Toward LLM-Driven Rule Generation for Enforcement Systems: An Exploratory Study on WAF

Quanzhi Fu
quanzhif@vt.edu
Virginia Tech
Blacksburg, VA, USA

Dan Williams
Virginia Tech
Blacksburg, VA, USA
djwillia@vt.edu

Abstract

Rule-based security enforcement systems, from WAF configurations to SELinux policies, require continuous updates to address emerging threats, yet manual rule maintenance remains slow and expertise-intensive. Large language models are a natural candidate for accelerating this process given their demonstrated capabilities in security analysis. However, static LLM-generated rules cannot generalize to unseen attacks, and LLM inference latency precludes direct real-time deployment. We explore a hybrid approach in which a fast rule engine handles known patterns while an LLM analyzes unmatched traffic and generates new rules to cover similar future requests. To investigate this idea, we build VibeWAF, a prototype pairing an LLM with a ModSecurity rule engine, and evaluate it on the SR-BH 2020 dataset. Our results demonstrate that the hybrid approach is viable: the feedback loop converges to a 88% rule hit rate, reducing average latency from 6.5 s to under 400 ms. We also find that when novel attack categories are introduced, previously accumulated whitelist rules can silently admit attacks, slowing the system's adaptive response. Additionally, blacklist and whitelist rules exhibit fundamentally different convergence behaviors, and append-only rule accumulation produces continuous growth. These findings provide empirical grounding for the design of hybrid enforcement systems and identify directions for future work including rule lifecycle management and architectural improvements such as asynchronous LLM analysis and warm-start initialization.

Keywords

Large Language Models, Rule Generation, Security Enforcement Systems, Agentic Systems

1 Introduction

Rule-based enforcement systems are ubiquitous in security infrastructure. Web application firewalls (WAF) match HTTP requests against detection rules, SELinux policies govern process permissions through access control rules, and seccomp filters restrict system calls via allowlist specifications. These systems share a common challenge: rules must continuously evolve to address emerging threats. Yet rule maintenance remains slow and expertise-intensive—when the Log4Shell vulnerability emerged in 2021 [12], major WAF providers required hours to patch each attack variant [20], still too slow to prevent widespread damage [5, 7, 21].

Large language models offer a promising capability for this problem. LLMs have demonstrated competence in analyzing network traffic [2, 30], understanding code semantics [9, 10], and generating structured security rules [3]. However, deploying LLMs directly for real-time classification is impractical, since inference latency of seconds per request far exceeds the sub-millisecond decisions required by most enforcement systems.

This tension motivates a hybrid approach: rather than generating static rules offline or classifying requests in real time, LLMs can continuously maintain rule sets by analyzing traffic that escapes existing rules and generating new rules to handle similar requests in the future. As rules accumulate, an increasing share of traffic shifts from slow LLM analysis to fast rule matching. If this feedback loop converges, the system would combine LLM-level detection with rule-engine speed. However, whether convergence actually occurs in practice, and what behaviors emerge along the way, remains an open question.

In this paper, we explore this hybrid idea through VibeWAF, a prototype that pairs an LLM with a ModSecurity [25] rule engine for WAF rule generation. We choose WAFs as our case study because rule syntax is well-standardized, labeled datasets exist for evaluation [16], and the domain represents a significant real-world application; however, we believe the approach generalizes to other rule-based enforcement systems. We identify four open questions about hybrid enforcement: 1. whether the feedback loop converges, 2. whether the system can adapt to novel attack categories, 3. how rule generation strategy affects detection, and 4. how the rule set scales over time. We empirically investigate each through experiments on the SR-BH 2020 dataset.

Our key findings include:

- VibeWAF achieve comparable detection performance to expert-crafted rule sets while responding to novel attacks within seconds, indicating their potential as a rapid-response complement to existing rule sets.
- The feedback loop converges: rule hit rate reaches approximately 88%, reducing average latency from ~6.5 s (LLM-only) to under 400 ms.
- When novel attack categories are introduced, previously accumulated whitelist rules can silently admit attacks, slowing adaptive response and revealing that append-only rule sets require lifecycle management to maintain detection integrity.
- Blacklist and whitelist rules exhibit fundamentally different convergence behaviors—blacklist hit rate rises to 88% while whitelist hit rate plateaus around 63%, revealing that LLMs currently generate more effective blocklist rules.
- Under append-only accumulation, the rule set grows continuously (750+ rules from 5,000 requests), motivating the need for rule consolidation mechanisms.
- Per-category analysis reveals complementary strengths between LLM-generated and expert-crafted rules, suggesting opportunities for warm-start initialization.

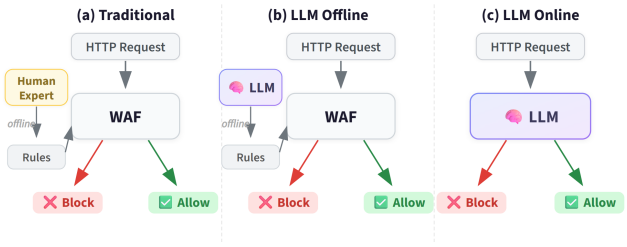

Figure 1: Approaches to WAF rule maintenance.

These findings provide empirical grounding for the design of hybrid enforcement systems and identify concrete directions for future work, including rule lifecycle management, and architectural improvements.

2 Background & Motivation

2.1 Large Language Model

Large language models (LLMs) are language generation models that have demonstrated strong capabilities on a variety of fields [4]. A key property of LLMs is their *instruction-following* ability: given a natural-language specification, an LLM can produce outputs that conform to precise workflow and syntactic constraints making them suitable as components in larger automated systems. Beyond structured generation, LLMs have demonstrated competence in a range of security-relevant tasks, including network traffic analysis [2, 30], code semantic comprehension and malware detection [9, 10], and automated security rule generation [3].

2.2 Rule-Based Enforcement Systems

We define a *rule-based enforcement system* as any system that evaluates operations against a set of predefined rules and permit or denies the operation. This abstraction includes a broad class of security mechanisms: OS-level mandatory access control frameworks such as SELinux [17], network firewalls (e.g., iptables/nftables), Web Application Firewalls such as ModSecurity [25], and network intrusion detection systems such as Suricata [13]. These systems share a common architecture: a *rule set* that specifies matching conditions and a *matching engine* that enforces them at runtime. Enforcement systems sit on the critical path of their respective workloads, and must respect strict latency requirements.

2.3 LLM for Enforcement System Rule Generation

In the remainder of this paper, we use WAFs as our primary case study, as they represent a well-understood and widely deployed instance of rule-based enforcement systems.

Figure 1(a) illustrates the conventional approach that relies on human experts to maintain detection rules offline. This workflow is bottlenecked by the manual rule-engineering cycle: manually crafting effective rules and updating them is inherently slow.

An intuitive improvement is to replace the human expert with an LLM that generates rules offline, while retaining the enforcement engine for real-time matching (Figure 1(b)). We first verify that LLMs can reliably produce valid rules. We prompt Claude Sonnet 4.5 [1] to generate WAF rules for 40 malicious requests from SR-BH 2020 [16]. Each request is fed individually to the LLM, which returns a `SecRule`; we then test whether that rule blocks the original request through the Coraza [24] engine.

Generation Method	Success Rate
Single-shot generation	90.7%
With error feedback loop	100.0%

Table 1: Success rate of LLM-generated rules in blocking target requests.

Table 1 shows that even with naive prompting, the LLM achieves 90.7% single-shot success; with a simple feedback loop that passes syntax error messages from Coraza back for regeneration, success rate reaches 100%. This confirms that LLMs can serve as reliable rule generators.

However, static rules failed to maintain coverage for new threat. We simulate the emergence of novel attack types via leave-one-category-out experiments. Using stratified sampling from SR-BH 2020 (N=312 per attack category, 12 categories total) with 50% normal and 50% attack requests, we generate rules from 11 attack categories and evaluate detection on the held-out category(SQL Injection). This models simulate scenarios where a new threat class emerges after initial rule generation.

Phase	Recall	F1	FPR
Training (seen categories)	0.676	0.746	0.124
Test (held-out SQL Injection)	0.359	0.470	0.170

Table 2: Detection rate on held-out SQL Injection category. Rules generated from other attack types fail to generalize.

Table 2 details the results. Rules achieve 67.6% recall on seen categories but only 35.9% on the held-out category, missing nearly two-thirds of attacks. This experiment shows that static rule sets cannot anticipate attack patterns they have never observed.

Given the limitations of static rules, another natural alternative is to use the LLM directly as a request classifier (Figure 1(c)). We randomly sampled 200 requests from the SR-BH 2020 dataset [16] and compared OWASP Core Rule Set (CRS) [14] with paranomia 4 against Claude Sonnet 4.5 [1].

Method	P	R	F1	Latency
OWASP CRS	0.83	0.78	0.80	<1ms
Claude Sonnet 4.5	0.75	0.85	0.79	~6s

Table 3: Classification performance comparison.

Table 3 shows the results. While the LLM achieves a comparable F1 score with OWASP CRS (0.80 vs. 0.79), it requires approximately 6 seconds per request, which is multiple orders of magnitude slower

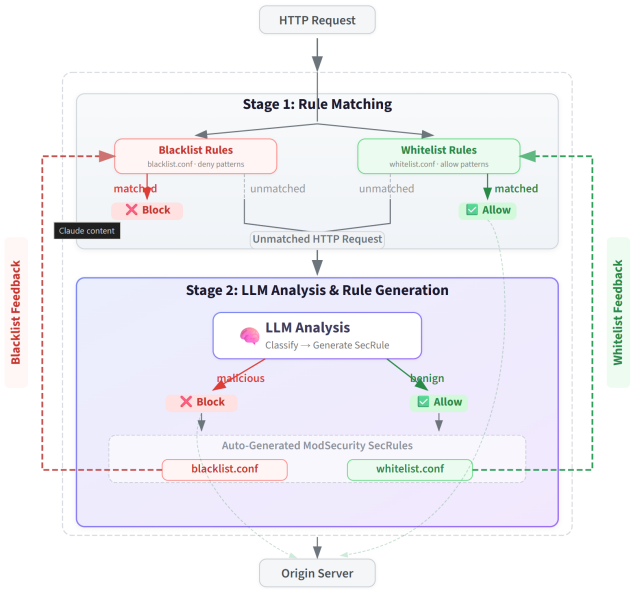

Figure 2: VibeWAF architecture. Stage 1 blocks known attacks via blacklist rules and allows known-safe traffic via whitelist rules. Stage 2 invokes the LLM only for unmatched traffic and generates new rules that feed back into Stage 1.

than real-time enforcement systems. This experiment confirms LLM capacity for traffic detection, while also showing that direct LLM deployment on the critical path is infeasible.

Our experiments demonstrated three findings: (1) LLMs can reliably generate valid enforcement rules (Table 1), (2) static rule sets fail to generalize to unseen attack categories (Table 2), and (3) direct LLM classification, while effective, is too slow for real-time enforcement (Table 3). Together, these findings indicate that the LLM must remain online to address novel threats, yet cannot serve as the sole decision-maker. This motivates us designing a hybrid architecture in which a fast rule engine handles the majority of traffic while an LLM analyzes requests that escape existing rules and generates new rules to absorb similar future requests.

3 VibeWAF: Prototype & Open Questions

To explore the hybrid enforcement idea, we build VibeWAF, a closed-loop prototype that continuously generates WAF rules from observed traffic. As illustrated in Figure 2, the system processes requests through two stages.

Stage 1: Rule Matching. Each incoming request is first evaluated against the current rule set. If any rule matches, the request is classified according to that rule (blocked or allowed) without invoking the LLM. This stage operates at sub-millisecond latency, making it suitable for high-throughput deployment.

Stage 2: LLM Analysis. Requests that escape all existing rules—i.e., *unmatched* traffic—are forwarded to the LLM for analysis. The LLM examines the full HTTP request (headers, URI, and body), determines whether it constitutes an attack, and if so, generates new rules to detect similar requests. Newly generated rules are validated and added to the rule set, where they influence Stage 1 matching for subsequent requests.

Together, these two stages form a feedback loop: Stage 2 continuously produces rules that expand Stage 1's coverage, which in turn reduces the volume of traffic reaching Stage 2. This *progressive offloading* is the central mechanism of the hybrid approach—if the rule set can absorb an increasing share of traffic over time, the expensive LLM analysis becomes sustainable. Whether this actually occurs, and what behaviors emerge along the way, motivates the following open questions.

Q1: Does the feedback loop converge? The hybrid architecture rests on the assumption that accumulated rules progressively absorb traffic, reducing LLM invocations to a sustainable level. If traffic diversity outpaces rule accumulation the hybrid approach becomes impractical. Whether and how quickly convergence occurs is therefore a prerequisite for evaluating any hybrid enforcement system.

Q2: Can the system adapt to novel attack categories? In practice, new threat classes emerge after deployment—the scenario that motivates keeping the LLM online. Whether the hybrid system can detect previously unseen attack categories and automatically generate effective rules without manual intervention remains untested.

Q3: How does rule generation strategy affect detection? Generated rules fall into two categories: blacklist rules that match known-malicious patterns and whitelist rules that match known-safe patterns. Blacklist rules risk missing attack variants they were not designed for; whitelist rules risk blocking legitimate requests that deviate from observed patterns. How these two strategies differ in detection accuracy and rule absorption efficiency, and whether LLMs can reliably produce effective rules for both, remains unclear.

Q4: How does the rule set scale? As the feedback loop operates, new rules continuously enter the rule set. Our prototype adopts the simplest append-only strategy to observe the natural rule set growth characteristics: how quickly does the rule set expand, and does the growth rate change as the system matures? These observations can inform the design of more sophisticated lifecycle management strategies.

4 Implementation

We implement VibeWAF in 1,590 lines of Go (v1.23.2) [23], using Coraza v3.2.1 [24], a Go-native WAF engine that supports the ModSecurity SecRule language [25], as the rule matching backend. We use Claude Sonnet 4.5 [1] as our LLM backend, accessed through Anthropic's API with temperature set to 0. The full system prompt template is provided in Appendix A.

5 Experiments

We empirically investigate the four open questions identified in §3 using the VibeWAF prototype. Specifically, We first present the overall detection performance to establish that the hybrid approach is viable, then examine whether the feedback loop converges (Q1), can VibeWAF seamlessly handle novel attack category (Q2), how blacklist and whitelist rule strategies differ in detection behavior

(Q3), and how the rule set grows under append-only accumulation (Q4).

5.1 Experimental Setup

Dataset. We use the SR-BH 2020 dataset [16], which contains HTTP requests collected from real-world production environments, labeled across 12 attack categories. Since the original dataset is heavily skewed (e.g., SQL Injection accounts for over 63% of attacks), we construct a balanced evaluation set of 4824 requests via stratified sampling: 50% attack requests sampled equally across all 12 categories (201 samples each) and 50% normal requests. VibeWAF starts with an empty rule set, allowing us to observe how the system constructs detection logic from scratch.

Metrics. We report *Accuracy* (fraction of correctly classified requests), *Precision* (fraction of blocked requests that are true attacks), *Recall* (fraction of attacks detected), and *F1 score* (harmonic mean of precision and recall).

5.2 Overall Detection Performance

Table 4 summarizes end-to-end detection performance after processing all 5,000 requests. We evaluate three VibeWAF configurations: combined (blacklist + whitelist rules), blacklist-only, and whitelist-only, alongside the two baselines.

Method	Acc.	Prec.	Rec.	F1
CRS	0.866	0.917	0.793	0.850
LLM-only	0.828	0.807	0.843	0.825
VibeWAF	0.806	0.736	0.928	0.821
VibeWAF-blacklist	0.738	0.693	0.813	0.749
VibeWAF-whitelist	0.809	0.818	0.754	0.785

Table 4: Overall detection performance.

Starting from an empty rule set, VibeWAF achieves the highest recall among all methods (0.928), surpassing both CRS (0.793) and LLM-only (0.843). This comes at the cost of lower precision (0.736 vs. 0.917 for CRS), suggesting that accumulated rules introduce false positives over time. The three VibeWAF configurations also exhibit notable differences: the blacklist-only variant favors recall while the whitelist-only variant favors precision, reflecting the generalization–precision trade-off inherent to rule strategy choice. We investigate these behaviors in detail through Q1–Q4 in the following sections.

Experiment cost. Over our experiment (14,472 total requests across three VibeWAF configurations), total LLM API cost was $10.32, averaging $0.71 per 1,000 requests.

5.3 Q1: Does the Feedback Loop Converge?

Figure 3 plots the rule hit rate and average latency over the course of 5,000 requests. The overall rule hit rate rises from 48% to 88%, while average latency drops correspondingly from around 3,500 ms to under 400 ms. This confirms that accumulated rules progressively absorb traffic, reducing reliance on expensive LLM analysis.

Figure 4 provides a complementary view by comparing the full VibeWAF system (rule + LLM) against the rules-only results along

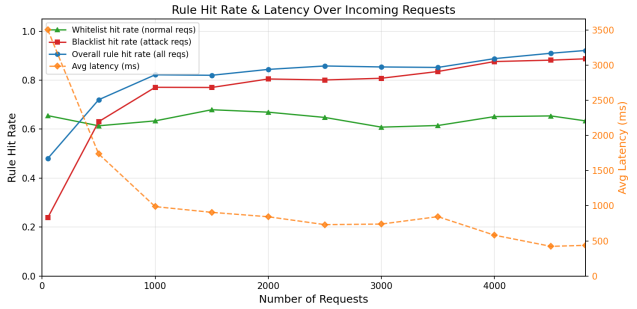

Figure 3: Rule hit rate and average latency over incoming requests. As rules accumulate, more requests are handled by fast rule matching, reducing average latency.

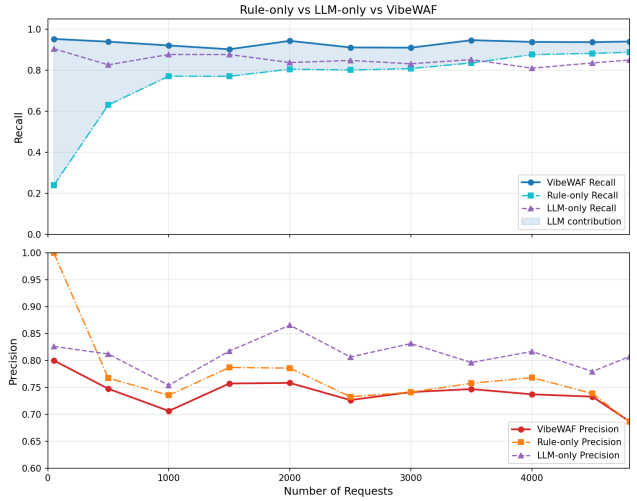

Figure 4: Detection metrics over time. Solid lines: full VibeWAF (rule + LLM); dashed lines: rules-only without LLM fallback. The shaded area represents the LLM's contribution to detection, which diminishes as rules accumulate.

the way with no LLM fallback. The full system maintains stable recall throughout (≈0.93), while the rules-only baseline starts at 0.23 and climbs steadily toward 0.89. The shrinking gap between the two curves represents the diminishing fraction of attacks that require LLM detection—visual evidence that the feedback loop is transferring detection capacity from the LLM to the rule set.

Notably, VibeWAF achieves consistently higher recall than LLM-only but lower precision, suggesting that generated rules cast a wider net than direct LLM judgments—catching more attacks but also introducing additional false positives. This points to the need for rule refinement mechanisms.

5.4 Q2: Can the System Adapt to Novel Attack Categories?

To test adaptation to unseen threats, we hold out SQL Injection from the initial training phase. The system first processes 4,493

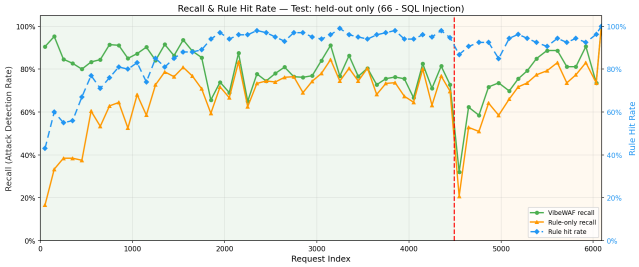

Figure 5: Recall and rule hit rate under novel attack injection. SQL Injection is held out during the first phase and introduced at the dashed vertical line. Post-injection metrics are computed over SQL Injection requests only.

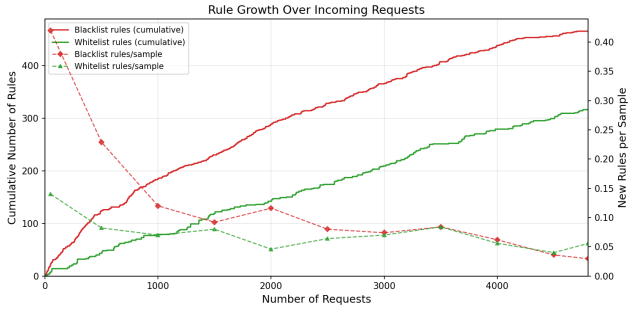

Figure 6: Cumulative rule count and per-sample rule generation rate over incoming requests, broken down by blacklist and whitelist rules.

requests drawn from the remaining 11 attack categories and normal traffic, allowing the feedback loop to converge. We then inject a mix of SQL Injection and normal requests (800 requests each) into the ongoing traffic stream and track how the system responds.

Figure 5 shows the results. Contrary to our expectation that the LLM would immediately detect and absorb the novel attacks, recall drops sharply at the injection point (dashed vertical line). Investigation reveals that previously accumulated whitelist rules inadvertently match many SQL Injection requests, allowing them to pass without reaching the LLM for analysis. Because these requests are absorbed by existing rules rather than flagged as unmatched, the LLM is never invoked to generate corresponding blacklist rules.

Recovery occurs only as some SQL Injection variants escape existing rule coverage, reach the LLM, and trigger new blacklist rule generation. As these rules accumulate, recall gradually improves, reaching approximately 80% by the end of the experiment. However, the recovery is slower than what the Q1 convergence results would predict, as existing whitelist limits the volume of novel attacks reaching the LLM.

This finding exposes a limitation of the current append-only design: once a whitelist rule is added, it permanently influences classification for all future requests, including attack types not yet observed when the rule was generated. Addressing this requires mechanisms to retrospectively evaluate and revise existing rules as new threat categories emerge.

5.5 Q3: How Does Rule Strategy Affect Detection?

Comparing the three VibeWAF configurations against LLM-only in Table 4 reveals that translating LLM judgments into static rules is not lossless. Blacklist-only achieves lower recall and precision than LLM-only (0.813/0.693 vs. 0.843/0.807), indicating that blacklist rules are an imperfect approximation of the LLM's detection capacity. The combined configuration recovers and exceeds LLM-only recall (0.928 vs. 0.843) by aggregating both rule types, but at the cost of lower precision (0.736 vs. 0.807)—the two rule sets stack their coverage but also stack their errors.

The hit rate breakdown in Figure 3 explains this divergence. The blacklist hit rate on attack requests rises from 0.24 to approximately 0.88, demonstrating strong convergence: rules generated for earlier attacks effectively match later variants. In contrast, the whitelist hit rate on normal requests remains relatively flat at 0.60–0.65, showing little improvement over time. This asymmetry reflects that attack traffic tends to follow recognizable patterns amenable to rule-based generalization, while normal traffic is far more diverse, making it difficult to capture safe patterns with a manageable set of allowlist rules.

These results expose a different facet of the whitelist challenge from Q2: while Q2 shows that whitelist rules can be too broad, admitting novel attacks, the convergence gap here shows they are simultaneously too narrow to cover the diversity of normal traffic. Together, these findings suggest that whitelist rule quality is a central bottleneck in the current system, motivating future work on rule generation strategies that better balance coverage and specificity.

5.6 Q4: How Does the Rule Set Scale?

Figure 6 shows rule accumulation under our append-only strategy. The cumulative rule count grows continuously, reaching approximately 450 blacklist rules and 310 whitelist rules after 5,000 requests. The generation rate (new rules per request, dashed lines) shows that blacklist rule generation drops sharply from approximately 0.40 to under 0.05 per request, while whitelist rule generation decreases more gradually from 0.15 to approximately 0.04.

The declining generation rate indicates that earlier rules successfully generalize to cover later requests—consistent with the rising hit rates observed in Q1. The faster decline of blacklist generation rate echoes the Q2 finding that attack patterns are more amenable to rule-based generalization than normal traffic patterns. Note that under append-only accumulation, any unmatched request necessarily triggers new rule generation, so the growth will not stop entirely regardless of rule quality.

These observations motivate the need for rule lifecycle management. With over 750 rules generated from just 5,000 requests, long-term deployment would require mechanisms such as offline rule refinement to merge redundant rules, prune obsolete ones, or consolidate overlapping patterns. The design of such mechanisms remains an open direction for future work.

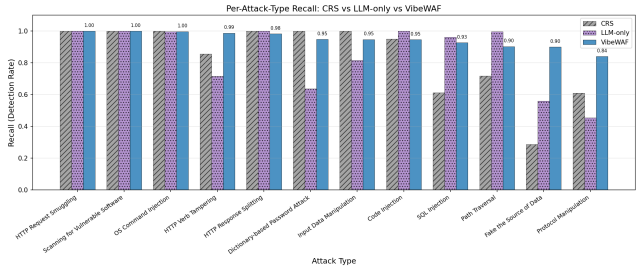

Figure 7: Per-category detection recall comparison between VibeWAF, LLM-only, and OWASP CRS.

5.7 Complementary: Per-Category Analysis

Beyond the Q1–Q4, we analyze VibeWAF's detection performance at the individual attack category level to understand where LLM-generated rules excel and where they fall short. We focus on recall because a request may be blocked by rules generated for a different attack category, making per-category precision ill-defined.

Figure 7 compares per-category recall across VibeWAF, LLM-only, and OWASP CRS. The three methods exhibit distinct strengths: CRS achieves near-perfect recall on categories such as HTTP Request Smuggling and OS Command Injection, while VibeWAF substantially outperforms CRS on Path Traversal and SQL Injection. Neither method dominates across all categories, suggesting that LLM-generated rules and expert-crafted rules capture fundamentally different attack characteristics.

6 Discussion & Future Work

Our experiments demonstrate that LLM-augmented hybrid enforcement is feasible: the feedback loop converges, with rules absorbing over 88% of traffic at steady state, and the system can adapt to novel attack categories through continued LLM analysis (Q2). However, the results also expose a fundamental tension in whitelist rule design. Q2 reveals that whitelist rules can silently admit novel attacks, while Q3 shows they simultaneously fail to cover normal traffic diversity. Since whitelist over-admission prevents novel attacks from reaching the LLM, it directly undermines the system's ability to self-correct, which is the core property that distinguishes the hybrid approach from static rule sets. We discuss key directions below.

Rule lifecycle management. The current append-only design lacks mechanisms to revise or remove rules after generation. Q2 shows that whitelist rules can silently admit novel attacks, and Q3 reveals imprecise pattern scoping in both rule types. Meanwhile, Q4 shows that continuous accumulation produces hundreds of rules from modest traffic volumes, risking redundancy and conflicts. These observations collectively point to the need for periodic rule set maintenance, potentially including matching statistics collection to identify problematic rules, merging of redundant patterns, and retirement of obsolete or overly broad rules. Designing and evaluating such mechanisms is a key direction for future work.

Architectural improvements. The remaining 5% of unmatched requests still incur seconds-long LLM latency on the critical path. Moving LLM analysis to an asynchronous sidecar would decouple throughput from inference latency, at the cost of a brief window in which novel threats go undetected. Additionally, initializing the system with existing rule sets such as OWASP CRS could accelerate convergence and reduce the early-stage attack window, as suggested by the complementary strengths observed in §5.7.

7 Related Work

Our work sits at the intersection of two areas: using LLMs to generate security rules, and automating enforcement system maintenance. We discuss each in turn.

7.1 LLM-based security rule generation.

Recent work has applied LLMs to automatically generate security rules across diverse domains, including detection rules for SIEM systems [3, 28], YARA and Semgrep rules for malicious package detection [29], IDS rules [11], firewall configurations [22], and access control policies [8, 26, 27]. These works have explored various pipeline designs to improve rule quality, including multi-stage generation, validation feedback, and iterative refinement [3, 28, 29]. However, they all generate rules from secondary sources such as threat intelligence reports, analyst descriptions, or static code artifacts. Our work explores generating rules directly from intercepted traffic in a live system.

7.2 ML-enhanced enforcement systems.

Machine learning has been widely applied to improve the detection capabilities of rule-based enforcement systems [6, 15, 18, 19]. These approaches learn statistical patterns from labeled traffic datasets to either generate new rules or augment existing rule engines. For example, Coscia et al. [6] train decision trees to automatically produce Suricata rules for DDoS detection, and Shah and Issac [19] augment Snort with an SVM plugin to reduce false positives. While effective within their training distributions, these methods require pre-collected labeled corpora and remain static after training, necessitating offline retraining to adapt to new attack patterns. Our work uses an LLM's semantic understanding to generate rules directly from individual requests without a training corpus, and operates as a continuous online loop rather than a train-then-deploy pipeline.

8 Conclusion

We explored the feasibility of LLM-driven rule generation for security enforcement systems through VibeWAF, a hybrid prototype that pairs an LLM with a rule matching engine. Our experiments on the SR-BH 2020 dataset reveal that the hybrid feedback loop converges to a 88% rule hit rate, demonstrating the viability of progressive offloading. We also identify open challenges including the asymmetry between blacklist and whitelist rule effectiveness and unsustainable rule growth under append-only accumulation. These findings provide empirical grounding for the design of hybrid enforcement systems and identify concrete directions for future work.

References

[1] Anthropic. 2025. *Claude: AI Assistant by Anthropic.* https://www.anthropic.com/claude

[2] Anonymous Authors. 2025. Large Language Models powered Malicious Traffic Detection: Architecture, Opportunities and Case Study. *arXiv preprint* (2025). arXiv:2503.18487 https://arxiv.org/abs/2503.18487

[3] CloudHunter Authors. 2024. CloudHunter: Harnessing LLMs for Automated Extraction of Detection Rules from Cloud-Based CTI. In *Proceedings of Security Conference* (2024). TODO: Complete citation information.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.

[5] Holly Cabrera. 2021. Facing cybersecurity threats, Quebec shuts down government websites for evaluation. *CBC News* (12 Dec. 2021). https://www.cbc.ca/news/canada/montreal/quebec-cybersecurity-threat-government-website-1.6283133

[6] Antonio Coscia, Vincenzo Dentamaro, Stefano Galantucci, Antonio Maci, and Giuseppe Pirlo. 2024. Automatic Decision Tree-based NIDPS Ruleset Generation for DoS/DDoS Attacks. *Journal of Information Security and Applications* 82 (2024), 103736. doi:10.1016/j.jisa.2024.103736

[7] Cyber Safety Review Board. 2022. *Review of the December 2021 Log4j Event.* Technical Report. U.S. Department of Homeland Security. https://www.cisa.gov/sites/default/files/publications/CSRB-Report-on-Log4-July-11-2022_508.pdf 19 actionable recommendations for government and industry.

[8] Sakuna Harinda Jayasundara, Nalin Asanka Gamagedara Arachchilage, and Giovanni Russello. 2024. SoK: Access Control Policy Generation from High-level Natural Language Requirements. *Comput. Surveys* 57, 4 (2024), 1–37. doi:10.1145/3706057

[9] Hamed Jelodar et al. 2025. Large Language Model (LLM) for Software Security: Code Analysis, Malware Analysis, Reverse Engineering. *arXiv preprint* (2025). arXiv:2504.07137 https://arxiv.org/abs/2504.07137

[10] Wang Lingxiang, Quanzhi Fu, Wenjia Song, Gelei Deng, Yi Liu, Dan Williams, and Ying Zhang. 2025. SAVANT: Vulnerability Detection in Application Dependencies through Semantic-Guided Reachability Analysis. *arXiv preprint arXiv:2506.17798* (2025).

[11] Shaswata Mitra, Azim Bazarov, Martin Duclos, Sudip Mittal, Aritran Piplai, Md Rayhanur Rahman, Edward Zieglar, and Shahram Rahimi. 2024. FALCON: Autonomous Cyber Threat Intelligence Mining with LLMs for IDS Rule Generation. *arXiv preprint arXiv:2508.18684* (2024). https://arxiv.org/abs/2508.18684

[12] MITRE Corporation. 2021. CVE-2021-44228: Apache Log4j2 JNDI Features Remote Code Execution Vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2021-44228. Accessed: 2026-01-15.

[13] Open Information Security Foundation. 2024. Suricata: Open Source IDS/IPS/NSM Engine. https://suricata.io

[14] OWASP CRS Project. 2021. *OWASP CRS (Official Repository).* https://github.com/coreruleset/coreruleset

[15] Kanubhai Patel and Bharat Buddhadev. 2015. Predictive Rule Discovery for Network Intrusion Detection. In *Emerging Research in Computing, Information, Communication and Applications*. Springer, 287–298. doi:10.1007/978-3-319-11227-5_25

[16] Tomás Sureda Riera, Juan-Ramón Bermejo Higuera, Javier Bermejo Higuera, José-Javier Martínez Herraiz, and Juan-Antonio Sicilia Montalvo. 2022. A new multi-label dataset for Web attacks CAPEC classification using machine learning techniques. *Computers & Security* 120 (2022), 102788. doi:10.1016/j.cose.2022.102788

[17] SELinux Project. 2024. SELinux: Security Enhanced Linux Userland Libraries and Tools. https://github.com/SELinuxProject/selinux. Accessed: 2026-01-15.

[18] Kamran Shafi and Hussein A. Abbass. 2009. An Adaptive Genetic-based Signature Learning Mechanism for Intrusion Detection. *Expert Systems with Applications* 36, 10 (2009), 12036–12043. doi:10.1016/j.eswa.2009.03.036

[19] Syed Ali Raza Shah and Biju Issac. 2018. Performance Comparison of Intrusion Detection Systems and Application of Machine Learning to Snort System. *Future Generation Computer Systems* 80 (2018), 157–170. doi:10.1016/j.future.2017.10.016

[20] Timo Stark. 2021. Mitigating the log4j Vulnerability (CVE-2021-44228) with NGINX. F5 NGINX Blog. https://www.f5.com/company/blog/nginx/mitigating-the-log4j-vulnerability-cve-2021-44228-with-nginx Accessed: 2026-01-15.

[21] Catherine Stupp. 2021. Hackers Exploit Log4j Flaw at Belgian Defense Ministry. *The Wall Street Journal* (21 Dec. 2021). https://www.wsj.com/articles/hackers-exploit-log4j-flaw-at-belgian-defense-ministry-11640020439

[22] F. Taghiyev and A. Aslanbayli. 2025. Natural Language Interface for Firewall Configuration. *arXiv preprint arXiv:2512.10789* (2025). https://arxiv.org/abs/2512.10789

[23] The Go Authors. 2024. The Go Programming Language, Version 1.23.2. https://go.dev/dl/. Released 2024-10-01. Available at https://go.dev/doc/devel/release#go1.23.2.

[24] Juan Pablo Tosso, Felipe Zipitría, and OWASP Foundation. 2024. OWASP Coraza WAF: A Golang ModSecurity Compatible Web Application Firewall Library. https://coraza.io/. GitHub: https://github.com/corazawaf/coraza.

[25] Trustwave SpiderLabs. 2002. *ModSecurity: Open Source Web Application Firewall.* https://github.com/owasp-modsecurity/ModSecurity Now maintained by OWASP.

[26] Adarsh Vatsa, Bethel Hall, and William Eiers. 2025. Exploring Large Language Models for Access Control Policy Synthesis and Summarization. *arXiv preprint arXiv:2510.20692* (2025). https://arxiv.org/abs/2510.20692

[27] Adarsh Vatsa, Pratyush Patel, and William Eiers. 2025. Synthesizing Access Control Policies using Large Language Models. *arXiv preprint arXiv:2503.11573* (2025). https://arxiv.org/abs/2503.11573

[28] Hongtai Wang et al. 2026. RulePilot: An LLM-Powered Agent for Security Rule Generation. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*. doi:10.1145/3744916.3773249 To appear.

[29] XiangRui Zhang, HaoYu Chen, Yongzhong He, Wenjia Niu, and Qiang Li. 2025. Automatically Generating Rules of Malicious Software Packages via Large Language Model. In *Proceedings of the 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 734–747. https://arxiv.org/abs/2504.17198

[30] XiangRui Zhang, XueJie Du, HaoYu Chen, Yongzhong He, Wenjia Niu, and Qiang Li. 2025. Automatically generating rules of malicious software packages via large language model. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 734–747.

A Prompt Template

```
Analyze the following HTTP request in a combined blacklist+whitelist WAF policy.

Request Details:
Method: {{.Method}}
Path: {{.Path}}
Query: {{.Query}}
Headers: {{.Headers}}
Body: {{.Body}}

Task:
1. Determine if this request is malicious or legitimate
2. Generate an appropriate ModSecurity rule:
  - If MALICIOUS: generate a DENY rule (blacklist) to block similar attacks
  - If LEGITIMATE: generate an ALLOW rule (whitelist) to permit similar requests without future LLM analysis

Output JSON format:
{
  "is_malicious": true/false,
  "attack_type": "sql_injection|xss|path_traversal|command_injection|lfi|rfi|xxe|csrf|other|
      legitimate_api|legitimate_web|legitimate_form",
  "confidence": 0.0-1.0,
  "reason": "Brief explanation of the classification",
  "rule_type": "blacklist|whitelist",
  "rule": "SecRule ... \"id:PLACEHOLDER,...\""
}

Rules for generating ModSecurity rules:
- For MALICIOUS requests (rule_type: "blacklist"):
  - Use deny action: "id:PLACEHOLDER,phase:2,deny,status:403,msg:'description'"
  - Use precise regex patterns targeting the attack vector
  - Consider false positive prevention
- For LEGITIMATE requests (rule_type: "whitelist"):
  - Use allow action: "id:PLACEHOLDER,phase:1,allow,msg:'description'"
  - Match the specific legitimate request pattern (method, path, query structure)
  - Be conservative - patterns should be specific enough to avoid allowing malicious variants

Important:
- Always use "id:PLACEHOLDER" for rule ID (system will replace with actual ID)
- Always set rule_type to match the generated rule action
- If malicious, rule_type MUST be "blacklist"
- If legitimate, rule_type MUST be "whitelist"
- Include descriptive messages in rules
```