Minimizing False Positives in CodeQL Taint Analysis Through LLM-Driven Contextual Reasoning

Nikhil Patil, Kelly Orjiude

***Abstract*—Static Application Security Testing (SAST) tools, such as CodeQL, are fundamental for identifying vulnerabilities at scale, yet they suffer from a significant credibility problem due to high False Positive Rates (FPR) [1] . This over-approximation leads to Alert Fatigue among security analysts, increasing the cost of manual triage [1]. The foundational UntrustIDE study demonstrated this challenge empirically, finding an effective FPR of approximately 98% in CodeQL's taint analysis of VS Code extensions [1]. We propose a novel, scalable, three-stage LLM Triage Pipeline that automates contextual reasoning to filter noisy CodeQL alerts [1]. By transforming the purely structural taint flow data produced by CodeQL into a contextual reasoning task for a Large Language Model (LLM), we aim to reduce the baseline FPR of > 98% to an industrially acceptable threshold of < 20% [1]. Our initial empirical validation on a sampled dataset demonstrates the LLM's capability for path-aware reasoning and contextual understanding, showcasing a promising path toward solving the most persistent precision problem of static analysis [1].**

***Index Terms*—CodeQL, static taint analysis, large language model, artificial intelligence, VS code, sources, sinks, taint flows.**

I. Introduction and Problem Definition

A. The Cost of Noise in Software Security

Software security relies heavily on static analysis to detect vulnerabilities. However, Static Taint Analysis (STA), like that performed by CodeQL, is inherently prone to over-approximation [1], [2]. It flags possible data flows rather than precisely identifying exploitable ones, creating an excessive volume of non-vulnerable findings [1]. The subsequent manual effort required to distinguish true positives (TPs) from false positives (FPs) is prohibitively expensive, leading to Alert Fatigue and significantly reducing the practical value and credibility of the tool [1].

B. The UntrustIDE Baseline

Our project is anchored by the methodology and findings of the **UntrustIDE** study which provides a large-scale, real-world baseline for CodeQL performance [1]. The study analyzed over 43,000 VS Code extensions and focused on flows from common untrusted sources (e.g., workspace settings, file reads, web servers) to critical sinks (e.g., shell commands, eval(), file writes).

A key result of the baseline study was the massive ratio of noise to signal. Specifically, of the 389 potential flows identified from a Workspace Setting source to a Shell Command sink, only 27 were manually investigated, resulting in a mere seven confirmed real vulnerable data flows [1]. This yielded an effective FPR of > 98% [1]. These data clearly illustrate the **critical gap** that exists between the structural possibility of a vulnerability reported by the static analysis engine and the contextual exploitability confirmed by a human auditor [1].

Our **Goal** is to automate this contextual sanity check to reduce the FPR from > 98% to a practical threshold, such as < 20% [1].

II. Context and Background: Bridging the Credibility Gap

A. The Limitations of Static Taint Analysis (CodeQL)

CodeQL functions as a semantic code analysis engine, converting source code into an extensible database for querying via the QL language [1]. Taint analysis relies on the identification of a flow path between a Source (untrusted input) and a Sink (dangerous operation) [1].

However, traditional CodeQL is purely structural and syntax-driven [1]. It excels at identifying the technical possibility of a data flow, but lacks the contextual human-level reasoning needed to detect unrecognized sanitizers, benign contexts, or logically infeasible flows [1]. This limitation constitutes the root cause of the false-positive credibility gap reported $> 98\%$ [1].

B. Comparative Analysis of AI-Augmented Precision

Recent research has explored augmenting static analysis with Machine Learning (ML) techniques to improve the trade-off between maximizing coverage (low False Negatives, FN) and maximizing precision (low False Positives, FP) [1].

Tools such as TASER [3] and InspectJS [4] infer taint specifications (sources, sinks, and propagation rules) to improve static-analysis coverage. Precision-oriented tools, such as Fluffy (a bimodal taint-analysis framework combining static flows with ML-based expectedness) [2] and DeepDFA (dataflow-inspired deep learning) [5], aim specifically to reduce false positives.

We specifically leverage LLM reasoning [6] because it provides a scalable mechanism for performing high-fidelity contextual auditing. LLMs combine the semantic understanding required by precision-focused ML systems with the ability to ingest the raw contextual information extracted by static analysis [1], offering a path toward reducing the long-standing credibility gap.

A high-level comparison of coverage, ML, and LLM-based precision approaches is summarized in Appendix Table II.

III. Methodology: LLM Triage Pipeline

Our core hypothesis is that an LLM can effectively substitute the manual effort of a security auditor by performing *Contextual Reasoning* over the CodeQL data flow trace [1]. The LLM Triage Pipeline is a three-stage process designed to convert the purely structural CodeQL trace into a fully contextual reasoning task [1].

A. Stage 1: Extraction

The goal of this stage is to convert the CodeQL findings from the repository database into a structured, contextual payload suitable for an LLM [1].

1) **Database Creation:** Utilize sample databases provided by the UntrustIDE authors for VS Code extensions, ensuring they are updated and executable with the latest CodeQL CLI [1].
2) **SARIF File Generation:** Run the target CodeQL query (`dataflow/shell-command-injection`) on the databases to produce results in SARIF (Static Analysis Results Interchange Format) [1]. SARIF is essential because it is a human- and machine-readable JSON format that captures rich metadata, including the full symbolic path of the taint flow [1].
3) **Context Extraction:** Post-process the SARIF files to create the full, structured input payload for the LLM [1]:
   a) The complete symbolic taint path from Source to Sink [1].
   b) Relevant code snippets (e.g., 5–10 surrounding lines) at the Source and Sink locations, and any intermediate hop locations [1].

B. Stage 2: LLM Contextual Filtering (The Triage)

This stage feeds the structured payload into a Large Language Model (LLM) for classification [1].

1) **LLM Role:** The LLM is instructed to act as an *Automated Senior Security Auditor*, triaging the alert [1].
2) **Dynamic Prompting:** The prompt is constructed dynamically for each alert, detailing the Rule ID, file path, line number, and providing the raw {sink context}, {source context}, and {data flow context} captured from the SARIF file [1].
3) **Core Reasoning Task:** The prompt forces the LLM to perform contextual sanity checks, asking questions such as: "Given the full code context, is the attacker able to successfully inject a malicious payload..., or is the flow sanitized/benign?". It further guides the LLM with specific constraints, including whether the input is user-controlled, whether sanitation exists, and if the path is reachable in a realistic scenario [1].
4) **Output Requirement [1]:** The LLM must return a structured JSON response with three keys:
   a) `"verdict"`: `"malicious"` — `"benign"` — `"unsure"`
   b) `"confidence"`: `"high"` — `"medium"` — `"low"`.
   c) `"reason"`: A concise explanation referencing specific details from the code snippets to ensure explainability and verify the LLM's reasoning.

C. Stage 3: Validation and Metrics

The final stage evaluates the LLM's performance against manually established Ground Truth (TPs and FPs) [1].

The primary metric for success is **Precision**

$$\frac{TP}{TP + FP}$$

and the overall **FPR Reduction**, while monitoring Recall

$$\frac{TP}{TP + FN}$$

as a guardrail against discarding legitimate vulnerabilities [1].

For the initial sampled dataset:

- **Total Alerts Triaged:** 15 [1].
- **Ground Truth TPs:** 2 confirmed, verifiable exploits [1].
- **Ground Truth FPs:** 13 alerts deemed non-exploitable [1].

The LLM achieved a calculated FPR of 0% on this sample (0 FPs / 15 Total Alerts) and a Triage Accuracy of approximately 86.67% (13 correctly classified / 15 total) [1]. It demonstrated success in FPR Reduction but incurred one False Negative, highlighting the Recall/FNR guardrail [1].

IV. Implementation

To integrate our proposed enhancements into the CodeQL analysis workflow, we introduced several new components into the project architecture. In particular, we added the following files:

- `gemini.py`
- `result_inspector.py`
- regenerated CodeQL databases for each of the five provided sample projects

Due to incompatibilities between the sample databases originally supplied and the current CodeQL release (at the time of writing this paper it is 2.23.6), it was necessary to rebuild each database from source. When reproducing this environment, the user must ensure that each sample project's `src` directory is fully unzipped prior to database creation. A representative command for generating a database is shown below (with user-specific paths generalized):

```
codeql database create <database-output-path>
--language=javascript
--source-root <path-to-unzipped-sample-project-src>
```

Once the databases have been created, any desired CodeQL queries may be executed. In our study, we focused on identifying potential shell-injection vulnerabilities using the `ShellCommandInjection-ATM` query. Following query execution, the results were exported to the SARIF format. SARIF was selected for its interoperability with a broad ecosystem of static-analysis tools, enabling independent verification and cross-tool comparison of the findings.

The `result_inspector.py` module provides the primary mechanism for parsing and interpreting the SARIF output. It extracts contextual code snippets and converts the results into structured JSON for downstream processing. Because CodeQL's native `.bqrs` (Binary Query Result Set) files are not human-readable, direct use of these binary artifacts would hinder natural-language reasoning. The JSON transformation enables seamless integration with the Gemini API and supports higher-quality explanatory outputs.

The `gemini.py` module serves as the interface to the Gemini LLM. For each detected taint flow, the system supplies the model with 5–10 lines of surrounding context from both the source and the sink, along with the associated data-flow metadata. The LLM is required to classify each flow as *benign*, *malicious*, or *unsure*, and must provide a structured rationale, including relevant code fragments. After producing its assessment, the system advances automatically to the next finding.

A. LLM Interaction Logic

The following code listing provides the primary logic used to initiate and manage LLM-driven analysis of individual findings:

```
def start_chat_session(client: genai.Client, finding_record: dict):
    """Starts an interactive chat session for a single security finding."""
    conversation_history: list[types.Content] = []
    initial_prompt = format_finding_as_prompt(finding_record)

    print("--- Sending Initial Prompt to Gemini ---")
    print(initial_prompt)
    print("----------------------------")

    try:
        reply_text = _send_and_record_response(client, conversation_history, initial_prompt)
        print("\n--- Gemini's Analysis ---")
        print(reply_text)
        print("-------------------------\n")
    except Exception as exc:
        print(f"An error occurred while communicating with the Gemini API: {exc}")
        return

    print("Entering interactive chat. Type 'next' to move to the next finding, or 'quit' to exit.")
    while True:
        try:
            user_input = input("You: ")
            lowered = user_input.strip().lower()
            if lowered == "quit":
                raise SystemExit("Exiting.")
            if lowered == "next":
                print("\nMoving to the next finding ...")
                break

            print("...sending to Gemini...")
            reply_text = _send_and_record_response(client, conversation_history, user_input)
            print(f"\nGemini: {reply_text}\n")
        except KeyboardInterrupt:
            raise SystemExit("\nExiting.")
        except Exception as exc:
            print(f"An error occurred: {exc}")
            break
```

B. Dynamic Prompt Construction and Context Injection

1) Contextual Extraction: For every alert processed, the system extracts the specific code snippets corresponding to the *Source* (the untrusted input origin) and the *Sink* (the dangerous function call). To provide the LLM with sufficient semantic understanding—such as checking for local sanitization functions or hardcoded values—we extract a configurable window of 5 to 10 lines of surrounding code for both the source and the sink. Additionally, the full *Taint Path* (the sequence of data flow steps) is parsed and formatted as a numbered list, allowing the model to trace the variable's journey through the application.

2) Prompt Structure and Reasoning: The constructed prompt leverages several prompt engineering techniques to maximize classification accuracy:

- **Persona Adoption:** The model is instructed to adopt the role of a "Senior Product Security Engineer," setting a baseline for high-precision, risk-averse analysis.
- **Few-Shot Prompting:** We include a concrete "Benign Example" within the prompt to demonstrate the expected logic for dismissing false positives (e.g., recognizing input sanitization).
- **Chain-of-Thought (CoT) Reasoning:** The instructions explicitly forbid the model from guessing a verdict immediately. Instead, it is forced to perform a step-by-step analysis—evaluating the source, sink, mitigations, and data flow reality—before generating a conclusion. This piece was missing from our original prompt, and we added this to ensure that the model was not giving quickly classified flows and rather giving more robust "thought" out classifications.

3) Structured Output: To ensure integration with our automated pipeline, the prompt enforces a strict JSON output schema. This requires the LLM to provide a `verdict` (Malicious, Benign, or Unsure), a `confidence` score, and a text-based `reason` that references specific variables from the provided snippets.

Figure 1 illustrates the template used to generate these prompts.

We evaluated both `Gemini-2.5-flash` and `Gemini-2.5-pro` for the core analysis task. While both models function within similar API quota constraints, `Gemini-2.5-pro` was selected for its consistently superior reasoning performance, which proved critical for minimizing false positives in ambiguous code contexts.

V. Discussion and Limitations

A. Key Findings: LLM Performance

The evaluation confirms the potential of the LLM approach [1]:

1) **Path-Aware Reasoning:** The LLM successfully traces multi-hop flows across files when provided with the ordered trace, distinguishing between benign string manipulation and command execution syntax [1].
2) **Contextual Understanding:** The LLM identifies key mitigating factors, such as file path normalization functions

```
# High Level Prompt Overview
prompt = f"""You are a Senior Product Security Engineer...
# ... [Persona and Role Definition] ...

## SINK (Execution Point)
{record.get('sink', {}).get('snippet')}

## SOURCE (Input Origin)
{record.get('source', {}).get('snippet')}

## TAINT PATH (Data Flow)
{path_str}

ANALYSIS INSTRUCTIONS:
Think step-by-step. Do not provide the JSON verdict yet.
Analyze the code in the following order:
1. Source Analysis: Is the source user-controlled?
2. Sink Analysis: Is the sink actually dangerous?
3. Mitigation Analysis: Is there sanitization?
4. Data Flow Reality: Is the path reachable?

RETURN ONLY JSON.
"""
```

Listing 1. Dynamic Prompt Template Construction

(e.g., `path.join()`) and whether the input source (`__dirname`) is realistically controllable by an attacker [1].

3) **Explainability:** The mandatory justification requirement ensures that every verdict is grounded in explicit references to the provided code snippets [1].

B. LLM Reliability and Cost

The LLM approach introduces new trade-offs related to reliability and cost [1]:

1) **Unexpected Weaknesses/Hallucinations:** The model can be susceptible to bias from variable naming (e.g., assuming a variable named "safe" is safe regardless of function), over-trusting weak mitigations, or, in rare cases, inferring flows outside the provided trace [1].
2) **Cost-Effectiveness:** While small-scale testing is economical, large-scale analysis across thousands of repositories would incur significant token costs [1]. Future work must address *Cost-to-Accuracy Optimization* by rigorously studying the performance trade-off across different models and query complexity [1].

C. Limitations

The scope of this project was subject to practical limitations inherent in large-scale analysis [1]:

1) **Scale of Replication:** Reproducing the full UntrustIDE analysis pipeline on 25,402 extensions was computationally and temporally infeasible, requiring immense storage and analysis time (some queries requiring 20 minutes each) [1]. Our work relies on demonstrating the efficacy of the LLM on a carefully selected and verified sample set [1].
2) **Setup Overhead:** Customizing the CodeQL queries to reliably extract structured SARIF output suitable for LLM consumption required significant setup and tuning overhead [1].
3) **Query Domain:** The initial study was limited to a specific vulnerability class (Shell Command Injection) and language environment (JavaScript/Node.js) [1].

The LLM approach provides the necessary contextual bridge between structured program analysis and human auditing, demonstrating a new, scalable method for reducing the debilitating False Positive Rate in static analysis [1].

VI. Conclusion

This study addressed the critical challenge of Alert Fatigue in software security, where high False Positive Rates (FPR) in tools like CodeQL significantly reduce their industrial utility. By introducing a novel, three-stage LLM Triage Pipeline, we successfully bridged the gap between the rigid, structural analysis of static tools and the semantic reasoning required for accurate auditing.

Our empirical validation demonstrated that leveraging Large Language Models, specifically Gemini, allows for effective contextual filtering of taint analysis results. On our validated sample set, the pipeline reduced the effective FPR from the industry baseline of >98% to 0%, achieving a triage accuracy of approximately 86.67%. Furthermore, the system satisfied the requirement for explainability by providing structured, code-referenced justifications for every verdict.

While the scale of this study was constrained by computational resources and manual verification efforts , the results confirm that LLM-driven reasoning can effectively differentiate between benign string manipulations and genuine command injection vulnerabilities. This approach offers a scalable path toward solving the precision problem in Static Application Security Testing (SAST), potentially transforming how security analysts interact with automated findings.

VII. Future Research Directions

1) **Automated CodeQL Query Generation.** Future work can leverage the LLM's explanatory justifications—particularly its reasoning about why certain flows are classified as False Positives—to automatically synthesize improved CodeQL rules and sanitizers. For example, if the LLM determines that "the path is normalized by `path.join()`" before reaching the sink, this insight can be programmatically translated into a new sanitizer specification. Such a system would form a closed feedback loop in which CodeQL continuously evolves based on empirical contextual understanding extracted from real-world code bases [1]. This could significantly accelerate the development of high-quality, domain-specific CodeQL libraries while reducing reliance on manual rule engineering.
2) **Cost-to-Accuracy Optimization.** A rigorous exploration is needed to quantify how LLM model scale, inference cost, and contextual prompt size jointly affect precision, recall, and overall false-positive reduction. This includes benchmarking small, medium, and large models under

controlled conditions and identifying the Pareto frontier between operational cost and triage performance. Additionally, deploying lightweight autonomous agents using a Model Context Protocol (MCP) server [1] may allow dynamic resource allocation—delegating simpler cases to smaller models while reserving large models for ambiguous or high-severity flows. This line of work aims to operationalize LLM-driven static analysis for industrial-scale CI pipelines.

3) **Generalization Across Queries and Languages.** To assess the scalability of the LLM Triage Pipeline, future evaluations should expand beyond Shell Command Injection to include broader vulnerability categories such as Taint Path analysis, SQL Injection, XXE, and Insecure Deserialization [1]. Furthermore, applying the methodology to diverse programming languages (e.g., Python, Java, Ruby, Go) will reveal how well the current prompting strategy and contextual extraction generalize across differing semantics and idioms. This research direction is essential for determining whether LLM-assisted triage can serve as a universal interface atop static analysis engines or whether language- and domain-specific tuning is required for optimal accuracy.

REFERENCES

[1] E. Lin, I. Koishybayev, T. Dunlap, W. Enck, and A. Kapravelos, "Untrustide: Exploiting weaknesses in vs code extensions," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2024, p. 24073.

[2] Y. W. Chow, M. Schäfer, and M. Pradel, "Beware of the unexpected: Bimodal taint analysis," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, USA, 2023, pp. 211–222.

[3] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for javascript libraries," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, Seoul, South Korea, 2020, pp. 198–209.

[4] S. Dutta, D. Garbervetsky, S. K. Lahiri, and M. Schäfer, "Inspectjs: Leveraging code similarity and user-feedback for effective taint specification inference for javascript," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Pittsburgh, PA, USA, 2022, pp. 165–174.

[5] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, 2024, pp. 16:1–16:13.

[6] C. Wang, W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang, "Llmdfa: Analyzing dataflow in code with large language models," *arXiv preprint arXiv:2402.10754*, 2024.

[7] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, 2023, pp. 1059–1076.

VIII. Appendix

Approach Category	Primary Goal	Key Papers/Tools	Shared Limitation
I. Flow Coverage	Ensure all technical flows are found (low FNs) [1].	TASER [3], FAST [7]	High cost (dynamic or abstract interpretation) to gain deep context [1].
II. ML Filtering	Filter noisy flows (low FPs) [1].	Fluffy [2], DeepDFA [5], InspectJS [4]	Low explainability (Black Box ML) or high training cost [1].
III. Our LLM Approach	Perform *Contextual Triage* via LLM reasoning [1].	LLMDFA [6]	Reliability/Hallucination Risk [1].

TABLE I
Comparison of Flow-Coverage, ML-Filtering, and LLM-Triage Precision Approaches