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

Jie Lin and David Mohaisen

University of Central Florida

NDSS Symposium 2025, San Diego, CA

Introduction to the Study

1 Context

Large Language Models show remarkable ability in in code understanding and and generation.

2 Why It Matters

The rise of LLMs is promising progress in vulnerability analysis and detection

3 What's Missing

In-depth insights into how how specific LLM attributes attributes affect detection detection outcomes remain remain underexplored.

Motivation

Traditional Approaches

Relies on static/dynamic analysis analysis and third-party tools, which can be resource-intensive. intensive.

LLMs' Potential

Capable of end-to-end code scanning to determine if a file is vulnerable without external aids.

Need for Breadth

Evaluating multiple architectures architectures across Java and C/C++ reveals strengths, weaknesses, and practical considerations.

Research Questions

1

Detection Efficacy

How accurately do large language models (LLMs) detect vulnerabilities in Java and C/C++ at the file level?

2

Role of Context Window

Does increasing the token limit limit (context window) lead to to more accurate vulnerability vulnerability detection?

3

Quantization Trade-Offs

Does model efficiency (through (through quantization) compromise detection accuracy?

4

Architectural Impact

Do advanced or specialized LLMs outperform earlier earlier versions in identifying vulnerabilities?

5

Few-Shot Learning

Does providing a handful of labeled examples significantly boost detection capability?

General Experiment Workflow

1

Dataset Preparation

Collect, clean, and filter raw Java and C/C++ code.

2

Curated Datasets

Finalize balanced sets sets of vulnerable and and non-vulnerable samples.

3

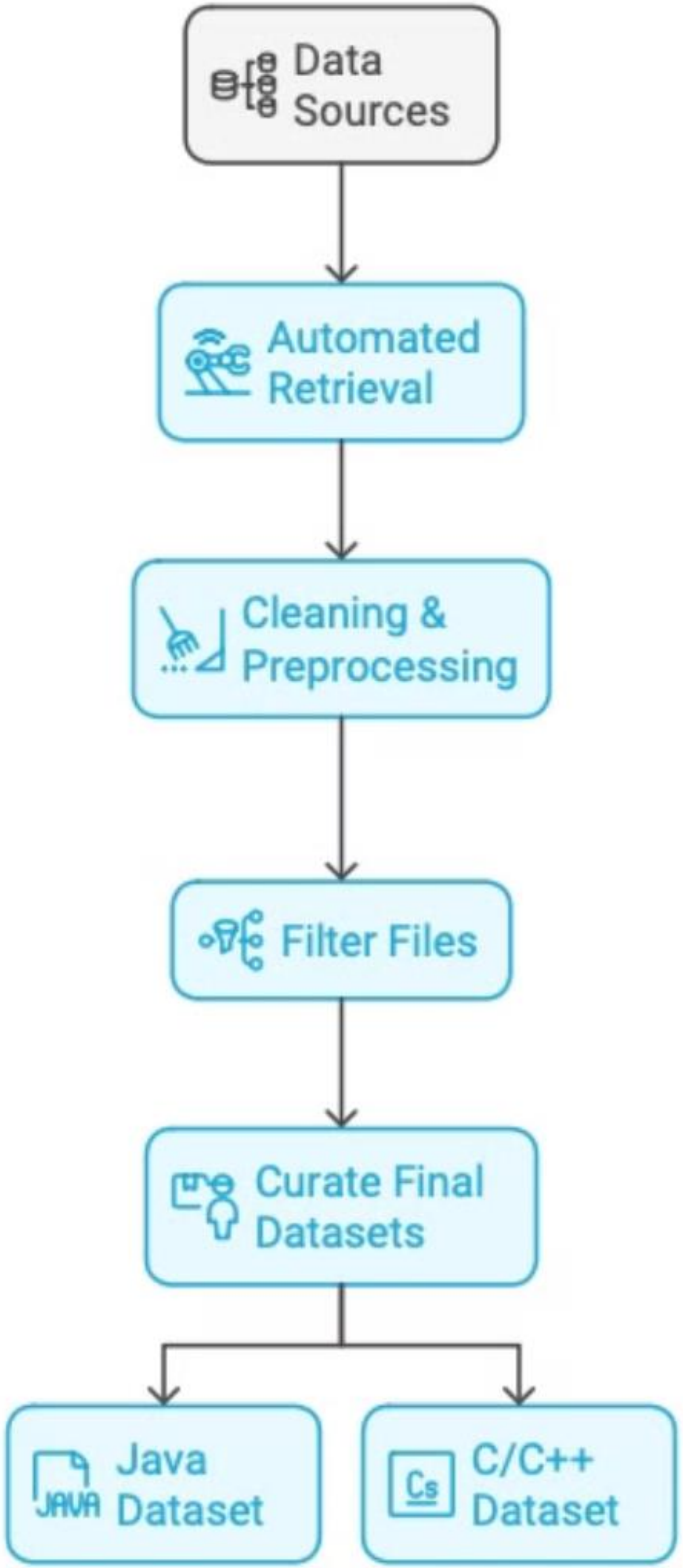
Experimental Pipeline

LLMs process code using using zero-shot, few-shot, shot, and controlled comparison approaches. approaches.

4

Evaluation

Compare predictions predictions against ground truth using custom and standard standard metrics.



Data Curation & Preprocessing

Data Sources

Java: **Vul4J** dataset (1,803 (1,803 fix commits; 51 projects)

C/C++: **Big-Vul** dataset (4,432 commits; 348 projects)

Cleaning Process

- Syntax parsing with Tree-sitter
- Remove comments and whitespace
- Exclude non-code files and incomplete repos

Final Datasets

Java: **280** files (140 vulnerable, 140 non-vulnerable)

C/C++: **200** files (100 vulnerable, 100 non-vulnerable)

Model Selection & Configurations

Model	Parameters	Version	Quantizations	Context Window
LLaMA-2	7B,13B,70B	-	q5_K_M	4096
CodeLLaMA	7B,34B,70B	-	q5_K_M	16384(7B/34B), 2048(70B)
LLaMA-3	8B,70B	-	q5_K_M	8192
Mistral	7B	v0.2	q5_K_M	32768
Mixtral	8Ѵ67B	v0.1	q5_K_M	32768
Gemma	2B,7B	v1.1	q5_K_M & fp16	8192
CodeGemma	7B	v1.1	q5_K_M & fp16	8192
Phi-2	2.7B	v2	q5_K_M & fp16	2048
Phi-3	3.8B*	-	q5_K_M & fp16	4096
GPT-4	-	-	-	-

Experimental Setup (Continued)

Example System Prompt

"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."

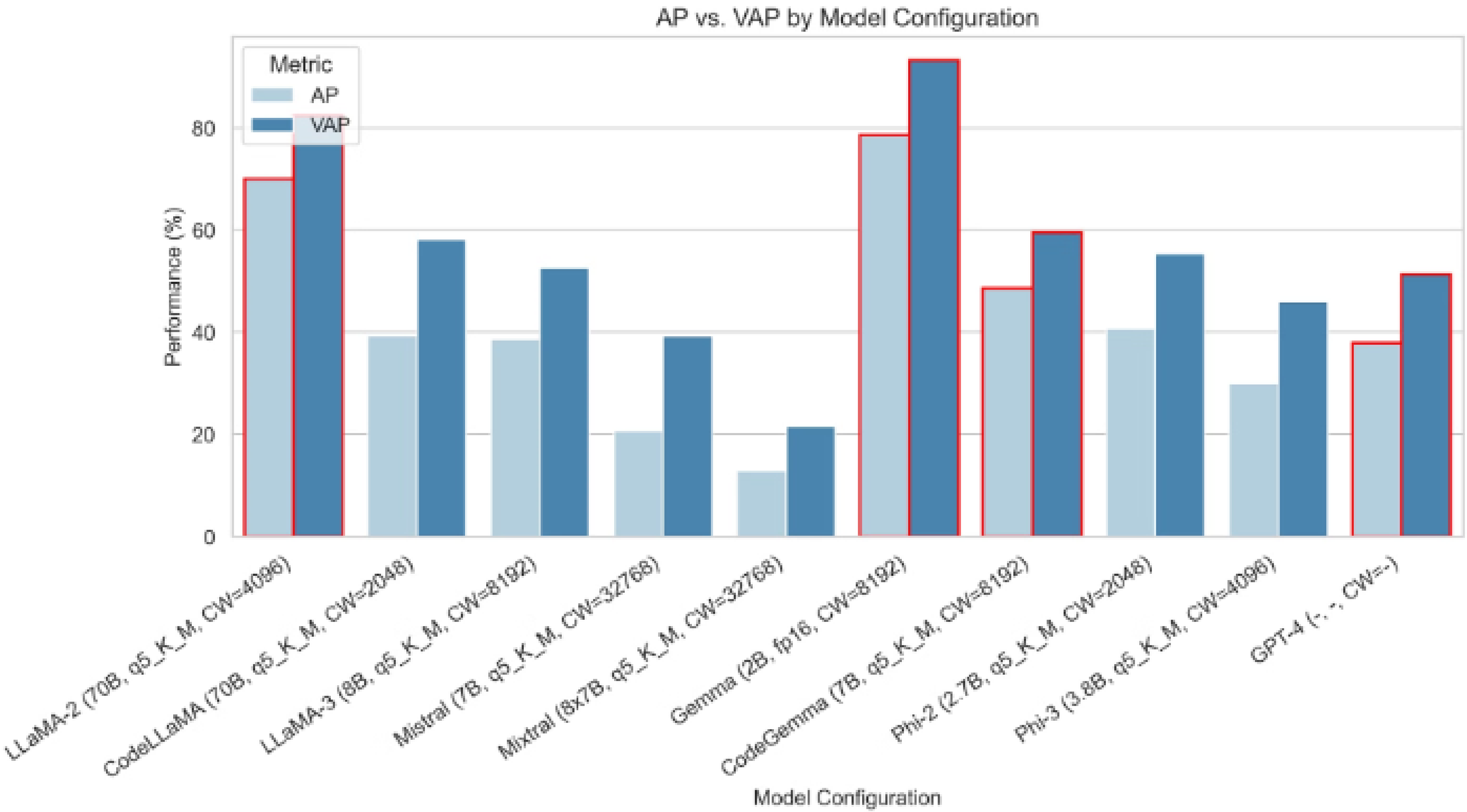
Model Parameters for (open-source)

- Temperature = 0.5
- Fixed Seed = 42
- Output Token Limit = 2048

Controlled Model Comparison (AP/VAP)

Model	Parameters	Quantization	Context Window	AP	VAP
LLaMA-2	70B	q5_K_M	4096	70	82.43
CodeLLaMA	70B	q5_K_M	2048	39.29	58.11
LLaMA-3	8B	q5_K_M	8192	38.57	52.7
Mistral	7B	q5_K_M	32768	20.71	39.19
Mixtral	8x7B	q5_K_M	32768	12.86	21.62
Gemma	2B	fp16	8192	78.57	93.24
CodeGemma	7B	q5_K_M	8192	48.57	59.46
Phi-2	2.7B	q5_K_M	2048	40.71	55.41
Phi-3	3.8B	q5_K_M	4096	30	45.95
GPT-4	-	-	-	37.86	51.35

Controlled Model Comparison (AP/VAP)



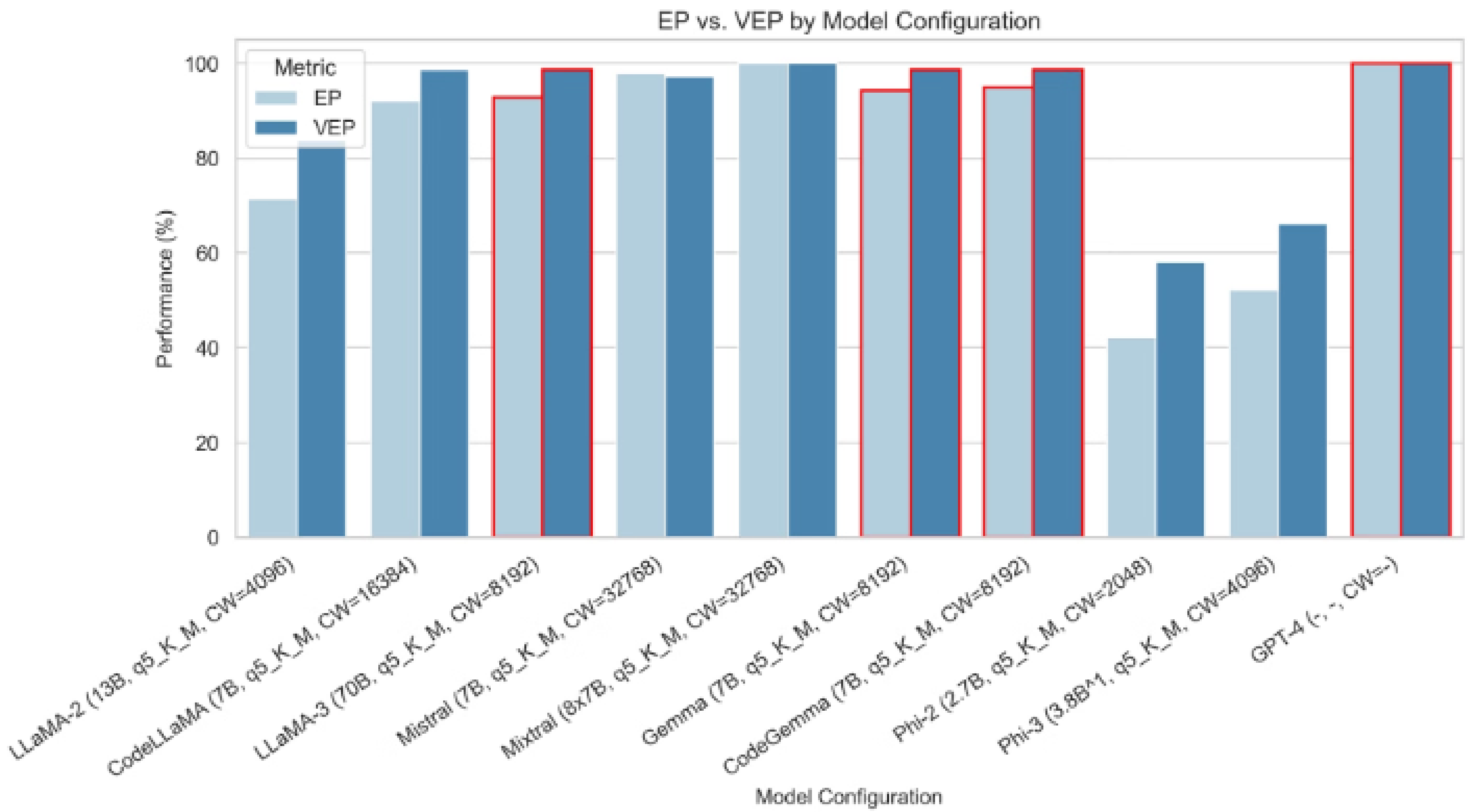
Key Takeaways

- **Top Performers**: Gemma (AP=78.57%, VAP=93.24) and LLaMA-2 (70.00%, 82.43) lead metrics.
- **Surprising Results**: Specialized code models perform below base models.
- **GPT-4**: Ranks lower (AP=37.86%, VAP=51.35) than several open-source models.
- **Impact**: Open-source solutions prove more effective than expected.

Controlled Model Comparison (EP/VEP)

Model	Parameters	Quantization	Context Window	EP	VEP
LLaMA-2	13B	q5_K_M	4096	71.43	83.78
CodeLLaMA	7B	q5_K_M	16384	92.14	98.65
LLaMA-3	70B	q5_K_M	8192	92.86	98.65
Mistral	7B	q5_K_M	32768	97.86	97.3
Mixtral	8x7B	q5_K_M	32768	100	100
Gemma	7B	q5_K_M	8192	94.29	98.65
CodeGemma	7B	q5_K_M	8192	95	98.65
Phi-2	2.7B	q5_K_M	2048	42.14	58.11
Phi-3	3.8B	q5_K_M	4096	52.14	66.22
GPT-4	-	-	-	100	100

Controlled Model Comparison (EP/VEP)



Key Takeaways

- **Top Performers**: GPT-4 and Mixtral achieve perfect EP/VEP scores, with CodeLLaMA and LLaMA-3 close behind.
- **Model Evolution**: Newer variants and domain-specific models consistently outperform their predecessors in explicitness.
- **Trade-offs**: High explicitness (EP/VEP) doesn't always correlate with strong detection accuracy (AP).

Performance Analysis: AP/VAP Results for Positive Sample Testing

Model Size

Larger parameter counts don't always yield higher AP/VAP

Quantization

Effectiveness varies by model architecture

Context Window

Generally, larger CW improves detection, but not guaranteed

Mixed Results

Code-focused training doesn't always outperform general-purpose purpose models

Inconsistent Improvements

Newer versions (e.g., LLaMA-3, Phi-3) don't consistently surpass predecessors

Key Takeaway

Complex interplay between model size, quantization, CW length, and architectural tweaks

Positive and Negative Java Samples Settings for Vulnerability Detection

Extended Evaluation

280 Java files (140 vulnerable +
+ 140 non-vulnerable) tested in
in zero-shot prompt strategy

System Prompt

Consistent with previous tests,
tests, asking for yes/no + concise
concise reasoning

Performance Metrics

Precision, Recall, and F1 score
used to evaluate detection
accuracy

Positive and Negative Java Samples Results

Model	Parameters	Quantization	Context Window	Precision	Recall	F1
LLaMA-3	70B	q5_K_M	8192	23.53	2.86	5.1
Gemma	2B	fp16	8192	44.35	78.57	56.7
Gemma	7B	fp16	8192	46.19	77.86	57.98
CodeGemma	7B	q5_K_M	8192	65.38	48.57	55.74
Phi-3	3.8B	fp16	4096	23.4	23.57	23.49

Positive and Negative Java Samples Analysis

1 Key Observations

Highest precision:

CodeGemma 7B at 65.38%.

Highest recall: Gemma 2B at

78.57%.

2 Parameter Size Impact

Larger models don't always

outperform smaller ones

3 Quantization Effects

Varies by model family; some

some improve with fp16,

others with q5_K_M

4 Architecture Comparisons

Advanced or larger architectures don't guarantee better performance

Positive and Negative C/C++ Samples Settings for Vulnerability Detection

Extended Evaluation

200 C/C++ files (100 vulnerable + vulnerable + 100 non-vulnerable) vulnerable) tested in zero-shot shot prompt strategy

System Prompt

Consistent with previous tests, asking for yes/no + concise reasoning

Performance Metrics

Precision, Recall, and F1 score used to evaluate detection accuracy

Positive and Negative C/C++ Samples Results

Model	Parameters	Quantization	Context Window	Precision	Recall	F1
CodeLLaMA	7B	q5_K_M	16384	28.57	32	30.19
LLaMA-3	70B	q5_K_M	8192	0	0	0
Gemma	7B	q5_K_M	8192	29.29	41	34.17
Gemma	7B	fp16	8192	29.58	42	34.71
Phi-3	3.8B	fp16	4096	4.12	4	4.06

Vulnerability Detection in C/C++ Analysis

1

Best Overall Performance (F1)

Gemma 7B (fp16) at 34.71% (Precision: 29.58%, Recall: 42.00%)

2

Performance Range

Precision peaks around 30%, some models (e.g., Phi-3) struggle with many false positives

3

Quantization Impact

Varies by model family; some some benefit from fp16, others others from q5_K_M

4

Model Size Effects

Larger models don't consistently outperform smaller smaller counterparts

5

Zero Performance Concern

LLaMA-3 70B scores 0.00% in precision, recall, F1, F1, suggesting severe task mismatch

Few-Shot Learning (Java) Experimental Settings

1

Setup

The prompt is enhanced with two example cases, one containing a vulnerability and the other other secure.

2

Model Selection

We chose top-performing performing models from the the zero-shot phase (LLaMA-2 70B, Mistral 7B, 7B, Gemma 7B, Phi-2 2.7B) 2.7B) for this experiment. experiment.

3

Goal

This setup aims to assess assess how a limited number of examples (few- (few-shot) influences vulnerability detection accuracy.

Few-Shot Learning (Java) Results

Model	Parameters	Quantization	Context Window	Precision	Recall	F1
LLaMA-2	70B	q5_K_M	4096	27.66	37.41	31.8
Mistral	7B	q5_K_M	32768	33.33	2.16	4.05
Gemma	7B	fp16	8192	43.24	46.04	44.6
Phi-2	2.7B	q5_K_M	2048	0	0	0

Learning from Examples (C/C++) Results and Analysis

Setup

Four models tested with two example code samples (one vulnerable, one safe)

Model Performance

All models showed 0% accuracy with specific patterns:

- Mistral 7B: Consistently labeled all code "safe"
- CodeLLaMA 7B: Inconsistent errors in both directions
- Gemma 7B: Only 4% safe detection with 46% false alarms
- Phi-2: Merely 1% safe detection detection with 49% false alarms alarms

Key Implications

Poor few-shot learning performance suggests immediate need for:

- Different training methods
- Better example selection

Vulnerability Type Identification (Java) Settings

Why This Matters

Moves beyond simple classification to specific vulnerability types (e.g., SQL injection)

Experimental Setup

Modified prompt to request request CVE ID and short description of each vulnerability

Metrics

AP (Accurate Responses Percentage) and C (Correct Vulnerability Type Count)

Vulnerability Type Identification (Java) Results

Model	Parameters	Quantization	Context Window	Zero-Shot AP (%)	Few-Shot AP (%)
LLaMA-2	70B	q5_K_M	4096	68.57	21.01
CodeLLaMA	7B	q5_K_M	16384	12.86	34.06
LLaMA-3	70B	q5_K_M	8192	4.29	20.29
Gemma	7B	q5_K_M	8192	75.71	37.68
Gemma	7B	fp16	8192	77.86	39.86

Vulnerability Type Identification (Java) Analysis

1

Few-Shot Impact

Many models lose accuracy from zero-shot to few-shot (e.g., Gemma 7B drops from 77.86% to 39.86%)

2

Type Identification

Very limited success; mostly 0 correct type identifications across models

3

Performance Comparison

Simpler "is it vulnerable?" vulnerable?" tasks yielded yielded higher AP than specific type identification identification

Prompt Evaluation Time Analysis

1

Context Window Impact

Larger CW: Slower prompt processing, faster response generation

2

Parameter Size Effect

Bigger models lead to longer longer total processing times times

3

Quantization Benefits

Efficient methods (e.g., q5_K_M) reduce evaluation duration

Addressing Research Questions

RQ1: LLMs for Vulnerability Detection

Yes, but with substantial variability variability across languages and and tasks

RQ2: Context Window Impact

Larger CW generally leads to better context retention and higher accuracy

RQ3: Quantization Effects

Impact is model-dependent; some some improve with fp16, others others with q5_K_M

RQ4: Advanced Architectures

Not consistently better; architectural updates don't don't guarantee improved detection

RQ5: Few-Shot Learning

Counterintuitively, often degraded performance compared to zero-shot approaches

Conclusion

1

Study Overview

Evaluated 38 LLM configurations for vulnerability detection in Java and C/C++

2

Key Takeaways

Architectural gains not guaranteed; context window crucial; few-shot shot scenarios can degrade degrade performance; open-sourced Models can can surpass closed-source source models.

3

Future Directions

Refine architectures, balance context, improve improve prompt engineering, explore more more code domains

Q&A

Jie Lin (ji132432@ucf.edu)

David Mohaisen (mohaisen@ucf.edu)