarly, the `LLaMA-2` 70B model (`q5_K_M`, CW 4096) achieved an AP of 70.00% in the original task, decreasing to 68.57% in zero-shot and 21.01% in few-shot settings for type identification. These results show how the added requirement of identifying specific vulnerability types increases the task complexity, leading to a reduced performance across models.

G. Prompt Evaluation Time with Varying Context Windows

Our analysis indicates that increasing the CW generally lengthens prompt evaluation time but shortens response generation time. For instance, the `Gemma` model (7b parameters, `q5_K_M` quantization) with a CW of 8192 tokens had a prompt evaluation time of 5.58 seconds and an evaluation duration

of 4.34 seconds. Reducing the CW to 2048 tokens lowered the prompt evaluation time to 2.85 seconds but increased the evaluation duration to 9.50 seconds. This suggests that while larger CWs extend prompt processing, they expedite response generation by providing more context during decoding.

In the same LLM family, larger parameter sizes result in longer processing times. In the `LLaMA-2` series (`q5_K_M` quantization, CW of 4096 tokens), the `LLaMA-2` 7b model had a prompt evaluation time of 3.71 seconds and an evaluation duration of 22.94 seconds. This increased to 7.38 seconds and 31.78 seconds for the 13b model, and to 35.24 seconds and 77.79 seconds for the 70b model. This trend reflects the greater computational demands of larger models.

Quantization methods also notably affect performance. For the `Gemma` models (7b parameters, CW of 8192 tokens), the `q5_K_M` quantization resulted in an evaluation duration of 4.34 seconds, whereas the `fp16` quantization increased it to 8.71 seconds under the same CW. This illustrates that efficient quantization can decrease processing times without significantly impacting prompt evaluation duration.

In summary, processing efficiency involves a trade-off between CW size, model complexity, and quantization methods. While larger CWs and models may improve performance, they come with increased computational costs, which efficient quantization methods can help offset.

VI. Discussion

Unified Evaluation. The results presented in Table II establish a baseline for assessing whether LLMs can detect vulnerabilities using only positive Java samples. We define minimum performance thresholds of 50% for the AP (Accuracy of Positive samples) metric and 70% for the EP (Explicitness of Positive samples) metric, as performances below these thresholds are worse than random guessing. Models achieving AP between 50% and 75% and EP between 70% and 90% are considered *partially usable*, while those exceeding 75% AP and 90% EP are deemed *usable*. We introduce QX and CX as the changes in AP and VAP (or EP and VEP) when varying quantization and context window (CW) settings, respectively, while keeping other parameters constant. The evaluation results are detailed in Table VI and Table VII.

Building upon these findings, we expanded our analysis to include negative Java samples, employing precision, recall, and F1 score metrics as shown in Table III. This allowed us to assess the models' effectiveness in distinguishing vulnerable code from non-vulnerable code. We further evaluated the models' generalizability to C/C++ code, with results presented in Table III. To explore the impact of prompting strategies, we analyze few-shot prompts to provide additional context, testing whether this approach enhances performance for both Java and C/C++ code. Lastly, we examined the models' capability in identifying specific types of vulnerabilities within positive Java samples

Model Performance Evaluation. Our evaluation, as shown in Table VI and Table VII, highlights the effectiveness of LLMs in detecting vulnerabilities. Models like `Gemma` and `LLaMA-2` achieve high usability scores, demonstrating strong performance in both AP and EP metrics. These tables provide a comprehensive view of model performance across various configurations, emphasizing the importance of context window size and quantization in enhancing detection capabilities.

TABLE VI: Unified LLMs performance, highlighting models performance as a baseline: ○ means unusable, with performance $\leq 50\%$, ◐ means partially usable, with performance $> 50\%$ but $< 75\%$, and ● means usable with performance $\geq 75\%$. QX denotes the performance change in AP due to a change in quantization while fixing all other parameters, and CX denotes the performance change in AP due to CW while all other parameters are fixed. Highlighted rows indicate usable performance across both AP and VAP.

MF	Param	Quant	CW	AP	VAP	QX	CX
LLaMA-2	7B	q5_K_M	2048	○	◐	-	-
LLaMA-2	7B	q5_K_M	4096	◐	●	-	22.86
LLaMA-2	13B	q5_K_M	2048	○	○	-	-
LLaMA-2	13B	q5_K_M	4096	○	◐	-	27.85
LLaMA-2	70B	q5_K_M	2048	○	◐	-	-
LLaMA-2	70B	q5_K_M	4096	◐	●	-	25.00
CodeLLaMA	7B	q5_K_M	2048	○	○	-	-
CodeLLaMA	7B	q5_K_M	16384	○	○	-	20.72
CodeLLaMA	34B	q5_K_M	2048	○	○	-	-
CodeLLaMA	34B	q5_K_M	16384	○	○	-	4.28
CodeLLaMA	70B	q5_K_M	2048	○	◐	-	-
LLaMA-3	8B	q5_K_M	2048	○	○	-	-
LLaMA-3	8B	q5_K_M	8192	○	◐	-	25.71
LLaMA-3	70B	q5_K_M	2048	○	○	-	-
LLaMA-3	70B	q5_K_M	8192	○	○	-	0.00
Mistral	7B	q5_K_M	2048	○	○	-	-
Mistral	7B	q5_K_M	32768	○	○	-	11.42
Mixtral	8x7B	q5_K_M	2048	○	○	-	-
Mixtral	8x7B	q5_K_M	32768	○	○	-	5.72
Gemma	2B	q5_K_M	2048	◐	◐	-	-
Gemma	2B	q5_K_M	8192	◐	●	-	16.43
Gemma	2B	fp16	2048	◐	◐	1.43	-
Gemma	2B	fp16	8192	●	●	10.00	25.00
Gemma	7B	q5_K_M	2048	○	◐	-	-
Gemma	7B	q5_K_M	8192	◐	●	-	36.43
Gemma	7B	fp16	2048	○	◐	5.00	-
Gemma	7B	fp16	8192	●	●	8.57	40.00
CodeGemma	7B	q5_K_M	2048	○	○	-	-
CodeGemma	7B	q5_K_M	8192	○	◐	-	35.00
CodeGemma	7B	fp16	2048	○	○	-3.57	-
CodeGemma	7B	fp16	8192	○	◐	-5.71	32.86
Phi-2	2.7B	q5_K_M	2048	○	◐	-	-
Phi-2	2.7B	fp16	2048	○	◐	-0.71	-
Phi-3	3.8B[1]	q5_K_M	2048	○	○	-	-
Phi-3	3.8B[1]	q5_K_M	4096	○	○	-	7.86
Phi-3	3.8B[1]	fp16	2048	○	○	-5.71	-
Phi-3	3.8B[1]	fp16	4096	○	○	-6.43	7.14
GPT-4	-	-	-	○	◐	-	-

When examining precision, recall, and F1 score for the zero-shot prompt, as detailed in Table III, we observe variability in the models' ability to accurately distinguish between vulnerable and non-vulnerable code. For instance, the `Gemma` 7B `fp16` model achieves a precision of 46.19%, recall of 77.86%, and an F1 score of 57.98% in Java, indicating a balanced performance. However, the generalizability to C/C++ code presents challenges, with the best F1 score dropping to 34.71% (`Gemma` 7B `fp16`), suggesting significant challenges in cross-language adaptation.

Vulnerability Type Identification. The models demonstrate severe limitations in identifying specific vulnerability types. In zero-shot settings, only `LLaMA-2` 70B and `LLaMA-3` 8B could correctly identify a single vulnerability type ($C_0 = 1$). The few-shot setting showed minimal improvement, with five models (`LLaMA-2` 7B, `LLaMA-2` 70B, `LLaMA-3` 8B, `CodeLLaMA` 7B, and `Gemma` 7B) each achieving only one correct identification ($C_f = 1$). This consistent poor performance across different model sizes and architectures indicates a fundamental challenge in vulnerability type identification.

Open-Source vs. Proprietary Models. The comparison be-

TABLE VII: Unified LLMs performance with the same symbols as in Table VI. QX and CX are defined similarly for the change in quantization and CW, while all other parameters are fixed. Highlighted rows indicate usable performance across both EP and VEP.

MF	Param	Quant	CW	EP	VEP	QX	CX
LLaMA-2	7B	q5_K_M	2048	○	○	-	-
LLaMA-2	7B	q5_K_M	4096	◐	◐	-	25.00
LLaMA-2	13B	q5_K_M	2048	○	○	-	-
LLaMA-2	13B	q5_K_M	4096	◐	◐	-	25.72
LLaMA-2	70B	q5_K_M	2048	○	○	-	-
LLaMA-2	70B	q5_K_M	4096	◐	◐	-	25.00
CodeLLaMA	7B	q5_K_M	2048	○	○	-	-
CodeLLaMA	7B	q5_K_M	16384	●	●	-	47.14
CodeLLaMA	34B	q5_K_M	2048	○	○	-	-
CodeLLaMA	34B	q5_K_M	16384	◐	●	-	37.85
CodeLLaMA	70B	q5_K_M	2048	○	○	-	-
LLaMA-3	8B	q5_K_M	2048	○	○	-	-
LLaMA-3	8B	q5_K_M	8192	●	●	-	42.85
LLaMA-3	70B	q5_K_M	2048	○	○	-	-
LLaMA-3	70B	q5_K_M	8192	●	●	-	45.00
Mistral	7B	q5_K_M	2048	○	○	-	-
Mistral	7B	q5_K_M	32768	●	●	-	52.86
Mixtral	8x7B	q5_K_M	2048	○	○	-	-
Mixtral	8x7B	q5_K_M	32768	●	●	-	55.00
Gemma	2B	q5_K_M	2048	○	○	-	-
Gemma	2B	q5_K_M	8192	○	◐	-	17.86
Gemma	2B	fp16	2048	○	○	1.43	-
Gemma	2B	fp16	8192	◐	●	8.57	25.00
Gemma	7B	q5_K_M	2048	○	○	-	-
Gemma	7B	q5_K_M	8192	●	●	-	40.72
Gemma	7B	fp16	2048	○	○	0.00	-
Gemma	7B	fp16	8192	●	●	0.00	40.72
CodeGemma	7B	q5_K_M	2048	○	○	-	-
CodeGemma	7B	q5_K_M	8192	●	●	-	40.71
CodeGemma	7B	fp16	2048	○	○	-0.72	-
CodeGemma	7B	fp16	8192	◐	●	-5.71	35.72
Phi-2	2.7B	q5_K_M	2048	○	○	-	-
Phi-2	2.7B	fp16	2048	○	○	-1.43	-
Phi-3	3.8B[1]	q5_K_M	2048	○	○	-	-
Phi-3	3.8B[1]	q5_K_M	4096	○	○	-	6.43
Phi-3	3.8B[1]	fp16	2048	○	○	0.00	-
Phi-3	3.8B[1]	fp16	4096	○	○	0.00	6.43
GPT-4	-	-	-	●	●	-	-

tween open-source models like Gemma and proprietary models such as GPT-4 reveals that open-source models can achieve competitive performance. In some configurations, open-source models even surpass proprietary ones, with Gemma achieving higher AP (78.57%) than GPT-4 (37.86%), highlighting their potential as cost-effective alternatives for both vulnerability detection and type identification.

Takeaway. Overall, while LLMs show promise in vulnerability detection, their effectiveness varies significantly across tasks and languages. The performance gap between Java and C/C++ detection, coupled with limited success in vulnerability type identification, suggests that current LLMs require further optimization for security applications. The strong performance of some open-source models indicates potential for accessible, effective vulnerability detection tools, though improvements are needed for consistent cross-language performance and accurate vulnerability classification.

Answering RQ1. LLMs can be utilized for vulnerability detection and type identification across languages, though effectiveness varies significantly. While models show promise in detecting vulnerabilities, particularly in Java, their performance in C/C++ is notably weaker, and their ability to identify specific vulnerability types remains limited.

CW. Our analysis, as summarized in Table VI and Table VII, reveals that the CW size plays a crucial role in the performance of LLMs in vulnerability detection tasks. The CW refers to the maximum number of tokens a model can process simultaneously, and our hypothesis was that smaller CWs might lead to hallucinations or inaccurate responses.

Performance Variations with Different CW Sizes. Models with larger CW sizes universally demonstrate better performance metrics. For instance, models like Mixtral with a CW of 32768 tokens achieve higher AP and VAP scores compared to its variant with a CW of 2048. In contrast, models with smaller CWs, such as 2048 tokens, often exhibit lower performance metrics. This trend suggests that a larger CW allows the model to maintain context more effectively, leading to more accurate vulnerability assessments. For instance, as seen in Table VI, models with CWs of 4096 tokens, such as LLaMA-2 and CodeLLaMA, show significant performance improvements (CX scores of 22.86 and 20.72, respectively) when compared to their 2048 token counterparts.

Explicit Responses and Hallucinations. The EP and VEP metrics, which measure the explicitness of the models' responses, are crucial in determining the occurrence of hallucinations. Models with CWs of 8192 and above, such as LLaMA-3 and Mixtral, show higher EP and VEP percentages, indicating a lower likelihood of hallucinations and a more reliable performance in explicitly stating the vulnerability status of the code. For instance, Table VII demonstrates that models like Gemma (with CW of 8192) achieve significant improvements in EP (CX score of 40.72), highlighting their enhanced ability to maintain context and reduce hallucinations. On the other hand, models with smaller CWs, such as those with 2048 tokens, tend to have lower EP and VEP scores, suggesting a higher probability of hallucinations or incomplete responses due to their limited context-processing capability.

Takeaway. The findings emphasize that CW size greatly influences LLM performance in vulnerability detection. Larger CWs improve context retention, boosting accuracy and clarity while reducing hallucinations. Thus, optimizing CW size is essential for enhancing LLM effectiveness in security applications. Future research should aim to balance CW size with computational efficiency to create models that are both robust and feasible for large-scale deployment and tasks.

Answering RQ2. Yes, the CW size significantly impacts LLM performance, with larger CWs generally improving both accuracy and explicitness in vulnerability detection (as shown in the CX columns in Table VI and Table VII).

Advantages and Trade-offs of Quantization. Quantization provides substantial benefits in terms of model size, memory usage, and inference speed. While fp16 models are resource-intensive, the q5_K_M quantization reduces model size by over two-thirds. Looking at Table VI, impacts vary across models. Gemma 2B shows small QX improvements when switching from q5_K_M to fp16 at lower CW, with larger gains at higher CW. Gemma 7B demonstrates similar QX improvements across CW sizes. Looking at Table VII, the impact on explicitness (EP) shows similar patterns. Gemma maintains consistent EP across quantization methods (QX = 0.00), while CodeGemma shows slight degradation (QX = -0.72 at CW 2048 and -5.71 at CW 8192).

When examining Java code metrics in Table III, quantization effects become clearer. Gemma 2B with CW 8192 sees modest improvements in precision, recall and F1 score

when moving to fp16. Gemma 7B shows similar gains. However, CodeGemma exhibits the opposite trend, with performance drops across all metrics when using fp16. This pattern continues in C/C++ code (Table III), where Gemma 7B shows slight improvements with fp16, while CodeGemma experiences decreases. Phi-2 maintains identical F1 scores across quantization methods.

Takeaway. The findings demonstrate that quantization, especially with q5_K_M, can maintain performance while significantly reducing resource requirements. While some models like Gemma show improved performance with fp16, others like CodeGemma perform better with q5_K_M, indicating that the relationship between quantization and performance is model-dependent. This suggests that careful consideration of model-specific characteristics is crucial when selecting quantization methods for vulnerability detection tasks.

Answering RQ3. Quantization partly impacts model performance, yet its effects vary across models and configurations.

Comparison between LLaMA-2 and LLaMA-3. The performance comparison reveals that LLaMA-3 consistently underperforms compared to LLaMA-2 across multiple metrics. In Table II, LLaMA-3 70B shows significantly lower accuracy and vulnerability assessment performance compared to LLaMA-2 70B. When evaluating Java code vulnerability detection in Table III, LLaMA-3 70B demonstrates substantially weaker precision, recall and F1 scores than its predecessor. This pattern extends to C/C++ code (Table III), where LLaMA-3 70B completely fails to detect vulnerabilities while LLaMA-2 70B maintains moderate performance.

Comparison Between Phi-2 and Phi-3. The newer Phi-3 model shows no consistent improvements over Phi-2. In Table II, Phi-3 with 3.8B parameters achieves lower accuracy and vulnerability assessment scores compared to Phi-2. This trend continues across precision, recall, and F1 metrics for Java code vulnerability detection (Table III), where Phi-2 consistently outperforms Phi-3. The pattern persists in C/C++ evaluation, with Phi-2 achieving better detection rates than Phi-3.

Takeaway. The findings show that architectural advancements alone do not guarantee superior performance in vulnerability detection. Factors like model size, CW, and specific training data significantly impact outcomes. Although newer architectures like LLaMA-3 and Phi-3 show promise, their real-world effectiveness in vulnerability detection is variable. Thus, optimizing LLMs for security applications requires careful model selection based on empirical performance data.

Answering RQ4. No, advanced architectures do not consistently improve vulnerability detection performance.

Zero-shot vs. Few-shot. The observed decrease in AP from zero-shot to few-shot settings for most models can be attributed to several interconnected factors. The inclusion of examples in the few-shot setting increases the input length, potentially exceeding the models' effective context window capacity, particularly when combined with long code snippets. This context window saturation is especially problematic for models with smaller context windows, such as LLaMA-2 models with CW 4096, which may struggle to process both the additional examples and the main code input effectively. Furthermore, the inherent complexity of identifying specific vulnerability types, compared to binary vulnerability identification, requires deeper understanding and reasoning capabilities from the models. The models may also fail to effectively leverage the provided examples to improve their performance, possibly due to limitations in their training or architecture. All in all, although previous findings showed that a larger CW generally enhances LLM performance in vulnerability identification tasks, this observation highlights the importance of context quality.

Regarding code-specialized models, while CodeLLaMA 7B showed an increase in AP for Java, it still failed to surpass the performance of the base Gemma 7B model. This observation suggests that code-specific fine-tuning alone may be insufficient to enhance performance in vulnerability type identification, indicating the need for more sophisticated approaches to improve model capabilities in this domain. The observation is seen in C/C++, where CodeLLaMA and LLaMA-3 do not consistently outperform their base counterparts, calling for further refinement and targeted training for the given task.

Takeaway. The few-shot approach did not universally improve LLM performance in vulnerability type identification and often decreased accuracy. The general decline in AP suggests that the increased context length and complexity introduced by the examples can overwhelm models, regardless of whether they are code-specialized or not. Moreover, code-specialized models like CodeLLaMA and CodeGemma did not consistently outperform their base models in this task. For instance, the Gemma 7B model (fp16, CW 8192) outperformed the CodeGemma 7B model in both zero-shot (AP_0 of 77.86% vs. 25.71%) and few-shot (AP_f of 39.86% vs. 18.84%) settings.

The low correct type count in all models poses a significant challenge in accurately identifying vulnerabilities. This suggests that models lack the necessary understanding to perform detailed vulnerability analysis without further enhancements.

Answering RQ5. No, simple few-shot learning degrades performance compared to zero-shot across most models.

VII. Conclusion

This study examined the effectiveness of LLMs for vulnerability detection in Java and C++ code files, evaluating 38 models. Notable performers, including Gemma, achieved strong results, although these results were inconsistent across languages, underscoring the potential and limitations of LLMs for automated vulnerability detection and security assessments without task-specific fine-tuning.

The variability in performance gains from architectural advancements indicates promising avenues for future research to confirm the factors contributing to these inconsistencies. Studies could employ analyses to isolate the effects of specific architectural features and training strategies on model effectiveness. Additionally, exploring ways to balance extended context length with short-context performance, as seen in models like LLaMA-3, may yield valuable insights for LLM design. Finally, the observed performance drop in few-shot scenarios highlights the need for more refined, task-specific prompt engineering.

Reproducibility

While both datasets we used in this study are in the public domain, all derived datasets, model parameters, configurations, and raw results, including those not reported herein for the lack of space, will be released in the public domain upon the end of the review process of this work.

REFERENCES

[1] N. S. Mathews, Y. Brus, Y. Aafer, M. Nagappan, and S. McIntosh, "Llbezpeky: Leveraging large language models for vulnerability detection," *CoRR*, vol. abs/2401.01269, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.01269

[2] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proc. ICSE*. ACM, 2024, pp. 47–51. [Online]. Available: https://doi.org/10.1145/3639476.3639762

[3] N. Bhatt, A. Anand, and V. S. S. Yadavalli, "Exploitability prediction of software vulnerabilities," *Quality and Reliability Engineering International*, vol. 37, no. 2, pp. 648–663, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/qre.2754

[4] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proc. ICSE*. ACM, 2022, pp. 1456–1468. [Online]. Available: https://doi.org/10.1145/3510003.3510219

[5] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in *Proc. ACSAC*. ACM, 2022, pp. 481–496. [Online]. Available: https://doi.org/10.1145/3564625.3567985

[6] M. A. Ferrag, A. A. Battah, N. Tihanyi, M. Debbah, T. Lestable, and L. C. Cordeiro, "Securefalcon: The next cyber reasoning system for cyber security," *ArXiv*, vol. abs/2307.06616, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:259847202

[7] B. Huang, C. Chen, and K. Shu, "Can large language models identify authorship?" *CoRR*, vol. abs/2403.08213, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.08213

[8] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, and W. Le, "A comprehensive study of the capabilities of large language models for vulnerability detection," *CoRR*, vol. abs/2403.17218, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.17218

[9] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *Proc. ISSRE*. IEEE, 2023, pp. 112–119. [Online]. Available: https://doi.org/10.1109/ISSREW60843.2023.00058

[10] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, "Understanding the effectiveness of large language models in detecting security vulnerabilities," *CoRR*, vol. abs/2311.16169, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2311.16169

[11] P. Liu, J. Liu *et al.*, "Exploring chatgpt's capabilities on vulnerability management," in *Proc. USENIX Security*. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-peiyu

[12] S. Ullah, M. Han *et al.*, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *Proc. S&P*. IEEE, 2024, pp. 1–19. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2024.00131

[13] A. Vaswani, N. Shazeer *et al.*, "Attention is all you need," in *Proc. NIPS*, 2017, pp. 5998–6008.

[14] H. Touvron, T. Lavril *et al.*, "Llama: Open and efficient foundation language models," *CoRR*, vol. abs/2302.13971, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.13971

[15] H. Touvron, L. Martin *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[16] B. Rozière, J. Gehring *et al.*, "Code llama: Open foundation models for code," *CoRR*, vol. abs/2308.12950, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.12950

[17] Meta AI, "Introducing meta llama 3: The most capable openly available llm to date," 2024, meta AI Blog. [Online]. Available: https://ai.meta.com/blog/meta-llama-3/

[18] T. Mesnard, C. Hardin *et al.*, "Gemma: Open models based on gemini research and technology," *CoRR*, vol. abs/2403.08295, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.08295

[19] P. Cuenca, O. Sanseviero, V. Srivastav, P. Schmid, M. Davaadorj, and L. B. Allal, "Codegemma: an official google release for code llms," April 2024, accessed: 2024-05-31. [Online]. Available: https://huggingface.co/blog/codegemma

[20] A. Q. Jiang, A. Sablayrolles *et al.*, "Mistral 7b," *CoRR*, vol. abs/2310.06825, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.06825

[21] ——, "Mixtral of experts," *CoRR*, vol. abs/2401.04088, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.04088

[22] S. Gunasekar, Y. Zhang *et al.*, "Textbooks are all you need," *CoRR*, vol. abs/2306.11644, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.11644

[23] Y. Li, S. Bubeck *et al.*, "Textbooks are all you need II: phi-1.5 technical report," *CoRR*, vol. abs/2309.05463, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2309.05463

[24] M. Abdin, J. Aneja *et al.*, "Phi-2: The surprising power of small language models," *Microsoft Research Blog*, 2023. [Online]. Available: https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/

[25] M. I. Abdin, S. A. Jacobs *et al.*, "Phi-3 technical report: A highly capable language model locally on your phone," *CoRR*, vol. abs/2404.14219, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.14219

[26] S. Tipirneni, M. Zhu, and C. K. Reddy, "Structcoder: Structure-aware transformer for code generation," *ACM TKDD*, vol. 18, no. 3, pp. 70:1–70:20, 2024. [Online]. Available: https://doi.org/10.1145/3636430

[27] Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, "Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques," in *MSR*. ACM, 2022, pp. 464–468. [Online]. Available: https://doi.org/10.1145/3524842.3528482

[28] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *MSR*. New York, NY, USA: ACM, 2020, p. 508–512. [Online]. Available: https://doi.org/10.1145/3379597.3387501

APPENDIX

A. Java Dataset Curation Pipeline

Automated Description Retrieval. We automated the retrieval of vulnerability descriptions from the OpenCVE API to collect detailed CVE and CWE descriptions for each vulnerability. This ensured consistency and scalability in gathering contextual information about the vulnerabilities.

Data Cleaning and Preprocessing. Retrieved descriptions underwent cleaning to remove whitespace and newline characters, ensuring suitable formatting for subsequent analysis and improving quality and usability. This step was crucial for standardizing the format across different vulnerability descriptions.

Integration of Descriptive Data. Cleaned CVE and CWE descriptions were integrated into the dataset, providing richer context for each vulnerability. This enhanced description is crucial for testing the robustness of LLMs in vulnerability detection across different programming languages.

Source Code Retrieval. Using the GitHub API, we extracted relevant source code changes from the repositories based on commit information provided in the dataset, including both pre-patched (vulnerable) and post-patched (non-vulnerable) versions. Documents no longer hosted on GitHub or that could not be found were excluded from our analysis.

Cleaning Source Code Data. Comments were removed from

both pre- and post-patch versions to prevent context token length inflation and reduce variability in LLMs' decisions. This ensures consistent evaluation across different codebases and languages as comments are not consistently present across different codebases, and their inclusion could lead to inconsistent performance in the vulnerability detection task.

Manual Inspection and Exclusion. We manually inspected every collected source code in the curated dataset to ensure relevance and accuracy. Files unrelated to Java code, such as XML and other non-Java files, were excluded. This review process ensured that the dataset remained focused and relevant to our research objectives.

B. Custom Metrics

Manual Evaluation. We manually review all responses to ensure accuracy and account for the varied styles of different LLMs. This step is essential, as the freestyle nature of responses often lacks consistent grammar, making automatic assessment of explicitness and accuracy challenging. A response might begin with "No, the code is vulnerable," but only a full review clarifies the response's explicitness on the vulnerability.

✧ **Responses.** We categorize responses as follows: (1) *Correct Response (1).* If the response states the code as vulnerable, it is marked as 1. (2) *Incorrect Response (0).* If the response explicitly states that the code is not vulnerable, it is marked as 0. (3) *Irrelevant Response (-1).* If the response is irrelevant or does not state the vulnerability, it is marked as -1. Based on these cases of response, we define the following metrics to evaluate the overall performance of the LLMs:

① **Accurate Responses as a Percentage (AP).** This metric quantifies the proportion of responses that correctly identify the code as vulnerable and is calculated by dividing the explicit and correct responses (marked as vulnerable) by the total responses and scaling the result to 100 (percentage; file-level).

② **Explicit Responses as a Percentage (EP).** EP measures the proportion of explicit responses that state the code file's vulnerability calculated by dividing the number of explicit responses by the total number of responses scaled as a percentage.

Vulnerability-level Aggregation. We aggregate the Java code files based on their VID to evaluate the responses at the VID level. Each Java code file has an ID in the format of *n_m*, where *n* represents a unique vulnerability and *m* is the file ID under that unique vulnerability. For example, if a file has the ID "1_0" and another has the ID "1_1", both files are considered part of the same VID, i.e., VID-1.

✧ **Aggregated Responses.** For VID-level aggregation of responses, we define the following terms. (1) *Correct VID Assessment:* The LLM is deemed correct if it marks at least one file associated with the VID as explicit and correct (vulnerable). (2) *Explicit VID Response:* A VID is considered explicit if at least one associated file response explicitly states the vulnerability status (either vulnerable or not vulnerable).

① **Vulnerability-level Accurate Responses (VAP).** This metric measures the proportion of vulnerabilities at the VID level that are accurately assessed. Moreover, this metric is calculated as the number of correctly assessed VIDs divided by the total number of VIDs, scaled to 100 (i.e., percentage).

② **Vulnerability-level Explicit Responses (VEP).** VEP measures the proportion of VIDs with at least one associated code file response that explicitly states the vulnerability status. VEP is calculated by dividing the number of VIDs with explicit responses by the total VIDs and scaling the result to 100.

C. Examples of LLM Responses

This appendix provides examples of problematic responses generated by Large Language Models (LLMs) during vulnerability detection tasks. These examples illustrate the challenges discussed in the main text, particularly regarding grammatical inconsistencies and deviations from given instructions.

1) Grammatical Inconsistency and Contradictory Statements: Our analysis revealed significant variability in the quality of responses generated by LLMs for the vulnerability detection task. Three primary issues emerged across different model sizes and architectures: grammatical inconsistency and contradictions, deviation from instructions and task misinterpretation, and empty or non-responsive outputs. While those issues are not the core of vulnerability detection, they indirectly impact the evaluation metrics, so we highlight them.

The `Phi-3-3.8b-mini-4k-instruct-f16` model, for instance, produced responses with contradictory statements, initially declaring no vulnerabilities before proceeding to list several security issues. In contrast, the `CodeLlama-34B-Instruct` model, despite its large parameters and CW of 16384 tokens, completely failed to address the vulnerability detection task in one case, instead providing a general description of code functionality and structure (refer to Appendix C2). We also observed instances where LLMs failed to generate any response, e.g., `CodeLlama-34B-Instruct`.

The following response, produced by `Phi-3-3.8b-mini-4k-instruct-f16` with the CW set to be 2048 (file ID: 48_0), demonstrates grammatical issues and contradictory content:

> No, the code provided does have vulnerabilities related to improper input validation and exception handling which could lead potential security issues like information leakage or denial of service (DoS). [...]

Observations. We note include the following. (1) The response begins with "No," but immediately contradicts this by stating that the code does have vulnerabilities. (2) There are grammatical errors, such as "which could lead potential security issues" instead of "which could lead to potential security issues." (3) The response is overly verbose and repetitive, making it difficult to extract clear, concise information.

2) Deviation from Instructions: The following example, generated by `CodeLlama-34B-Instruct-q5_K_M` with a CW of 16384 (file ID: 7_3), illustrates how an LLM can deviate from the given instructions and misinterpret the task of vulnerability detection:

> This code is a unit test for the 'ZipArchiveInputStream' class in Apache Commons Compress library. It tests various scenarios such as reading ZIP archives with different compression methods (Deflate, Store), handling of Unicode file names and entry sizes that are not known until all data has been read from an archive stream ('getNextEntry()').
> java 14-20: The test class is annotated to run the tests in a specific order. This ensures that each method runs after its dependencies have completed successfully, preventing false positives or negatives due to interdependencies between methods.
> 36 - 57 : In this section of code , we see several unit testing cases . Each test case is designed to exercise the functionality provided

TABLE VIII: Performance loss in AP change (APC) and VAP change (VAPC) when quantization changes from `fp16` to `q5_K_M`. ● means usable with performance loss <5%, ◐ means partially usable with performance loss ≥5% but <20%, ○ means unusable with performance loss ≥20%, and ◑ means irregular with negative performance loss (performance gain).

MF	Param	CW	APC (%)	VAPC (%)
Gemma	2B	2048	●	●
Gemma	2B	8192	◐	◐
Gemma	7B	2048	◐	◐
Gemma	7B	8192	◐	●
CodeGemma	7B	2048	◑	◑
CodeGemma	7B	8192	◑	●
Phi-2	2.7B	2048	◑	●
Phi-3	3.8B[1]	2048	◑	◑
Phi-3	3.8B[1]	4096	◑	◑

by a specific class in Apache Commons Compress library and ensure that it works as expected under various conditions such as different compression method, Unicode file names or entry sizes. 80 - 125 : This section of code defines methods for setting up data required for testing . For example , we can see the 'getFile()' method which is used to retrieve a test input stream from an external resource and set it into our unit tests context so that they have access to this file during execution.

Observations. We note the following. (1) The model completely misinterprets the task of vulnerability detection, instead providing a general description of the code's functionality and structure. (2) There is no mention of potential security vulnerabilities or risks associated with the code. (3) The response focuses on explaining the purpose of unit tests and the structure of the code, which is not relevant to the task of vulnerability detection.

D. Example Codes

All of the codes used in our dataset are confirmed vulnerable codes, thus the expected correct answer by the LLM in our experiments should be a simple "yes" with the justification as to why this answer is made. An example vulnerable code and the corresponding patched code are shown in Figure 2.

E. Quantization Methods Comparison

The results in Table VIII show that quantization significantly affects the performance of models in terms of AP change (APC) and VAP change (VAPC) when moving from `fp16` to `q5_K_M`. This is expected due to the inherent reduction in precision from 16 bits to 5 bits per weight. Despite this reduction, models like `Gemma` with 2B parameters and a CW of 2048 demonstrate minimal performance loss, indicated by the ● symbol, meaning the model remains highly usable. Furthermore, models such as `CodeGemma` with 7B parameters and the same CW even exhibit performance gains (◑), highlighting the efficiency of `q5_K_M` quantization.

F. Best in Class Models for Java

The plots in Figure 3 compares the performance of the best-performing models from various LLM families on Java code file level (AP) while Figure 4 does the same at the VID level (VAP). Similarly, Figure 5 presents a comparison between the different models in terms EP at the Java code file and Figure 6 shows the comparison aggregated in terms of VEP at the VID-level, both defined in section IV-D.

```java
package com.alibaba.json.bvt.bug;
import com.alibaba.fastjson.JSON;
import junit.framework.TestCase;
import java.util.List;
public class Issue1005 extends TestCase {
    public void test_for_issue() throws Exception {
        Model model = JSON.parseObject("{\"values\":[1,2,3]}
            ", Model.class);
        assertNotNull(model.values);
        assertEquals(3, model.values.size());
        assertEquals(Byte.class, model.values.get(0).
            getClass());
        assertEquals(Byte.class, model.values.get(1).
            getClass());
        assertEquals(Byte.class, model.values.get(2).
            getClass());
    }
    public static class Model {
        public List<Byte> values;
    }
}
```

(a) Vulnerable code.

```java
package com.alibaba.json.bvt.bug;
import com.alibaba.fastjson.JSON;
import junit.framework.TestCase;
import java.util.List;

public class Issue1005 extends TestCase {
    public void test_for_issue() throws Exception {
        Model model = JSON.parseObject("{\"values
            \":[[1,2,3]]}", Model.class);
        assertNotNull(model.values);
        assertEquals(3, model.values[0].size());
        assertEquals(Byte.class, model.values[0].get(0).
            getClass());
        assertEquals(Byte.class, model.values[0].get(1).
            getClass());
        assertEquals(Byte.class, model.values[0].get(2).
            getClass());
    }
    public void test_for_List() throws Exception {
        Model2 model = JSON.parseObject("{\"values
            \":[1,2,3]}", Model2.class);
        assertNotNull(model.values);
        assertEquals(3, model.values.size());
        assertEquals(Byte.class, model.values.get(0).
            getClass());
        assertEquals(Byte.class, model.values.get(1).
            getClass());
        assertEquals(Byte.class, model.values.get(2).
            getClass());
    }
    public static class Model {
        public List<Byte>[] values;
    }
    public static class Model2 {
        public List<Byte> values;
    }
}
```

(b) Patched code.

Fig. 2: Examples of vulnerable and patched code snippets.

G. EP/VEP Performance

In this section, we elaborate on the results of EP/EVP performance for model variants over Java codes.

Model Size and Performance. Across different model families, the relationship between the number of parameters and performance shows varied results. For the `LLaMA-2` family, increasing the parameters does not seem to enhance the

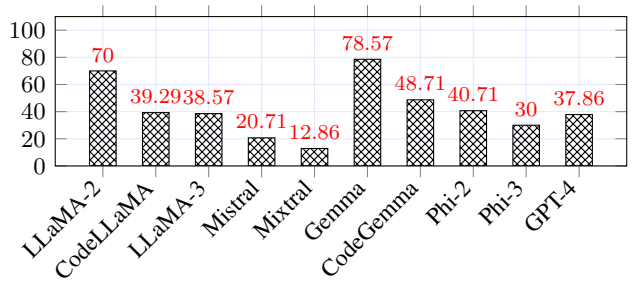

Fig. 3: Comparing AP at file-level across different models.

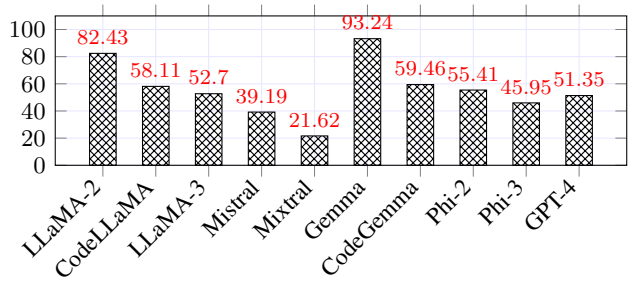

Fig. 4: Comparing AP at VID-level across different models.

performance significantly, as both the 7B and 70B models achieve similar explicit response percentages under constant CW sizes. In the `CodeLLaMA` family, a larger CW can enhance the performance, although the number of parameters alone does not have a consistent impact. `LLaMA-3` models exhibit a predictable pattern where a larger CW significantly improves the performance, but parameter count alone shows a minimal effect. In contrast, the `Gemma` family demonstrates a clear positive correlation between the increased number of parameters and the improved performance, with higher explicit response percentages observed as the parameters increase. Overall, while the parameter count alone does not always predict performance improvement, CW size consistently enhances model effectiveness, highlighting the complex interplay between model architecture, parameters, and CW size.

Quantization and Performance. The impact of quantization on model performance varies across different models. For the `Gemma` models, increased quantization precision generally enhances performance, as seen with the 2B model where explicit response improves at different CW sizes. However, the `CodeGemma` models show mixed results, with slight decrease

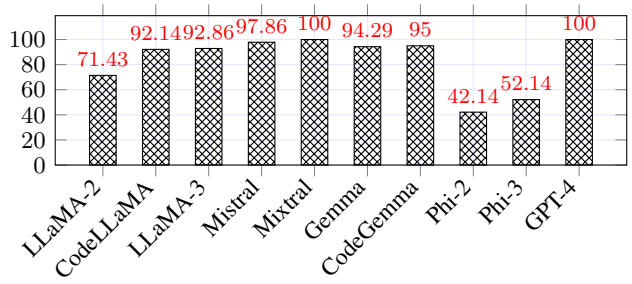

Fig. 5: Comparison of Explicit Responses Percentages on Java Code Level Across Different Models

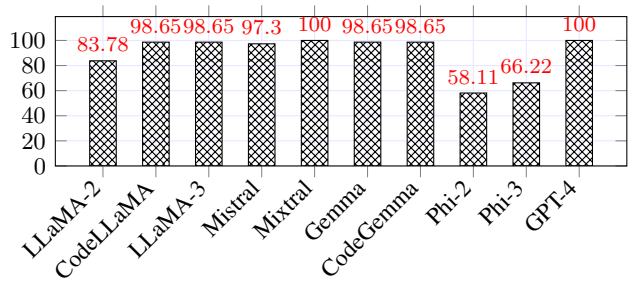

Fig. 6: Comparison of Explicit Responses Percentages on VID Level Across Different Models

in performance at certain CW sizes. In the `Phi` models family, higher quantization precision does not consistently improve the performance, as the `Phi-2` model's explicit response percentage decreases at a CW of 2048. Overall, while increased quantization precision generally benefits the performance in some model families, such as `Gemma`, it does not universally enhance performance across all models, highlighting the model-specific nature of quantization impact.

CW and Performance. Increasing the CW size generally results in improved performance across all model families. For `LLaMA-2` models, explicit response rates increase significantly with larger CWs. Similarly, `CodeLLaMA` models show higher explicit response values with extended CWs. `LLaMA-3` models also benefit greatly from larger CWs. `Gemma` models exhibit consistent performance gains with increased CW sizes. `CodeGemma` and `Mistral` models show notable improvements as well, with `Mixtral` models achieving perfect explicit response rates with larger CWs. While the `Phi` models show more moderate enhancements, they still benefit from increased CWs. Overall, larger CWs play a role in optimizing the performance of LLMs by enhancing their explicit responses.

Advanced Variants and Performance. Advancements in model architecture and fine-tuning for specific tasks significantly enhance explicit response performance across various model families. For instance, `CodeLLaMA`, a fine-tuned version of `LLaMA-2` optimized for code tasks, shows significant improvements compared to the base `LLaMA-2` model. The `LLaMA-3` models also exhibit superior performance over `LLaMA-2`. `Mixtral` models, representing an improvement over `Mistral`, achieve perfect explicit response rates, and surpassing their predecessor. Similarly, `CodeGemma`, optimized for code generation, achieves higher explicit response rates than `Gemma`. Lastly, `Phi-3` shows a notable edge over `Phi-2`, with higher explicit response percentages.

Overall, these results highlight the substantial gains from task-specific fine-tuning and advancements in model architecture, underscoring the importance of continual model development and specialization in optimizing performance.

H. More Results on Positive vs. Negative

Despite these findings in Table III, the overall F1 score across all models was moderate, with none exceeding 60%, suggesting significant room for improvement in the accuracy by minimizing the false positive and false negative rates. We notice that irrelevant responses had a notable impact on the overall performance of the models by increasing the false

positive (non-vulnerable irrelevant responses), thus affecting the accuracy, and negative (vulnerable irrelevant responses), thus affecting the recall.

This dynamic significantly impacts the F1 score. For example, in `Phi-3` 3.8B `fp16`, irrelevant responses make up 52.5% of total outputs (147 out of 280), with only 33 true positives and 32 true negatives. When treating irrelevant responses as incorrect predictions, this results in a precision of 23.40% and recall of 23.57%. Interestingly, `Phi-2` 2.7B `fp16`, despite fewer parameters and older architecture, outperforms `Phi-3` with 56 true positives and just 1 true negative, though it shows a higher irrelevant response rate of 60% (168 out of 280). With a precision of 28.72% and recall of 40.00%, `Phi-2` demonstrates that architectural advances and increased parameter counts don't guarantee better vulnerability detection.

This is further evidenced by `Mixtral` 8x7B `q5_K_M`, which, despite its advanced architecture and lowest irrelevant response rate (0.36% or 1 out of 280), achieves only 18 true positives, resulting in an F1 score of 20.45%. This suggests that while irrelevant responses affect metrics, the key factor is a model's inherent ability to accurately identify vulnerabilities, as shown by true positive and true negative rates.

Takeaway. Evaluating LLMs for vulnerability detection requires an approach with multiple metrics. Notably, `Gemma` excels in delivering both explicit and accurate responses, with high EP and VEP scores indicating its ability to provide clear vulnerability status and high AP and VAP scores crucial for accurate detection. Open-source models have shown promise, sometimes surpassing proprietary models on specific metrics. However, the varied performance across model variants and versions underscores the importance of thorough evaluation and the ongoing need for research to develop models that perform consistently well across all metrics in vulnerability detection.

I. Customized Models

There have been a lot of efforts to build language models tailored for codes only, such as StructCoder [26], and a natural question is whether such models can be used more effectively to address the problem at hand. While StructCoder [26] presents a novel approach for code generation by incorporating code structure, several limitations make it less suitable for vulnerability detection tasks. Firstly, StructCoder's preprocessing stage relies on generating Data Flow Graph (DFG) information, which may not be feasible for all code snippets in our dataset. Secondly, the dynamic updating of node types during preprocessing can lead to incompatibility issues and incorrect model initialization. Furthermore, our initial experiments found that StructCoder imposes limitations on the code generation process by truncating the input and output based on fixed maximum lengths, which is unsuitable for detection, where the patched code size varies significantly.

StructCoder also requires detailed natural language descriptions for generating code, which may not be available in our vulnerability detection scenario. Lastly, StructCoder is *intended* for code translation tasks rather than code generation, as it does not require an accurate language description for the expected code output. Given these limitations, we conclude that directly applying the StructCoder or other related approaches to our vulnerability detection is infeasible.

J. Limitations

Dataset Limitations. While our evaluation used established benchmarks (Vul4J for Java and Big-Vul for C/C++), these datasets may not fully represent the diversity and complexity of vulnerabilities found in real-world applications. The datasets are also limited in size (280 Java samples and 200 C/C++ samples), which may affect the generalizability of our findings. A limitation of our pipeline is its reliance on pre- and post-patch code as proxies for vulnerable and non-vulnerable samples, assuming post-patch code is non-vulnerable. However, these samples may require future patches for new vulnerabilities, reflecting the evolving nature of vulnerability detection. Moreover, to mitigate data leakage, we implemented two measures: (1) fully offloading the LLM with each user prompt to prevent influence from prior interactions, and (2) acknowledging that while Vul4J and Big-Vul datasets (from 2022 and 2020) were used, we could not verify if these were in the LLMs' training data. Future work could address this by generating new datasets from recent projects, expanding dataset size and vulnerability coverage while verifiably minimizing or precluding the overlap with LLM training data.

Model Selection. Although we evaluated a range of open-source LLMs, our study does not include all models. Some potentially effective models may have been excluded due to resource constraints or availability issues. Additionally, our evaluation focused primarily on models with parameter sizes ranging from 2B to 70B, potentially missing insights from smaller or larger models. Recent models such as `LLaMA-3.1` and `LLaMA-3.2`, and proprietary models like `GPT-4`, `Claude Sonnet 3.5`, and `GPT-4-o` were not evaluated.

Zero-Shot and Few-Shot Limitations. Our zero-shot and few-shot approaches, while demonstrating the models' inherent capabilities, may not represent the optimal way to leverage LLMs for vulnerability detection. The dramatic performance degradation in few-shot learning, particularly for C/C++ code, suggests that our prompt engineering approach may need refinement or that alternative methods might be more effective, including alternative classification frameworks.

Language Coverage. Our study focused on Java and C/C++ code, leaving out many other popular programming languages. The effectiveness of LLMs in detecting vulnerabilities in other languages, particularly those with different paradigms or security models, remains unexplored.

Resource Requirements. The computational resources required for running these models, particularly the larger ones, may limit their practical applicability in real-time vulnerability detection. Our evaluation was conducted on Apple M-series hardware, which may not be representative of all deployments.

Advanced Techniques. Our study did not explore several approaches that could potentially improve vulnerability detection. Implementing Retrieval-Augmented Generation (RAG) could enhance the models' ability to access and utilize relevant vulnerability information. Moreover, using multiple LLM agents collaboratively for vulnerability detection through multi-agent systems could provide more robust and accurate results.

Comparison with Specialized Tools. Our study did not direct compare with specialized vulnerability detection tools, as that falls out of the scope of this work. Future work should compare with such analyzers and commercial scanners to possibly devise hybrid approaches combining LLMs with such tools.