

Exploring Automated Vulnerability Identification in JavaScript Code Using Large Language Models

Manit Kaushik

manit22277@iiitd.ac.in

Ishir Bhardwaj

ishir22223@iiitd.ac.in

Pranav Gupta

pranav22364@iiitd.ac.in

Dr. Pankaj Jalote

jalote@iiitd.ac.in

Dr. Arun Balaji Buduru

arunb@iiitd.ac.in

Indraprastha Institute of Information Technology Delhi
New Delhi, India

Abstract

JavaScript powers approximately 98.8% of all websites, making vulnerabilities in its code a significant security risk, yet existing detection approaches such as Static Application Security Testing (SAST) tools often fail to identify many real-world vulnerabilities when applied to isolated code snippets. This paper presents an empirical study of Large Language Model (LLM)-based vulnerability identification for JavaScript programs, evaluating three LLM families (Gemini 1.5 Flash, GPT-4o Mini, DeepSeek-R1-Distill-Llama-8B) across multiple prompting strategies (zero-shot, chain-of-thought, few-shot) and fine-tuning approaches on a dataset of 1,125 JavaScript code snippets spanning five Common Weakness Enumeration (CWE) categories: Injection (CWE-74), OS Command Injection (CWE-78), Cross-Site Scripting (CWE-79), SQL Injection (CWE-89), and Uncontrolled Resource Consumption (CWE-400). Our experiments show that LLMs substantially outperform traditional SAST tools on snippet-level vulnerability identification, with a fine-tuned Gemini 1.5 Flash model achieving 60% detection accuracy compared to near-zero performance from rule-based analyzers. We find that fine-tuning improves accuracy from 29% to 60%, Chain-of-Thought prompting benefits reasoning-capable models such as GPT-4o Mini, few-shot prompting is effective for polymorphic vulnerabilities such as Cross-Site Scripting, and performance varies across vulnerability categories, reaching up to 84% accuracy for structured vulnerabilities such as SQL Injection. These results indicate that LLMs provide a practical approach for automated vulnerability identification in JavaScript code, particularly when combined with task-aligned supervision, though they should complement rather than replace existing security analysis workflows due to limited recall and uneven performance across vulnerability types.

1 Introduction

1.1 Motivation and Background

Vulnerability detection is a core component of software security, enabling the identification of weaknesses before they are exploited in production systems. The Common Weakness Enumeration (CWE) framework provides a standardized taxonomy for classifying software vulnerabilities and supports consistent analysis across programming languages and platforms MITRE Corporation [2024a]. Traditional detection approaches rely primarily on Static Application Security Testing (SAST), which uses pattern matching and rule-based analysis to identify known vulnerability signatures in source code. While SAST tools have been widely adopted in development workflows, they face practical limitations when analyzing isolated code snippets: rule-based analyzers typically require full project context to trace data flow from sources to sinks, and their precision and coverage degrade significantly when applied to incomplete code fragments OWASP Foundation [2024], Kluban et al. [2022], Brito et al., Keltek et al. [2024], Mateo Tudela et al. [2020]. Recent advances in artificial intelligence have introduced Large Language Models (LLMs) as a potential alternative for vulnerability detection, due to their ability to reason over source code and capture semantic patterns without requiring explicit control-flow or data-flow graphs Sheng et al. [2025], Zhang et al. [2024]. However, JavaScript remains comparatively underexamined in this setting, even though it powers approximately 98.8% of websites and is frequently associated with security-critical behavior such as user input handling, authentication, and DOM manipulation Kluban et al. [2022], Nikitopoulos et al. [2021], Zenodo Repository [2024]. This work focuses exclusively on automated vulnerability identification in JavaScript using LLMs, constructing a dataset of real-world JavaScript snippets spanning multiple CWE categories and evaluating three LLM families (Gemini 1.5 Flash, GPT-4o Mini, DeepSeek-R1-Distill-Llama-8B) under different prompting and fine-tuning settings, with model performance benchmarked against widely used

SAST tools.

1.2 Research Questions

We study the following research questions to assess the viability of LLM-based vulnerability identification in JavaScript:

- **RQ1:** How do LLMs perform in identifying JavaScript vulnerabilities compared to SAST tools?
- **RQ2:** How do prompting strategies affect detection performance across CWE categories?
- **RQ3:** What is the effect of fine-tuning on vulnerability identification accuracy?
- **RQ4:** How does performance vary across different CWE categories?

1.3 Paper Organization

The remainder of this paper is organized as follows. Section 2 discusses related work on LLM-based vulnerability detection and positions our contribution within the broader research landscape. Section 3 describes our experimental design and methodology, including dataset construction, model selection, prompting strategies, and fine-tuning procedures. Section 4 presents our experimental results organized by research question. Section 5 discusses threats to validity and limitations of our approach. Section 6 concludes with a summary of findings and directions for future work.

2 Related Work

Recent work has examined the use of Large Language Models for software vulnerability detection, with a focus on prompt design, reasoning, and model adaptation Sheng et al. [2025]. Gao et al. [2023] report that GPT-4 outperforms traditional deep learning models and static analyzers on Capture The Flag (CTF)-style datasets, publicly available challenge sets where participants identify intentionally embedded vulnerabilities, although performance decreases on more complex software projects. Kostina et al. [2025] find that LLMs outperform classical machine learning approaches across complex binary and multiclass classification tasks in vulnerability detection.

Several studies explore structured prompting strategies to improve detection accuracy and explanation quality. Ullah et al. [2023] apply few-shot prompting and Chain-of-Thought reasoning for C/C++ vulnerabilities, while Nong et al. [2024] introduce vulnerability semantics-guided prompting, showing improvements in precision and explanation quality for Java and C++ code.

Despite these advances, prior work has limited support for JavaScript. Sheng et al. [2025] identify weak cross-language generalization and scarce JavaScript evaluation as open gaps in current research. Existing datasets and experiments primarily emphasize C, C++, Java, and Python,

with fewer studies examining JavaScript’s dynamic execution model and frontend-oriented vulnerability patterns.

Our work contributes to this line of research by focusing specifically on *automated vulnerability identification in JavaScript*. We construct a dataset of real-world JavaScript snippets and evaluate LLMs across prompting strategies, fine-tuning settings, and vulnerability categories, benchmarking against SAST tools rather than repair-oriented workflows.

3 Experimental Design and Methodology

3.1 Dataset

We curated a dataset of vulnerable JavaScript code snippets, each annotated with a Common Weakness Enumeration (CWE) identifier, from three publicly available sources: Kluban et al.’s real-world JavaScript vulnerabilities Kluban et al. [2022], CrossVul’s cross-language vulnerability repository with commit-level documentation Nikitopoulos et al. [2021], and Zenodo’s function-level vulnerability datasets Zenodo Repository [2024]. We selected these three datasets because they provide real-world JavaScript vulnerabilities with verified CWE labels, commit histories that link vulnerable and patched code, and sufficient coverage across the five most frequent CWE categories in JavaScript. The number of datasets is limited to three because few publicly available JavaScript vulnerability datasets meet the requirements of verified CWE annotations, code-level granularity, and sufficient samples per vulnerability type. We selected five CWE categories based on their frequency in the source datasets and their relevance to JavaScript web applications: CWE-74 (Injection) MITRE Corporation [2024c], CWE-78 (OS Command Injection) MITRE Corporation [2024d], CWE-79 (Cross-Site Scripting) MITRE Corporation [2024e], CWE-89 (SQL Injection) MITRE Corporation [2024f], and CWE-400 (Uncontrolled Resource Consumption) MITRE Corporation [2024b]. The final dataset contains 1,000 training samples and 125 test samples, with a balanced distribution of 200 training samples and 25 test samples per CWE category.

3.2 Problem Formulation

We treat automated vulnerability identification as a classification task over JavaScript code snippets. In the multi-class setting, the model predicts one of five CWE categories for a given snippet. In the single-class setting, the model determines whether a snippet contains a specific CWE type (binary Yes/No classification). This formulation allows us to evaluate both general vulnerability detection capability and category-specific identification performance.

3.3 Prompting Strategies

We evaluate three prompting strategies to assess how task framing affects LLM performance on vulnerability identifica-

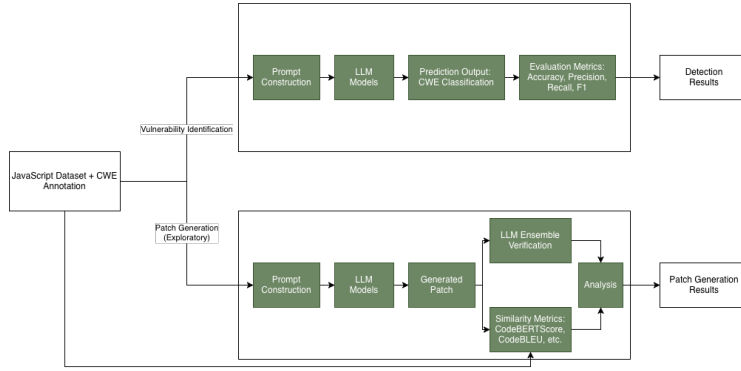


Figure 1: Experiment pipeline for vulnerability identification and patch generation.

tion. Zero-shot prompting evaluates how well an LLM can identify vulnerabilities without examples or task-specific conditioning, serving as a baseline that reveals which vulnerability patterns are already captured in pretraining. For multi-class identification, each snippet contains exactly one of the five CWEs, and the model outputs only the CWE identifier; for single-class identification, the model responds Yes or No for a target CWE. Chain-of-Thought (CoT) prompting introduces short reasoning steps before classification Wei et al. [2023], guiding the model to (1) locate potential input or attack surface, (2) assess whether the input is exploitable, and (3) map the behavior to a CWE, which allows us to study whether explicit reasoning improves identification accuracy across models, consistent with prior findings in vulnerability analysis Nong et al. [2024]. Few-shot prompting is applied only in the single-class setting, where each prompt includes one positive and one negative example retrieved using embedding similarity, allowing us to test whether in-context examples help models recognize polymorphic vulnerability patterns such as Cross-Site Scripting. We apply few-shot prompting only in the single-class setting because it requires category-specific examples, making it impractical for multi-class classification where the model must distinguish among all five CWE categories simultaneously.

3.4 LLMs Explored

We evaluate three LLM families to compare performance across commercial and open-source ecosystems, reasoning-oriented versus lightweight architectures, and cloud versus local execution contexts. GPT-4o Mini was selected as a commercially deployed model with strong reasoning ability at moderate cost, fine-tuned via the OpenAI Dashboard with a learning rate multiplier of 0.02 and Adam-based optimization OpenAI [2024]. Gemini 1.5 Flash was chosen to represent a lightweight commercial architecture optimized for fast inference, fine-tuned via Google AI Studio with a learning rate multiplier of 0.002 and batch size 16 gem [2024]. DeepSeek-R1-Distill-Llama-8B was included as an open-source, locally deployable alternative to support reproducibility and privacy-sensitive settings, fine-tuned using Low-Rank Adaptation (LoRA) via the Unsloth framework Unsloth AI [2024], Hu

et al. [2021] with rank 16 adapters on attention and MLP layers, 4-bit model loading, gradient checkpointing, mixed precision training, and AdamW 8-bit optimization dee [2025]. Models are fine-tuned on 1,000 labeled samples to align them with the vulnerability identification task, with standard fine-tuning using code-CWE pairs and CoT fine-tuning additionally including short reasoning traces; all models are trained for 10 epochs with batch size 16 and linear learning-rate scheduling.

3.5 SAST Baselines

We benchmark LLM performance against two JavaScript-oriented SAST tools chosen to reflect commonly used rule-based detection pipelines. Bearer is a security scanning platform with CWE mapping and prior use in LLM-assisted workflows, making it a suitable baseline for comparison with AI-based methods Keltek et al. [2024]. NodeJSScan is a JavaScript/Node.js static analyzer that supports snippet-level scanning and direct CWE outputs, enabling fair comparison with our evaluation setup Brito et al.. Both tools operate without full-project context, aligning with our snippet-level identification task.

4 Results

We present experimental results organized by research question, comparing LLM-based vulnerability identification against SAST tools and evaluating the effects of prompting strategies, fine-tuning, and vulnerability category on detection performance. Performance is measured using accuracy, precision, recall, and F1-score computed over the 125-sample test set, with separate evaluations for multi-class and single-class identification tasks.

4.1 RQ1: LLMs vs. SAST Tools

Bearer and NodeJSScan achieve near-zero performance on snippet-level evaluation, confirming that rule-based SAST tools designed for whole-project analysis cannot effectively identify vulnerabilities in isolated code fragments Brito et al.,

Table 1: Performance Comparison: SAST Tools vs. Vanilla LLMs

Method	Type	Acc.	Prec.	Rec.	F1
Bearer	SAST	0.000	0.000	0.000	0.000
NodeJSScan	SAST	0.008	0.200	0.008	0.020
DeepSeek-R1-8B	LLM	0.080	0.260	0.080	0.120
Gemini 1.5 Flash	LLM	0.290	0.400	0.290	0.260
ChatGPT 4o-mini	LLM	0.340	0.430	0.340	0.310

Kelte et al. [2024]. This reflects a fundamental architectural limitation: SAST tools trace vulnerability propagation through program flows, requiring project structure that snippets lack. In contrast, LLMs achieve measurable accuracy by reasoning over local semantic patterns. ChatGPT 4o-mini reaches 34% accuracy (0.31 F1), successfully identifying roughly one-third of vulnerabilities without task-specific training, validating semantic understanding from pretraining. Gemini 1.5 Flash achieves 29% (0.26 F1), while DeepSeek-R1-8B lags at 8%. The substantial gap between LLMs and SAST (34% vs. near 0%) directly addresses RQ1, showing LLMs provide a viable alternative for vulnerability detection where traditional static analysis fails, such as code review, security audits, or educational contexts. However, modest recall indicates LLMs miss many vulnerabilities, limiting standalone use.

Finding (RQ1)

LLMs clearly outperform the SAST tools on snippet-level JavaScript vulnerability identification, but their recall is still limited, leaving many vulnerabilities undetected.

These results reveal architectural trade-offs in vulnerability detection. The SAST-snippet mismatch suggests context-dependent strategies: whole-program analysis for production codebases, LLM approaches where SAST cannot operate. The 34% baseline indicates LLMs internalized security patterns during pretraining from security discussions and code reviews in training corpora, though the accuracy-recall gap reveals high false negatives, as LLMs identify vulnerabilities correctly when detected but miss many instances.

4.2 RQ2: Effect of Prompting Strategies

Table 2: Effect of Prompting Strategies on Vanilla LLMs

Model	Strategy	Acc.	Prec.	Rec.	F1
Gemini 1.5 Flash	Zero-Shot	0.29	0.40	0.29	0.26
Gemini 1.5 Flash	CoT	0.18	0.43	0.18	0.21
ChatGPT 4o-mini	Zero-Shot	0.34	0.43	0.34	0.31
ChatGPT 4o-mini	CoT	0.35	0.47	0.35	0.32

Chain-of-Thought prompting effects vary by model architecture, revealing how models process explicit reasoning

(RQ2). For ChatGPT 4o-mini, lightweight reasoning guidance produces small gains: accuracy improves from 34% to 35% and F1 from 0.31 to 0.32, indicating structured thought processes help disambiguate similar vulnerability patterns by decomposing identification into subtasks (locating inputs, assessing exploitability, mapping to CWE). Gemini 1.5 Flash shows reduced accuracy (29% to 18%) despite increased precision (0.40 to 0.43), suggesting its lightweight architecture and preference for concise outputs conflict with verbose reasoning prompts, favoring precision over coverage. These results demonstrate prompting strategies must align with model capabilities: CoT benefits models trained for extended reasoning but may hinder efficiency-optimized architectures. Practitioners should validate prompting approaches per model rather than assuming universal applicability.

Finding (RQ2)

CoT prompting yields small but consistent gains for ChatGPT 4o-mini, while it reduces accuracy for Gemini 1.5 Flash, indicating that the benefit of explicit reasoning is model-dependent.

4.3 RQ3: Effect of Fine-Tuning

Table 3: Vanilla vs. Fine-Tuned LLMs (Macro Accuracy and F1)

LLM	Vanilla		Standard FT		CoT FT	
	Acc.	F1	Acc.	F1	Acc.	F1
DeepSeek-R1-8B	0.08	0.12	N/A	N/A	0.27	0.27
Gemini 1.5 Flash	0.29	0.26	0.60	0.60	0.22	0.21
ChatGPT 4o-mini	0.34	0.31	0.49	0.48	0.30	0.31

Standard supervised fine-tuning demonstrates substantial improvements, directly answering RQ3 by showing task-aligned supervision addresses critical limitations in pretrained models. Gemini 1.5 Flash exhibits the largest gains, jumping from 29% to 60% accuracy (0.26 to 0.60 F1), effectively doubling identification capability. This indicates that while pretraining captures some vulnerability patterns, explicit supervision on CWE-labeled JavaScript snippets is necessary to reliably map code features to vulnerability types. ChatGPT 4o-mini increases from 34% to 49% (0.31 to 0.48 F1), with more modest gains likely due to stronger pretrained reasoning. DeepSeek-R1-8B, fine-tuned with CoT supervision, improves from 8% to 27%, demonstrating reasoning-oriented training aligns with its architecture. However, CoT fine-tuning underperforms standard fine-tuning for Gemini and ChatGPT, suggesting direct label supervision is more sample-efficient than supervising intermediate reasoning steps, practitioners should prioritize simple input-output pairs for classification tasks.

Finding (RQ3)

Standard fine-tuning on labeled JavaScript vulnerabilities yields large gains (Gemini: 29% to 60% accuracy). CoT-style fine-tuning helps DeepSeek but does not improve Gemini or ChatGPT beyond standard fine-tuning.

Fine-tuning results reveal that pretrained models possess latent security knowledge requiring task-specific supervision to surface reliably. Gemini’s 31-point improvement shows general language understanding does not automatically translate to security classification, explicit alignment through supervised examples bridges this gap. Standard fine-tuning’s superiority over CoT fine-tuning suggests direct input-output mappings are more efficient for classification tasks.

4.4 RQ4: CWE-wise Performance and Few-Shot Effects

Performance across vulnerability categories reveals patterns addressing RQ4 and insights into what LLMs learn about security. SQL Injection (CWE-89) emerges as easiest, with ChatGPT 4o-mini and Gemini 1.5 Flash achieving 82–84% accuracy zero-shot, approaching practical utility. This reflects SQL injection’s structured nature: vulnerable patterns involve direct string concatenation into queries (`query = "SELECT * FROM users WHERE id=" + userId`), a signature frequent in security documentation and pretraining corpora. OS Command Injection (CWE-78) shows similar performance (78–80%), as shell command construction follows predictable patterns, indicating LLMs internalize syntactic templates for common injection vulnerabilities. Cross-Site Scripting (CWE-79) and generic Injection (CWE-74) exhibit larger variability. For CWE-79, ChatGPT 4o-mini jumps from 46% (zero-shot) to 76–78% (few-shot), a 30+ point improvement demonstrating value of in-context examples for polymorphic vulnerabilities. XSS patterns vary by context (innerHTML vs. document.write vs. attribute injection), and few-shot examples help recognize variations by anchoring to concrete instances. For CWE-400, few-shot improves ChatGPT and DeepSeek, though Gemini shows inconsistent behavior, suggesting lightweight architectures struggle with less-structured types.

Model-specific few-shot effects reveal architectural trade-offs: DeepSeek’s few-shot performance declines in several categories, likely because its internal reasoning conflicts with externally provided examples. Gemini and ChatGPT, optimized for in-context learning, typically gain from few-shot for variable vulnerability types where examples reduce ambiguity, suggesting few-shot is most valuable for models designed for in-context adaptation.

Finding (RQ4)

LLMs detect structured vulnerabilities such as SQL Injection (CWE-89) and OS Command Injection (CWE-78) well in zero-shot, while few-shot prompting is most useful for polymorphic categories like XSS (CWE-79). The benefit of few-shot varies by model and CWE.

5 Discussion

Our evaluation demonstrates the potential of LLM-based vulnerability identification in JavaScript while highlighting important methodological considerations. We discuss key limitations and their implications for practical deployment.

5.1 Evaluation Context

Our snippet-level evaluation setting favors LLMs over traditional SAST tools, which are typically optimized for whole-project analysis with interprocedural data-flow tracking. The near-zero SAST performance on isolated snippets reflects architectural differences rather than fundamental tool inadequacy, as LLMs reason over local patterns without explicit flow graphs, while rule-based analyzers require richer structural context. This suggests complementary rather than competitive roles in security workflows.

5.2 Generalization and Dataset Scope

Ground-truth labels are derived from publicly documented vulnerability–fix records with explicit CWE annotations. This study focuses on JavaScript and five common CWE categories; results may not generalize to other languages with different type systems or execution models, or to vulnerability types requiring complex interprocedural reasoning. We use consistent fine-tuning schedules across models to ensure fair comparison rather than optimizing hyperparameters individually, which may affect absolute performance values.

5.3 Practical Implications

Our results show moderate recall (34–60%), indicating that many vulnerabilities remain undetected. This suggests LLMs are best deployed as augmentations to existing workflows, serving as rapid filters for incomplete code or exploratory contexts, rather than replacements for comprehensive static analysis. Security-critical applications should continue to employ multiple complementary detection methods.

6 Conclusion and Future Work

This paper investigated automated vulnerability identification in JavaScript using Large Language Models, evaluating three model families under diverse prompting and fine-tuning configurations against SAST baselines. Our study yields several

Table 4: Vanilla vs. Fine-Tuned LLMs Across CWE Categories and Prompting Methods

S. No.	LLM	Method	Accuracy		F1-Score	
			Vanilla	Fine-Tuned	Vanilla	Fine-Tuned
CWE 74						
1	DeepSeek-R1-Distill-Llama-8B	Zero Shot	0.42	0.53	0.48	0.59
2	DeepSeek-R1-Distill-Llama-8B	Few Shot	0.53	0.53	0.57	0.56
3	Gemini 1.5 Flash	Zero Shot	0.68	0.53	0.67	0.57
4	Gemini 1.5 Flash	Few Shot	0.71	0.58	0.68	0.62
5	ChatGPT 4o-mini	Zero Shot	0.51	0.66	0.50	0.65
6	ChatGPT 4o-mini	Few Shot	0.71	0.78	0.70	0.73
CWE 78						
7	DeepSeek-R1-Distill-Llama-8B	Zero Shot	0.51	0.57	0.57	0.61
8	DeepSeek-R1-Distill-Llama-8B	Few Shot	0.49	0.41	0.56	0.50
9	Gemini 1.5 Flash	Zero Shot	0.79	0.78	0.72	0.71
10	Gemini 1.5 Flash	Few Shot	0.79	0.80	0.72	0.76
11	ChatGPT 4o-mini	Zero Shot	0.77	0.78	0.73	0.74
12	ChatGPT 4o-mini	Few Shot	0.78	0.80	0.73	0.73
CWE 79						
13	DeepSeek-R1-Distill-Llama-8B	Zero Shot	0.42	0.56	0.49	0.61
14	DeepSeek-R1-Distill-Llama-8B	Few Shot	0.52	0.40	0.58	0.49
15	Gemini 1.5 Flash	Zero Shot	0.73	0.74	0.72	0.76
16	Gemini 1.5 Flash	Few Shot	0.76	0.74	0.70	0.74
17	ChatGPT 4o-mini	Zero Shot	0.46	0.58	0.46	0.71
18	ChatGPT 4o-mini	Few Shot	0.76	0.78	0.70	0.70
CWE 89						
19	DeepSeek-R1-Distill-Llama-8B	Zero Shot	0.67	0.59	0.70	0.63
20	DeepSeek-R1-Distill-Llama-8B	Few Shot	0.62	0.50	0.65	0.56
21	Gemini 1.5 Flash	Zero Shot	0.82	0.78	0.77	0.74
22	Gemini 1.5 Flash	Few Shot	0.83	0.82	0.78	0.80
23	ChatGPT 4o-mini	Zero Shot	0.84	0.84	0.79	0.79
24	ChatGPT 4o-mini	Few Shot	0.85	0.83	0.81	0.78
CWE 400						
25	DeepSeek-R1-Distill-Llama-8B	Zero Shot	0.53	0.56	0.58	0.60
26	DeepSeek-R1-Distill-Llama-8B	Few Shot	0.48	0.60	0.53	0.65
27	Gemini 1.5 Flash	Zero Shot	0.74	0.48	0.70	0.53
28	Gemini 1.5 Flash	Few Shot	0.80	0.66	0.81	0.73
29	ChatGPT 4o-mini	Zero Shot	0.53	0.66	0.58	0.74
30	ChatGPT 4o-mini	Few Shot	0.75	0.79	0.71	0.71

contributions. First, LLMs substantially outperform SAST tools in snippet-level identification, achieving 34% accuracy without fine-tuning versus near-zero SAST performance, establishing LLMs as viable alternatives where rule-based analyzers fail due to missing project context. Second, supervised fine-tuning produces dramatic improvements (29% to 60% for Gemini), revealing pretrained models contain latent security knowledge requiring task-specific alignment to surface reliably. This 31-percentage-point gain demonstrates that general language understanding does not automatically translate to security classification without explicit supervision. Third, we identify model-dependent prompting effects: Chain-of-Thought benefits reasoning-oriented models like ChatGPT but hinders efficiency-optimized architectures like Gemini, while few-shot prompting helps recognize polymorphic vulnerabilities like XSS by anchoring reasoning to concrete examples. Fourth, CWE-level analysis reveals LLMs excel at structured injection patterns (82–84% accuracy for SQL injection) but struggle with context-dependent vulnerabilities, suggesting pretraining corpora emphasize syntactic vulnerability signatures over semantic security properties. These findings establish that LLMs offer practical value for JavaScript security in exploratory contexts, such as code review, educational settings, and rapid prototyping, where traditional SAST can-

not operate effectively, though limited recall (34–60%) necessitates integration with existing workflows rather than replacement. The substantial performance gap between vanilla and fine-tuned models indicates domain adaptation is critical for security applications, while CWE-level variance (46–84% accuracy) suggests practitioners should validate LLM performance on vulnerability types relevant to their specific use cases rather than assuming uniform effectiveness.

Research directions emerge from our findings. Project-level evaluation would assess whether LLMs can leverage inter-procedural control-flow and data-flow information beyond local pattern matching, potentially narrowing the performance gap with SAST tools in their intended whole-program analysis settings. Expanding dataset coverage to additional vulnerability classes, authorization errors, cryptographic misuse, race conditions, would reveal whether our findings generalize beyond injection-style weaknesses or whether LLMs require distinct training strategies for different vulnerability categories. Investigating hybrid approaches that combine LLM semantic reasoning with static analysis taint-tracking may improve recall while preserving precision, leveraging complementary strengths of neural and symbolic methods. Finally, longitudinal studies tracking how LLM security capabilities evolve across model generations would clarify whether vulner-

ability detection improves naturally through scale and diverse pretraining, or whether it requires dedicated security-oriented training data and objectives.

References

- Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024. URL <https://arxiv.org/abs/2403.05530>.
- Deepseek-rl: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL <https://arxiv.org/abs/2501.12948>.
- Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Frago Santos, and Nuno Santos. Study of javascript static analysis tools for vulnerability detection in node.js packages. *IEEE Transactions on Reliability*, 72(4). ISSN 1558-1721. doi: 10.1109/tr.2023.3286301. URL <http://dx.doi.org/10.1109/TR.2023.3286301>.
- Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. How far have we gone in vulnerability detection using large language models, 2023. URL <https://arxiv.org/abs/2311.12420>.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL <https://arxiv.org/abs/2106.09685>.
- Mete Keltek, Rong Hu, Mohammadreza Fani Sani, and Ziyue Li. Boosting cybersecurity vulnerability scanning based on llm-supported static application security testing, 2024. URL <https://arxiv.org/abs/2409.15735>.
- Maryna Kluban, Mohammad Mannan, and Amr Youssef. On measuring vulnerable javascript functions in the wild. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022. URL <https://dl.acm.org/doi/10.1145/3488932.3497769>.
- Arina Kostina, Marios D. Dikaiakos, Dimosthenis Stefanidis, and George Pallis. Large language models for text classification: Case study and comprehensive review, 2025. URL <https://arxiv.org/abs/2501.08457>.
- Francesc Mateo Tudela, Juan-Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan-Antonio Sicilia Montalvo, and Michael I. Argyros. On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications. *Applied Sciences*, 10(24), 2020. ISSN 2076-3417. doi: 10.3390/app10249119. URL <https://www.mdpi.com/2076-3417/10/24/9119>.
- MITRE Corporation. Common weakness enumeration. <https://cwe.mitre.org/>, 2024a. Accessed: 2024.
- MITRE Corporation. Cwe-400: Uncontrolled resource consumption. <https://cwe.mitre.org/data/definitions/400.html>, 2024b. Accessed 2024.
- MITRE Corporation. Cwe-74: Improper neutralization of special elements in output used by a downstream component (injection). <https://cwe.mitre.org/data/definitions/74.html>, 2024c. Accessed 2024.
- MITRE Corporation. Cwe-78: Improper neutralization of special elements used in an os command (os command injection). <https://cwe.mitre.org/data/definitions/78.html>, 2024d. Accessed 2024.
- MITRE Corporation. Cwe-79: Improper neutralization of input during web page generation (cross-site scripting). <https://cwe.mitre.org/data/definitions/79.html>, 2024e. Accessed 2024.
- MITRE Corporation. Cwe-89: Improper neutralization of special elements used in an sql command (sql injection). <https://cwe.mitre.org/data/definitions/89.html>, 2024f. Accessed 2024.
- Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. Crossvul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1565–1569, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3473122. URL <https://doi.org/10.1145/3468264.3473122>.
- Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities, 2024. URL <https://arxiv.org/abs/2402.17230>.
- OpenAI. Gpt-4o: A multimodal ai model. <https://openai.com/index/hello-gpt-4o/>, 2024. Accessed 2024.
- OWASP Foundation. Static code analysis. https://owasp.org/www-community/controls/Static_Code_Analysis, 2024. Accessed 2024.
- Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. Llms in software security: A survey of vulnerability detection techniques and insights, 2025. URL <https://arxiv.org/abs/2502.07049>.
- Saad Ullah, Ayse Coskun, Alessandro Morari, Saurabh Pujar, and Gianluca Stringhini. Step-by-step vulnerability detection using large language models. Poster, USENIX Security Symposium, 2023. URL https://www.bu.edu/peaclab/files/2023/08/USENIX_23_Poster.pdf. Boston University and IBM Research.

Unsloth AI. Unsloth: Fast, memory-efficient fine-tuning of llama models. <https://github.com/unslothai/unsloth>, 2024. Accessed 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL <https://arxiv.org/abs/2201.11903>.

Zenodo Repository. Software vulnerability detection datasets. <https://zenodo.org/records/13870382>, 2024. Dataset. Accessed 2024.

Jie Zhang, Haoyu Bu, Hui Wen, Yongji Liu, Haiqiang Fei, Rongrong Xi, Lun Li, Yun Yang, Hongsong Zhu, and Dan Meng. When llms meet cybersecurity: A systematic literature review, 2024. URL <https://arxiv.org/abs/2405.03644>.