Reducing Noise: Hybrid Static Application Security Testing–Large Language Model Pipeline for Code Security

Vaibhav Agrawal[1], Senior Security Engineer, Google, Mountain View, CA, USA
Kiarash Ahi[2], Founder, Virelya Intelligence Research Labs, San Francisco Bay Area, CA, USA

Abstract — Traditional SAST tools often overwhelm developers with high false-positive rates and theoretical findings. This article discusses, a hybrid triage pipeline that integrates SAST with LLM to provide an intelligent triage and automated exploit generation, reducing false positives by 91% in real-world deployments.

Keywords, SAST, Large Language Models (LLMs), Generative AI, Cybersecurity, Vulnerability Detection, Vulnerability Discovery, Static Analysis, Code Security.

I. Introduction

Static Application Security Testing (SAST) has long served as a foundational, "shift-left" approach in cybersecurity. By automatically scanning source code for vulnerabilities before execution, SAST embeds security early in the Software Development Life Cycle (SDLC), enabling developers to identify and remediate risks proactively and enforce compliance with security standards. SAST tools generally analyze code without execution, inspecting source code, bytecode, or binaries for patterns that match known security vulnerabilities or insecure configurations.

Concurrently, Large Language Models (LLMs) have rapidly emerged as transformative tools in software engineering. These models have significantly benefited developers by generating functional and easy-to-adopt code snippets from natural language descriptions, thereby accelerating the development process. Such capabilities also position LLMs as a powerful new asset for software vulnerability detection [2].

Despite their individual strengths, both SAST and LLMs possess inherent limitations. SAST, while thorough in its static analysis, often struggles with contextual understanding and high false positive rates [1]. LLMs, while adept at language and pattern recognition, can be slow, hallucinate, and return inconsistent responses [2].

In this article, we share our experience developing and deploying a SAST-LLM triage pipeline (we call it SAST-Genius) and discuss the practical challenges, tradeoffs, and lessons learned. We combined a commercial SAST tool with a fine-tuned LLM and conducted an evaluation on a curated dataset of 25 open-source projects to measure the performance gains. Our evaluation demonstrates that this synergistic combination not only significantly reduces false positive rates but also validated complex, contextual vulnerabilities missed or misclassified by traditional SAST.

II. Advantages: LLM-Enhanced Static Analysis for Next-Generation Vulnerability triage

The integration of LLMs with SAST enhances the benefits of standalone SAST tools. Traditional SAST tools struggle with complex logical flaws, multi-file dependencies, and whole-repository analysis, often missing vulnerabilities in "hard-to-track" code paths [1]. This is where LLM synergy becomes critical. **By integrating LLM-based semantic reasoning with SAST outputs, the hybrid triage pipeline can prioritize and validate code regions that traditional tools miss or flag as uncertain**, effectively complementing SAST. Furthermore, in certain cases LLMs can assist in de-obfuscating code, for example, Android application code, restoring meaningful context to obfuscated functions, variables, and classes, thereby enabling static analysis in scenarios where it was previously not possible. This synergy transforms SAST from a localized, rule-bound scanner into a more holistic, context-aware analysis engine, allowing for a deeper understanding of data flow and potential vulnerabilities across an entire project, including its complex dependencies.

SAST's high false positive rates are a persistent challenge, leading to alert fatigue, wasted developer time, and potential abandonment of tools. LLMs can act as an intelligent triage layer to mitigate this. **By performing "contextual analysis" on detected vulnerable paths, LLMs can filter out false positives**. Context includes the code, full source-to-sink data flow together with cross-file function calls, including any sanitization logic within the code, vulnerability details etc. that allows LLMs to reason more reliably about whether a finding is truly exploitable. For example, SQL injections usually occurs due to a malicious user controlled string, but where a user-controlled parameter (user_id) flowed into a database query; however, contextual analysis showed that the input was cast using int(user_id) before query construction, preventing injection and allowing the LLM to correctly classify the finding as a false positive. This is arguably one of the most impactful synergies, directly addressing SAST's biggest pain point and enabling developer productivity.

Unlike traditional SAST, which provides only theoretical findings, **LLMs can reason through source-to-sink conditions to generate Proof-of-Concept (PoC) exploits**. This moves detection toward actionable validation through the creation of targeted test cases. Evaluation shown in Table 1 and III.B are consistent with findings from PoCGen [3].

LLMs can leverage natural language generation to replace hardcoded bug descriptions that comes with SAST with human-readable explanations of function logic and clear verdicts [4]. Furthermore, they can accelerate the remediation phase by suggesting concrete code repairs [5].

Defining custom SAST rules traditionally requires specialized knowledge of proprietary query languages limiting this capability to experienced security engineers. LLMs excel at translating natural language into formal syntax [6]. This can enable security analysts and even less technical stakeholders to define custom security policies and rules using plain English. The LLM can then translate these natural language specifications into the formal rules required by SAST tools, democratizing the rule creation.

III. bridging the gap: our approach with SAST-Genius

SAST-Genius is a two-stage pipeline designed to leverage the strengths of both deterministic static analysis and LLM-based semantic reasoning. Its architecture (Figure 1) is composed of a SAST tools that identifies potential security risks and a Triage engine, which consists of two subcomponents: (i) a lightweight middleware component, the Context extraction layer, responsible for parsing SAST outputs and retrieving the corresponding source code and metadata (ii) an LLM-based analysis layer that performs vulnerability triage, validation, and exploit generation.

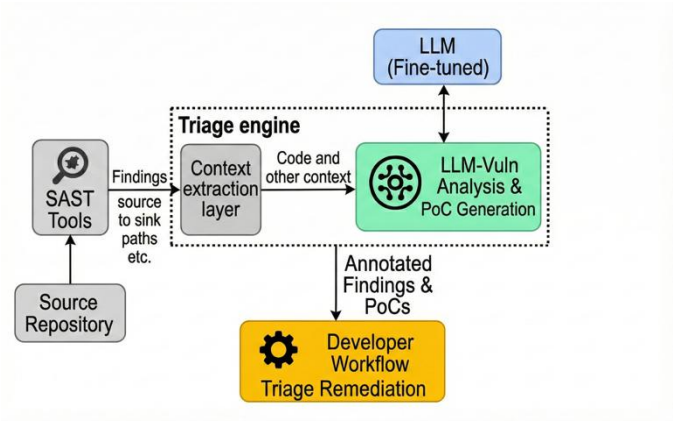

Figure 1. Illustrates the SAST-Genius architecture, a two-stage pipeline where a Triage engine intercepts initial findings from SAST tools to perform context extraction and LLM-based analysis (including PoC generation) before delivering validated, annotated findings to the developer workflow.

A. Intelligent Triage Engine

When the SAST core (we utilized Semgrep, widely used SAST tool) flags a potential vulnerability, the Context Extraction Layer retrieves the relevant code snippets, the identified source-to-sink data flow paths, and surrounding contextual information (e.g., function call graphs and library dependencies). This structured context is then packaged into a prompt and passed to the LLM-Based Analysis Layer for exploitability reasoning. Prompt also includes details such as the SAST finding details (e.g., the rule name and severity score), file path [7]. This method allows the LLM to reason more reliably about whether a finding is truly exploitable by analyzing the context that traditional SAST rules often miss. The output is a simple verdict like "True positive" or "False positive", with a reasoning summary.

B. Automated Exploit Generation

For vulnerabilities that the LLM confirms are exploitable, i.e verdict given as "True positive", our triage pipeline automatically generates a proof-of-concept (PoC) exploit. This is achieved by prompting the LLM with a simple system instruction like "You are a Senior Security Research Engineer. Your task is to evaluate a potential vulnerability identified by a SAST tool and LLM triage engine and generate a Proof of Concept (PoC) to validate its exploitability" and provide all the details that were provided in intelligent triage phase asking it to reason about the required input to traverse the vulnerable path and generate an executable payload or command. This process is crucial for verifying vulnerabilities and providing actionable intelligence to developers [3].

This capability is key for moving findings from theoretical to actionable insights. In our evaluation, SAST-Genius successfully generated valid PoCs for approximately 70% of exploitable findings in our dataset, significantly reducing the manual verification burden on security analysts.

C. Running Example: Triage Engine In Action

The following example illustrates the pipeline workflow on a representative finding.

Stage-1 SAST Tools: The pipeline begins when the SAST tool (Semgrep) scans the repository and generates a JSON output of all the vulnerabilities. Among other details, the raw findings include the semgrep rule ID, a severity rating, the file path, and the identified source-to-sink flow. The raw finding identifies that untrusted input enters through app/routes/download.py and flows into a dangerous file-read sink in src/utils/file_ops.py. The identified source-to-sink flow is: download.py: request.args.get("file_path") → file_ops.py: download_file(user_path) → send_file(full_path)

Stage-2 Context Extraction Layer: The middleware intercepts and parses the SAST tool's JSON output including vulnerability type etc. and enriches it by retrieving the full code snippet of vulnerable file and all the intermediate representation. This structured context is then passed forward.

Stage-3A LLM Analysis Layer: The analysis layer then packages the received context into a prompt and sends it to the LLM, instructing it to confirm the validity of the vulnerability, checking sanitization in place etc. The model returns a structured JSON verdict indicating a true positive and provided a reasoning:

```
{
"verdict": "True Positive",
"reasoning": "Path traversal in download.py file has been
verified, path traversal sequences (e.g., ../) can easily
escape the intended directory since no prefix check or
absolute path resolution is applied……"}
```

Stage-3B PoC Generation: Because the triage verdict is a True Positive, the pipeline prompts the LLM for exploit generation. The model outputs a validated executable payload:

```
{
  "vulnerability_type": "Path traversal vulnerability",
  "payload": "curl -X GET
'http://[server]/api/v1/download?file_path=../../../../et
c/passwd'",
  "execution_context": "Execute this command in any
standard Unix/Linux terminal or command-line interface
with 'curl' installed…"
}
```

This command would send a GET request to read the server's /etc/passwd file, which contains local user account information (usernames, IDs etc.), validating that vulnerability is a true positive.

Stage-4 Developer Workflow: Finally, a validated, annotated bug ticket can be automatically generated with all the necessary information. This is custom to the engineering teams.

Our sample triage & exploit prompts are available on SAST-Genius Github repo [7].

IV. Deployment Evaluation

A. Experimental Setup

Our evaluation was based on a diverse dataset of 25 open-source repositories from GitHub, totaling over 250,000 lines of code. The repositories were selected based on their active development and language diversity (Python, Java, JavaScript). A 'ground truth' set of 170 vulnerabilities across selected repositories was established through a combination of manual expert analysis and cross-referencing with public vulnerability reports.

Our baseline was Semgrep 1.97.0 with its standard rules, a widely-used SAST tool. We also used GPT-4 as a separate baseline to assess a purely LLM-based approach. SAST-Genius used a fine-tuned version of Llama 3 8B LLM, trained using supervised learning with low-rank adaptation (LoRA) for parameter-efficient fine-tuning, on a proprietary dataset comprising around 500 samples of validated security findings and historical false positives from retrieved from sources such as GitHub Security Advisories [8]. Crucially, this training dataset is entirely disjoint from the 25 open-source repositories and the 170 ground-truth vulnerabilities used for evaluation. No training data leaked

into the evaluation phase, this strict separation ensures the validity of the reported metrics. Due to the proprietary nature of our dataset for fine tuning, we can't release it entirely, but we have created a 10-sample synthetic representative dataset which perfectly mirrors the structural format, annotation schema used in our actual training pipeline, which is publicly available on our Github repo, along with the 25 repos we used in our experiment [7].

We measured the following metrics: Precision, Recall, F1-Score, and Time-to-Triage (the average time a security analyst took to confirm a finding). Recall is calculated against a ground-truth set of 170 vulnerabilities (Semgrep 73.5 %, GPT-4 77.1 %, SAST-Genius 100 %), which produces the F1-scores reported in Table 1.

B. Results and Analysis

Our results from this pilot shows a clear and significant advantage of the hybrid SAST-Genius pipeline over standalone tools.

TABLE 1: Observed Outcomes in Our Evaluation

Tool	Findings	True Positives	False Positives	Precision	F1	Recall
Semgrep	350	125	225	35.7%	48.3 %	73.5 %
GPT-4	200	131	69	65.5%	70.8 %	77.1 %
SAST-Genius	190	170	20	89.5%	94.5 %	100 %

SAST-Genius achieved a precision of 89.5%, a significant improvement over Semgrep's 35.7 % and GPT-4's 65.5 %. Our pipeline's ability to intelligently filter out non-exploitable findings directly addressed the high false positive rate that plagues traditional SAST tools.

Our analysis revealed that Semgrep produced 225 false positives, while SAST-Genius reduced this to just 20, a 91 % reduction. SAST-Genius achieved 100% recall within this bounded ground-truth set of 170 vulnerabilities identified through manual expert analysis and public advisories. Semgrep was executed using its default ruleset without custom tuning, and informational findings were excluded from baseline recall calculations but included as candidates in the SAST-Genius pipeline.

The LLM-based triage engine operates on structured outputs from the SAST tool rather than raw source code, constraining the reasoning space to explicit source-to-sink data flows and code context. The model is fine-tuned on approximately 500 expert-labeled security findings to perform binary exploitability classification. As a result, our system preserves SAST recall, while primarily improving precision through semantic reasoning. Although, for high-severity findings, the LLM's verdict is used as a prioritization signal, while final validation and remediation decisions remain under human expert review.

TABLE 2: SAST- Genius Latency

Component	Measurements	Metrics
Triage verdict	RTT for Llama 3 8B to generate verdict	~1s - 4s
PoC Generation	Time to reason and generate executable payload	~2s - 5s
Total Pipeline	End-to-end automated processing time	< 10 seconds

In our analysis, SAST-Genius introduced an average latency of less than 10 seconds per finding when running on a laptop with 32GB of RAM.

From a resource cost perspective, SAST-Genius utilizes an on-premise Llama 3 8B model, which eliminates the per-token costs associated with cloud-based APIs while ensuring data privacy. While hardware requirements vary by organizational need, but the marginal cost to run each triage scan is near-zero after the initial infrastructure setup, as it relies on local compute rather than external service fees for inference.

C. Analysis of Significant reduction of False Positives

The dramatic reduction in false positives (from 225 to 20) is achieved because our hybrid pipeline addresses the fundamental weakness of traditional SAST. Semgrep provides the deterministic evidence (the taint path etc.), and the fine-tuned LLM trained on a high-quality data set of publicly known vulnerabilities, provides the semantic context. This combination allows the LLM to act as an intelligent filter, reasoning whether external factors such as cross-function and cross-file data sanitization, or even multi-step code logic may render the taint path unexploitable.

For example, in our evaluation Semgrep flagged an XSS vulnerability in app/render.py, where a user input passed through a sanitization function sanitize_html, causing it to miss the sanitization step. Our Intelligent Triage Engine, with access to the entire function call graph and code context, recognized that sanitization was in place, confidently rendering the finding a False positive.

In another case, Semgrep flagged a potential SQL Injection vulnerability in api/user_info.py. The taint path was correctly identified with user-controlled input (user_id) used in a database query. However, the user_info was being converted into an integer (int(user_id)). Casting the input to an integer prevents the inclusion of single quotes (') required for a classic SQL Injection payload, the LLM confidently rendered the finding a False positive, contributing to our triage pipeline's improved precision.

A qualitative review of the remaining 20 false positives revealed two primary patterns: (1) complex sanitization logic that the LLM failed to recognize despite having the context, and (2) domain-specific business logic, such as vulnerable code paths guarded by environment flags (e.g., if (env == 'internal')) which effectively neutralized the risk in production but were flagged as actionable by SAST-Genius. All false positives produced by SAST-Genius were a strict subset of those generated by Semgrep and GPT-4 in isolation, the hybrid system did not introduce any unique new false positives.

D. Analysis of Missed Vulnerability Classes

Beyond quantitative improvements, SAST-Genius validated vulnerabilities that were missed or misclassified by the SAST tool.

Directory Traversal (Multi-File Flows) – SAST-Genius correctly identified a multi-file Directory Traversal vulnerability and generated a valid PoC. Semgrep flagged a user-controlled file path but could not analyze the multi-file logical flow to determine if a vulnerable download function in a separate module was called. The LLM, however, was able to reason about the complete data flow, providing the following trace, "*The user input in file_path is passed un-sanitized to the download_file function in src/utils/file_ops.py. This function concatenates the input path with the root directory, allowing a ../ attack to access system files*".

Obfuscated Secrets – Similarly, Semgrep flagged a Base64 encoded string embedded in a utility file as an informational (a potential secret) but LLM reasoned that the string was being passed to a function named auth.decrypt_config() and was subsequently used as a key header for an external API call to a partner service thus triaging and validating the finding.

Other identified issues include, SQL Injection in Nested Contexts, instances where un-sanitized user input was embedded in dynamically constructed queries across multiple layers of abstraction.

The success of SAST-Genius in validating vulnerabilities that were missed or misclassified by SAST alone is rooted in the fact that LLMs can reason over complex semantic relationships when provided with structured context.

V. Recommendations for Practitioners

The critical challenge for practitioners is integrating the LLM's intelligence without disrupting existing Continuous Integration/Continuous Delivery (CI/CD) pipelines. This is how we integrated LLM with existing SAST pipeline.

Seamless integration into DevSecOps Pipelines: SAST-Genius functions as an Orchestration Layer inserted between the traditional SAST scan and the final action (e.g., creating a bug ticket or failing a build).

1) Run the SAST tool (e.g., Semgrep Pro) as usual, but intercept the raw output with a custom middleware script, in our case "Context extraction layer", Crucially, this script should be configured to ingest all finding levels (including 'Informational' and 'Low Severity') that are typically suppressed in standard developer reports. This allows the LLM to analyze weak signals that may be true vulnerabilities. The script's job is to fetch all necessary cross-file function definitions, taint flow etc. from the Semgrep results file and convert the data into a structured prompt for the LLM as mentioned in the section **III** A.

2) Feed the structured prompt to the LLM Triage Engine. The LLM, ideally deployed on-premise or in a private cloud to eliminate data leakage risk, applies contextual reasoning to determine the definitive verdict (True/False Positive).

3) Implement logical gates based on the LLM's verdict (i.e true/false), such as auto-suppressing all false positives. For all true positives, immediately trigger PoC generation and auto-create a high-priority, validated bug ticket.

Integrate Multi-Tool Orchestration: The future of application security is not a single tool, but an AI-powered control plane that coordinates multiple tools. Security teams can explore extending the Triage Engine beyond SAST findings to integrate data from other sources. For instance, feeding the LLM with findings from Software Composition Analysis (SCA) tools (to prioritize vulnerabilities based on package exposure). The LLM becomes the unified brain for a modular, multi-tool security strategy.

Investing in Specialized Data and Models: The core performance success of SAST-Genius (e.g., 89.5% precision) lies in the specialized fine-tuning of the LLM component as mentioned in Section IV.A. Practitioners must view data curation as a strategic security investment. Organizations should begin building a proprietary, high-quality security dataset composed of their validated false positives, confirmed true positives, and historical bug reports. This custom dataset is the key to training a domain-specific model that understands the organization's unique code patterns and security policy exceptions, thereby achieving superior contextual triage and making the LLM a truly custom defense asset.

Operational Considerations: Integrating LLMs into security pipelines introduces new risks that teams must manage. Prompt injection is possible if attacker-controlled code is passed directly to the model, and training data quality strongly influences triage accuracy. To mitigate these risks, teams should sanitize inputs, carefully curate training data, and keep a human-in-the-loop for high-severity findings. In our experience, LLMs are most effective when used to reduce manual effort not to fully replace expert judgment.

ACKNOWLEDGMENT

The authors used generative AI tools (ChatGPT and Gemini) to assist with grammar and language improvements. All content was reviewed, verified, and finalized by the authors.

DISCLAIMER

Author contributions were made in a personal capacity and do not necessarily reflect the views of the author's employers.

References

1. M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE), 2016, pp. 332–343.
2. Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," arXiv preprint arXiv:2401.11817, 2024. [Online]. Available: https://arxiv.org/abs/2401.11817
3. D. Simsek, A. Eghbali, and M. Pradel, "PoCGen: Generating proof-of-concept exploits for vulnerabilities in npm packages," arXiv preprint arXiv:2506.04962, 2025. [Online]. Available: https://arxiv.org/abs/2506.04962
4. J. Ahmad, B. Ding, S. Fan, Z. Lin, Y. Lu, and C. Zheng, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in Proc. Int. Conf. Learn. Represent. (ICLR), 2025. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2025/hash/582d4e27fa24168f3af1f4582655034b-Abstract-Conference.html
5. Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "LLMs in software security: A survey of vulnerability detection techniques and insights," ACM Comput. Surv., vol. 58, no. 5, Art. no. 32, pp. 1–35, 2025, doi: 10.1145/3769082.
6. E. Basic and A. Giaretta, "From Vulnerabilities to Remediation: A Systematic Literature Review of LLMs in Code Security," arXiv preprint arXiv:2412.15004, 2024. [Online]. Available: https://arxiv.org/abs/2412.15004.
7. V. Agrawal, "SAST-Genius," GitHub, 2025. [Online]. Available: https://github.com/In3tinct/sast-genius.
8. "GitHub Security Advisories," GitHub, 2025. Accessed: Dec. 14, 2025. [Online]. Available: https://github.com/advisories.