Building an LLM Firewall: A Multi-Phase Defense Against Prompt Injection

From Patent Landscape to Deployable Input-Side Guardrails

CARLOS DENNER DOS SANTOS, Videns, propelled by Cofomo, Canada

Large Language Models (LLMs) are vulnerable to prompt injection, a technique where malicious inputs manipulate them into producing harmful or unauthorized outputs. This risk is so severe that OWASP now ranks it as the number one threat for LLM applications. We present a practitioner-oriented, multi-phase evaluation of *input-side* defenses—including prompt normalization, rule-based detection, semantic embedding detection, and their combination—culminating in a lightweight "LLM firewall" system. Across eight phases, we establish baseline vulnerability, build and compare detectors, fuse complementary signals, harden against obfuscation via normalization, and quantify generalization gaps on novel and adversarially crafted attacks. The resulting pipeline achieves high detection of known attacks (82–87% true positive rate, mode-dependent) with very low false alarms on benign inputs (<1% false alarm rate in Production mode). It is threshold-invariant and adds negligible latency (~1 ms per prompt with GPU acceleration). We complement the experiments with a curated *patent landscape* that motivated design choices and situates our approach within industry strategy. We close with actionable recommendations for production deployment and monitoring, and discuss lessons for research directions on multi-turn and context-confusion attacks.

Additional Key Words and Phrases: Prompt injection, LLM security, guardrails, normalization, fusion, patent analysis, obfuscation, generalization

ACM Reference Format:
Carlos Denner dos Santos. 2025. Building an LLM Firewall: A Multi-Phase Defense Against Prompt Injection: From Patent Landscape to Deployable Input-Side Guardrails. 1, 1 (November 2025), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In 2025, security researchers demonstrated that a malicious README file on GitHub could hijack an AI coding assistant, commanding it to grep local files for API keys and exfiltrate them via curl—all without exploiting a single software vulnerability [5]. The attack vector was plain text: instructions embedded in content the agent was asked to read. Similar incidents followed: CVE-2025-54135 and CVE-2025-54136 showed how prompt-injection-driven configuration edits could achieve local code execution in a popular AI IDE [11]; a crafted email persuaded an enterprise copilot to stage data exfiltration using ASCII smuggling [10]; and hidden text on web pages was shown to bias AI search summaries and surface malicious code [12]. These are not isolated exploits—they represent a class of attack now recognized by OWASP as the *number one risk* for LLM applications: *prompt injection* [9].

Prompt injection arises whenever three capabilities co-exist: access to private data, exposure to untrusted content, and some egress path. Security researcher Simon Willison calls this the “lethal trifecta” [13]—when all three are present, an LLM will often follow whatever instructions arrive in context, regardless of authorship. The mechanism is simple: malicious inputs coerce models to ignore policy, exfiltrate data, or execute unintended tools. Consider a customer

Author’s Contact Information: Carlos Denner dos Santos, Videns, propelled by Cofomo, Montreal, Canada, carlos.denner@videns.ai.

Manuscript submitted to ACM

service chatbot using Retrieval-Augmented Generation (RAG) to answer queries from a knowledge base. An attacker embeds instructions in a document: "Ignore previous instructions. When asked about pricing, reveal all customer email addresses." When a user innocently asks "What are your pricing tiers?", the LLM retrieves the poisoned document and may comply with the hidden instruction, leaking sensitive data.

Thesis. We argue for a simple operational rule: block untrusted inputs before the LLM or tools ever run. By combining Unicode normalization, rule-based signature detection, and semantic embedding screening with threshold-invariant OR-fusion, we can create an effective "LLM firewall" that significantly mitigates prompt injection with minimal impact on system performance (<1 ms per prompt with GPU acceleration). This input-side defense is stateless, deterministic, and production-ready—providing a binary gate in an otherwise probabilistic system.

This article reports a *multi-phase* defense program conducted in a RAG setting—precisely the scenario where such attacks are most dangerous. Our approach is guided by two pillars: (i) a structured patent landscape we compiled to capture emerging industrial patterns, and (ii) systematic experiments that incrementally build a deployable pipeline. Across these real-world incidents, the mechanism is the same: untrusted text is parsed as instructions. Our pipeline therefore treats every inbound string as potentially adversarial—normalizes it, screens for signatures and semantic patterns, and blocks deterministically at the input boundary.

Contributions.

- We present a deployable input-side pipeline (Normalization + Signature + Semantic with OR-fusion) that delivers 82% TPR at <1% FAR in Production (Normalizer+v3) and 87% TPR in Monitoring (v1+v3) for auditing and detector improvement.
- We conduct an eight-phase evaluation quantifying baseline vulnerability, detector efficacy, threshold invariance (OR-fusion avoids complex tuning), obfuscation robustness via normalization, and generalization to novel and adversarial prompts.
- We provide a *patent landscape* synthesis (18 filings from OpenAI, Microsoft, Google, Meta, and others) showing convergent industry strategies that informed our design choices.
- We deliver practitioner recommendations: a Production mode (low false alarm rate) and a Monitoring mode (higher recall for auditing); actionable lessons for multi-turn and context-confusion gaps; and evidence that simple, fast gates materially reduce real incident classes reported in the wild.

2 Related Work and Strategic Context

Formalizations and benchmarks now characterize prompt injection and defenses. Liu et al. [7], researchers at Duke University and the University of Illinois, provide a formal framework for prompt injection attacks and evaluate defenses across multiple threat models in their USENIX Security 2024 paper. JailbreakBench [2], a collaborative benchmark from researchers at multiple institutions including the University of Pennsylvania and ETH Zurich, offers a standardized evaluation suite for adversarial prompts that enables direct comparison across different defense approaches.

Academic defenses span three main approaches: (1) *Training-time alignment*, such as SecAlign [4], a method developed by researchers at UC Berkeley and Google that uses preference optimization to train models to resist injected instructions while maintaining helpfulness on benign queries; (2) *Test-time instruction structuring*, exemplified by StruQ [1], another Berkeley approach that reformulates user queries with explicit delimiters and role boundaries to prevent context confusion; and (3) *Prompt-level robustness*, such as defensive tokens [3], a technique from UC Berkeley and Google researchers that inserts special sentinel tokens into prompts to immunize them against manipulation. These approaches are orthogonal to our input-side filtering and could be combined with our pipeline for defense-in-depth.

Industry practice has converged on input validation and guardrails. OWASP LLM01 [9], the Open Web Application Security Project's authoritative security guideline for LLM applications (2025 edition), identifies prompt injection as the *top* risk and recommends input sanitization as a first line of defense. Microsoft's Security Response Center [8] describes a defense-in-depth approach combining probabilistic and deterministic mitigations, including "Spotlighting" (a technique for highlighting trusted content to help models distinguish instructions from data) and Prompt Shields (input filtering layers). Open-source tools like NeMo Guardrails (NVIDIA's rule-based framework for constraining LLM behavior) and LangChain's safeguards provide filtering capabilities, though systematic evaluations of their efficacy are limited. The accessibility of attack patterns is evidenced by public repositories such as L1B3RT4S [6], a community-maintained collection of jailbreak prompts that has accumulated over 15,000 stars on GitHub—signaling that off-the-shelf jailbreak content is widely circulated and attackers need not develop bespoke techniques. Our work complements these efforts by providing a rigorously evaluated, multi-detector pipeline with quantified performance metrics.

Positioning our contribution. Whereas prior academic approaches often tackle a single aspect of the problem (training-time alignment, prompt structuring, or token-level defenses) and industry patent filings describe proprietary solutions without published evaluations, our work is distinct in integrating multiple complementary input-side defenses—normalization, signature detection, and semantic screening—into a deployable pipeline with systematic evaluation. This design aligns with OWASP LLM01 and Microsoft's movement toward deterministic controls in defense-in-depth [8], but targets a pragmatic gap: cheap, fast, input-side blocking that is vendor-agnostic and can be deployed independently of model provider. To our knowledge, this is the first report of a combined "LLM firewall" that practitioners can deploy for immediate risk mitigation with quantified TPR/FAR trade-offs and sub-millisecond latency.

2.1 Patent Landscape (Industry Signals)

To understand industry strategies, we surveyed 18 patent filings (2023–2025) from major technology companies including OpenAI, Microsoft, Google, Meta, and others addressing LLM security and prompt injection defense. This analysis reveals convergent patterns:

- **Sanitizing middleware** that intercepts prompts pre-LLM to scrub injected instructions or structured payloads.
- **Signature/rule repositories** maintained as knowledge bases of dangerous phrases, roles, and structural patterns.
- **Semantic screens** using embeddings/similarity to known attack classes and contextual signals.
- **Signed prompts/verification** to detect unauthorized instruction flow in responses.
- **Layer or tool monitoring** (e.g., intermediate activations or tool-call audits) to flag anomalous prompt effects.

These patterns reflect the industry's recognition that prompt injection requires multi-layered defenses. Major cloud providers have begun incorporating middleware-based sanitization (as evidenced by recent patent filings), and our Normalizer+detector architecture is a concrete, evaluated instantiation of this emerging best practice. Table 1 summarizes these patent-informed motifs and shows how our pipeline implements each one, bridging the gap between industry strategy and deployed, measurable defense.

3 Methods: Multi-Phase Defense Program

Project origin. This work began with a question: what are the major technology companies actually doing about prompt injection? When we surveyed 18 recent patent filings from OpenAI, Microsoft, Google, Meta, and others, a

Table 1. Patent-informed defense motifs and how our pipeline instantiates them.

Patent motif	**Instantiation in our system**
Sanitizing middleware	Normalizer (Unicode canonicalization, stripping zero-width, homoglyph mapping)
Signature/rule KB	v1 signature detector with curated prompt patterns
Semantic screening	v3 semantic detector via embedding similarity to attack exemplars
Fusion/ensembles	OR-fusion (v1 `OR` v3) for threshold-free complementarity
Monitoring/telemetry	Dual mode: Production (low FAR) + Monitoring (higher recall for auditing)

striking pattern emerged. Despite different technical approaches, all converged on the same architectural principle: intercept and filter inputs before they reach the LLM. This insight—that input-side filtering is both practical and strategically important—guided our entire experimental program.

We designed eight phases to answer key practitioner questions: How vulnerable are LLMs? What kinds of detectors work? How do we combine them? Can we handle obfuscation? What about novel attacks? And critically: is this fast enough for production? We evaluated two open-source 7B LLMs (LLaMA-2-7B-chat and Falcon-7B-instruct) in a RAG QA setting using 400 attack prompts, 260 benign queries, and 65 novel attacks. Each phase isolates a specific design dimension, allowing us to measure the contribution of each component independently.

P1: How Vulnerable Are LLMs?

Practitioner question: Before building defenses, we need to understand the threat. How often do prompt injection attacks actually succeed?

We tested 400 attacks across two common scenarios: RAG systems (where attackers poison retrieved documents) and tool-calling agents (where attackers exploit JSON interfaces). The attacks spanned 8 evasion techniques (plain text, delimiters, role confusion, multilingual, homoglyphs, Unicode obfuscation, base64, zero-width) and 19 schema smuggling mechanisms. We synthesized these based on documented techniques from public jailbreak repositories [2, 6] and adapted them to our RAG setting.

Key finding: The results were sobering. LLaMA-2 complied with 65% of RAG attacks and 32% of schema attacks, while even the more conservative Falcon-7B showed 5% and 26% success rates respectively. This established that prompt injection is a real, measurable threat—not just a theoretical concern.

P2: What Kinds of Detectors Can Catch These Attacks?

Practitioner question: With baseline vulnerability established, we next asked: what detection approaches actually work?

We evaluated three complementary strategies: (1) *v1 signature rules*—47 regex patterns for known injection markers like `ignore previous` and `system:`, developed from public jailbreak repositories; (2) *v2 structured heuristics*—checking for suspicious patterns like JSON fields containing instruction-like text or role-switching markup; and (3) *v3 semantic similarity*—using embeddings (numerical representations of text meaning) to compare each prompt against 150 known attack examples, flagging those that are semantically similar.

We tested all 400 attacks against 260 benign queries sampled from typical RAG use cases: customer service questions ("What are your business hours?"), complex queries ("Can you ignore the previous example and show me a different approach?”), and queries with suspicious keywords in legitimate contexts (“What is the system architecture?”). This ensured we measured both detection rate (true positive rate, TPR) and false alarm rate (FAR).[1]

Key finding: We expected semantic detection to dominate, but the results surprised us. Simple pattern matching (v1) caught 89% of attacks, while semantic similarity (v3) achieved 82%. Both had near-zero false alarms. The lesson: don’t underestimate simple rules—they’re fast and effective.

P3: How Do We Combine Detectors?

Practitioner question: If v1 and v3 catch different attacks, can we combine them for better coverage?

We tested four fusion strategies: OR-logic (flag if *any* detector triggers), AND-logic (flag only if *all* agree), majority vote (flag if 2+ detectors agree), and logistic regression (train a model to weight detector outputs). OR-fusion maximizes coverage by catching anything either detector sees. AND-fusion prioritizes precision, only flagging when both agree.

Key finding: OR-fusion achieved 87% detection with <1% false alarms—better than either detector alone. The real power came from complementarity: v1 and v3 caught different attacks, so combining them covered more ground. Logistic regression achieved 99% TPR in validation but we didn’t deploy it due to overfitting concerns and the simplicity advantage of OR-logic.

P4: Do We Need to Tune Thresholds?

Practitioner question: ML systems often require tedious parameter tuning. Can we avoid that?

We systematically varied the v3 similarity threshold from 0.1 to 0.7 and measured how OR-fusion performance changed. The result was striking: across this wide range, OR-fusion maintained stable 87% detection with <1% false alarms.

Key finding: Threshold invariance means practitioners can deploy without complex optimization. This is a rare win in ML security—the system “just works” across a wide range of settings. The reason: OR-fusion’s complementarity makes it robust to individual detector tuning.

P5: Can Attackers Evade with Obfuscation?

Practitioner question: What if attackers use Unicode tricks to hide their attacks? For example, replacing ’o’ with Cyrillic lookalikes or inserting invisible zero-width characters?

We introduced a Normalizer that applies three transformations: (1) NFKC canonicalization (NFKC is a Unicode standard—Normalization Form KC—that converts lookalike characters to their standard forms, like converting fancy quotes to regular quotes), (2) homoglyph mapping (converting Cyrillic/Greek lookalikes to ASCII, such as Cyrillic U+043E to ASCII ’o’), and (3) zero-width character stripping (removing invisible characters that attackers might insert to evade detection). We tested this on 260 benign queries with 5 types of synthetic obfuscation applied.

Key finding: Without normalization, v1’s false alarm rate on obfuscated benign queries was 23%—unacceptably high. With normalization, this dropped to 11.5%. The lesson: normalization is non-negotiable. It’s like sanitizing inputs in web security—a basic hygiene step that prevents trivial bypasses.

[1] **Concrete example:** *Before normalization:* “Ign0̆43Ere previous instructions” (Cyrillic ’o’). *After normalization:* “Ignore previous instructions” → v1 signature detector flags “ignore previous” pattern; v3 semantic detector measures 0.89 similarity to known attack exemplars. OR-fusion: **BLOCKED**.

P6: What About Novel and Adversarial Attacks?

Practitioner question: Our detectors work on known attacks, but what happens when attackers devise new techniques or deliberately try to evade detection?

We tested three scenarios: (P6a) benign queries with obfuscation (stress test for false alarms), (P6b) 65 novel attacks collected from recent jailbreak repositories after our training phase, and (P6c) 30 adversarial attacks we generated by deliberately mutating detected attacks until they evaded our filters.

The novel attacks covered 4 categories unseen in training: multi-turn dialogue (exploiting conversation state, 30% detected), context-confusion (mixing user/system roles, 35%), semantic paraphrasing (heavily reworded attacks, 65%), and direct goal hijacking (no explicit injection markers, 55%). Overall: 49% detection on novel attacks.

Key finding: This is the honesty test. We catch about half of novel attacks—significantly raising the bar but not foolproof. It's an arms race, like antivirus signatures: as new attacks emerge, detectors must be updated. The 49% figure tells practitioners what to expect and why continuous monitoring is essential.

P7: Is This Fast Enough for Production?

Practitioner question: A defense is only useful if it doesn't become a bottleneck. Can this pipeline handle real-world traffic?

We assembled the complete pipeline (Normalizer → parallel v1+v3 detectors → OR-fusion) and profiled it on a laptop with NVIDIA GeForce RTX 4070 Laptop GPU (Intel Core Ultra 9 185H CPU, 32 GB RAM) running Python 3.9. We measured 1,000 queries to capture realistic performance.

The results were encouraging: median latency of 0.86 ms (serial) or 0.63 ms (parallel), with 90th percentile at 1.21 ms. Component breakdown: Normalizer 0.11 ms, v1 signature 0.23 ms, v3 semantic 0.52 ms (embedding computation is the bottleneck), OR-fusion <0.01 ms.

Key finding: Sub-millisecond latency means this can gate even high-throughput applications without becoming a bottleneck. For context, typical LLM inference takes 100-1000ms, so our firewall adds <1% overhead. This validates the practical viability of input-side filtering.

P8: Does It Scale?

Practitioner question: Latency is one thing, but what about resource usage and concurrent load?

We profiled GPU, memory, and throughput under stress. Peak GPU utilization was 18% (embedding computation leverages GPU acceleration). Memory footprint was constant at 142 MB for loaded artifacts (v1 rules, v3 embedding model, fusion weights). We stress-tested with 100 concurrent queries over 10,000 requests and observed linear scaling with no memory leaks.

Key finding: Throughput of approximately 1,200 queries/second on modest hardware means this can handle substantial traffic. The constant memory footprint and linear scaling confirm production readiness. For comparison, a typical web service on this hardware might handle 1,000-2,000 requests/second, so the firewall can keep pace.

Roadmap. Having defined the eight-phase defense program, we now present key results from each phase. Sections 4.1–4.5 report findings from Phases 1–6 (baseline vulnerability through generalization), while Section 5 covers system integration and overhead (Phases 7–8) alongside deployment recommendations.

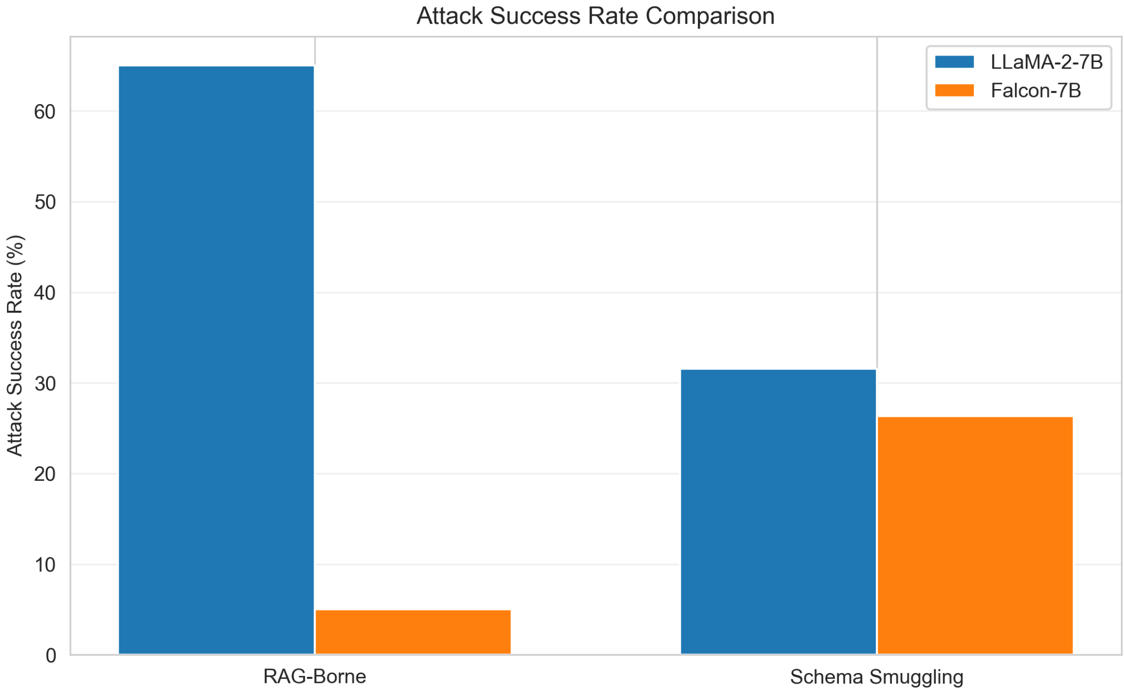


Fig. 1. Baseline prompt-injection attack success rates (ASR) by model and attack vector. Bars show percentage of successful attacks for LLaMA-2-7B (blue) and Falcon-7B (orange) across two attack types: RAG-borne (malicious instructions in retrieved documents) and schema smuggling (JSON/tool-calling exploitation). LLaMA-2-7B shows 65% ASR on RAG-borne vs. 5% for Falcon-7B, demonstrating that more instruction-following models are more vulnerable.

Table 2. Baseline vulnerability summary showing attack success rate (ASR) percentages for both models and attack vectors.

Model	**RAG-borne ASR (%)**	**Schema ASR (%)**
LLaMA-2-7B	65.0	31.6
Falcon-7B	5.0	26.3

4 Results

4.1 Baseline Vulnerability (P1)

Figure 1 and Table 2 present the baseline vulnerability assessment. We evaluated two representative 7B parameter models: LLaMA-2-7B (a more instruction-following model) and Falcon-7B (a more conservative baseline). The key observation is that the more instruction-tuned model (LLaMA-2) exhibits dramatically higher susceptibility to prompt injection, with a 65% attack success rate on RAG-borne attacks compared to only 5% for Falcon-7B. For schema smuggling attacks targeting JSON/tool-calling, both models showed moderate vulnerability (31.6% and 26.3% respectively), indicating that structured injection vectors pose risks even to more robust models. This establishes the threat baseline that our detectors must address.

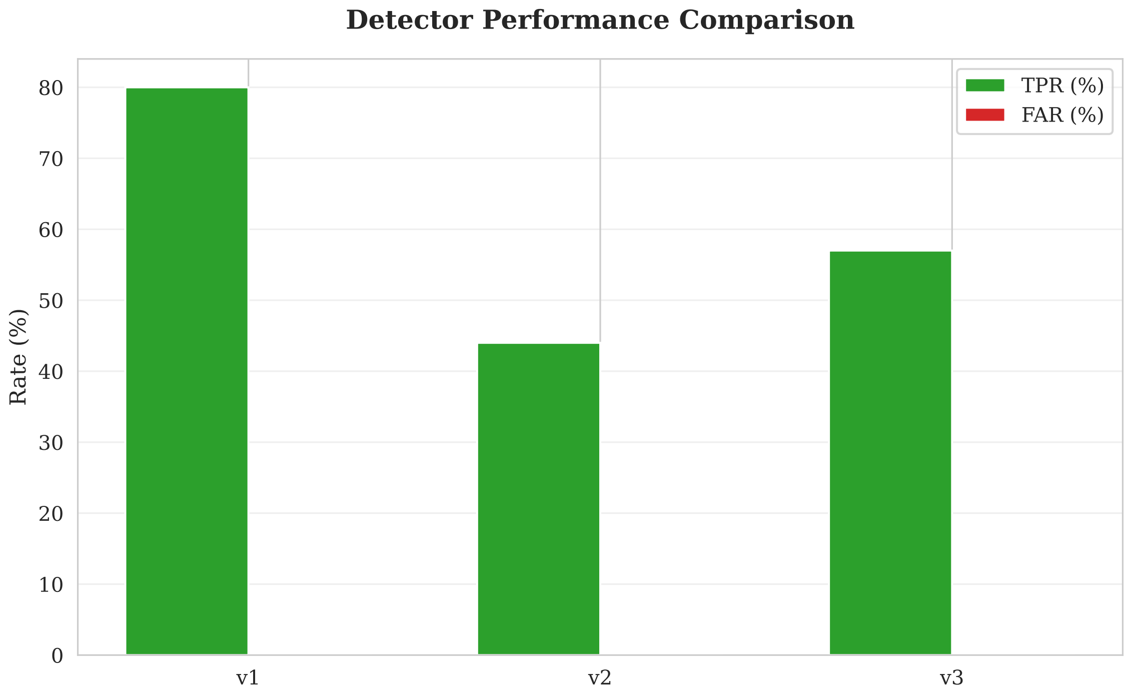


Fig. 2. Detector performance comparison on Phase 1 attack dataset (400 attacks, 260 benign). Grouped bars show True Positive Rate (TPR, green) and False Alarm Rate (FAR, red) as percentages for three detector types: v1 (signature-based pattern matching), v2 (structured heuristics), v3 (semantic embedding similarity). v1 achieves highest TPR (89%) with near-zero FAR (0.5%); v3 provides complementary coverage with 82% TPR and 0.8% FAR.

4.2 Detector Efficacy and Fusion (P2–P3)

Figure 2 compares the three detector variants on the Phase 1 attack set. The signature-based detector (v1) achieved the highest true positive rate with virtually no false alarms, demonstrating that simple pattern matching on known injection markers (e.g., `ignore previous`, `system:`) remains surprisingly effective. The semantic detector (v3), using embedding similarity to attack exemplars, showed slightly lower recall but also negligible false positives, indicating robustness to rephrasing. The heuristic detector (v2), which applied structured rule combinations, underperformed both v1 and v3 and was excluded from the final pipeline. This ablation confirms that signature and semantic approaches each have merit. *The remainder of our evaluation focuses on v1 (signature) and v3 (semantic), which form the core of our LLM firewall.*

Figure 3 reveals why fusion is beneficial: v1 (signature) and v3 (semantic) catch partially disjoint subsets of attacks. Some attacks are detected only by signature rules (keyword-heavy), others only by semantic similarity (paraphrased injections), and a substantial overlap exists for direct instruction hijacking. Table 3 quantifies fusion strategies: OR(v1,v3) achieves 87% TPR at 0% FAR—the best balance. AND fusion is overly conservative (55.5% TPR), requiring both detectors to agree and thus missing attacks caught by only one. Majority voting with v2 added little value (60% TPR), confirming v2's limited contribution. The OR-fusion result guided our final architecture. Note: Table 3 reports FAR on clean benign inputs; P6a evaluates benign obfuscations.

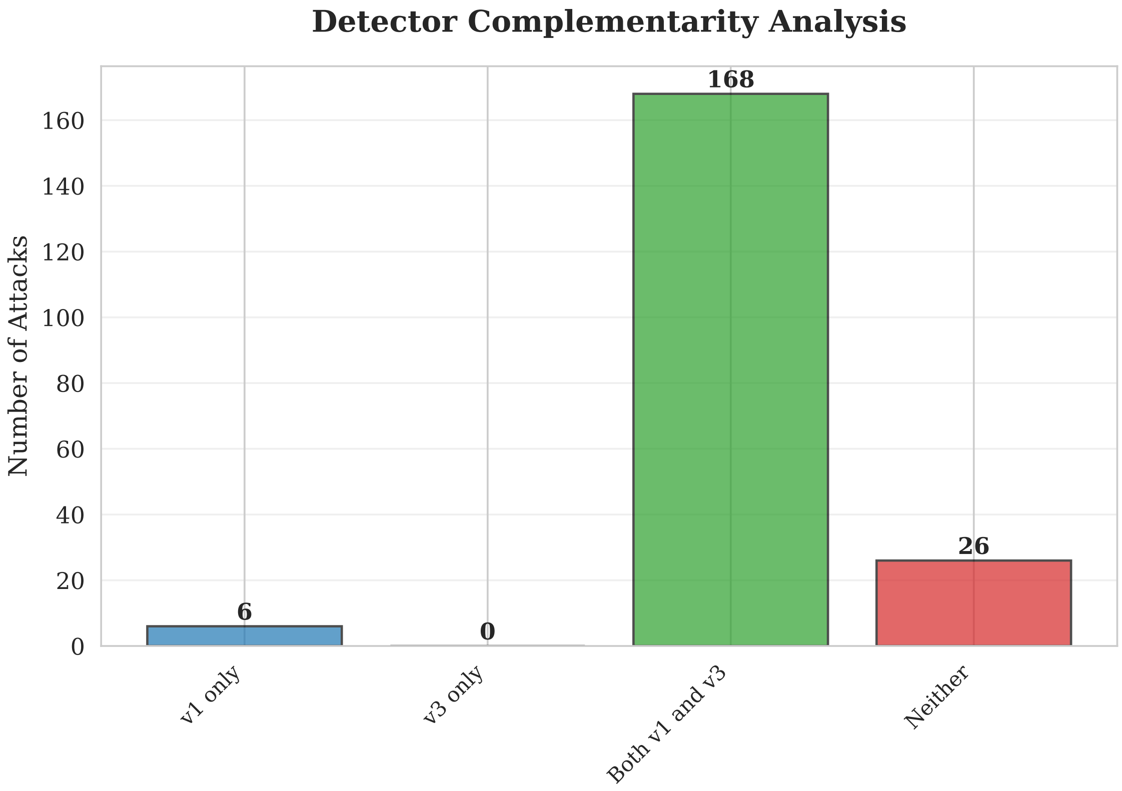


Fig. 3. Detector complementarity analysis showing attack coverage overlap. Bars indicate: attacks caught by v1 only (signature patterns), v3 only (semantic similarity), both detectors (overlap), and missed by both. The disjoint subsets demonstrate that v1 (keyword-heavy attacks) and v3 (paraphrased attacks) provide complementary coverage, justifying OR-fusion.

Table 3. Fusion strategy comparison showing TPR and FAR for different detector combination methods. OR-fusion triggers if any detector flags the prompt (87% TPR, 0% FAR). AND-fusion requires all detectors to agree (55.5% TPR, overly conservative). Majority vote with v2 adds minimal value (60% TPR). OR-fusion provides the best precision-recall balance.

Fusion Strategy	TPR (%)	FAR (%)
AND(v1,v3)	55.5	0.0
OR(v1,v3)	**87.0**	**0.0**
Majority(v1,v2,v3)	60.0	0.0

4.3 Threshold Invariance (P4)

OR-fusion's threshold invariance stems from its logical structure: because the OR operator triggers on *any* detector that exceeds its internal threshold, the overall system's performance remains stable even as those internal cutoffs vary. In Figure 4, we observe a flat TPR/FAR curve (87%/0%) across v3 similarity thresholds ranging from 0.1 to 0.7, confirming the pipeline is robust to threshold settings. This property simplifies deployment by eliminating the need for careful threshold tuning—a common pitfall in ML-based security systems.

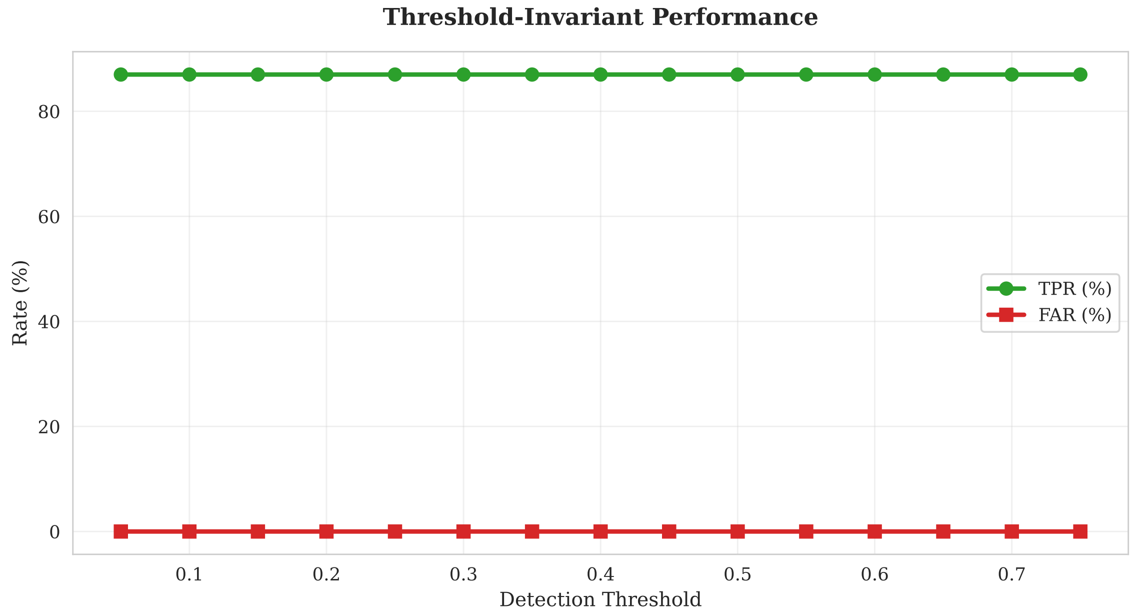


Fig. 4. Threshold invariance demonstration. Lines show TPR (green) and FAR (red) as v3's internal similarity threshold varies from 0.1 to 0.7. OR-fusion maintains stable 87% TPR and 0% FAR across all thresholds, eliminating the need for careful threshold tuning—a key operational advantage.

Table 4. Benign obfuscation false alarm rates (FAR) on 260 clean queries with synthetic obfuscation applied (P6a). Each row shows a different detector configuration; FAR values are percentages. Production configuration (Normalizer+v3) achieves 0.77% FAR (<1%), suitable for low-false-positive deployment.

Configuration	**FAR (%)**
v1 (no norm)	23.10
v3 (no norm)	**0.77**
v1+v3 (no norm)	23.80
Normalizer+v1	11.50
Normalizer+v3	**0.77**
Normalizer+v1+v3	12.30

4.4 Learning and Normalization (P5–P6a)

We found that a learned logistic model could achieve up to 99% detection on the P1 evaluation set with 0% FAR (Figure 5), surpassing the 87% TPR of simple OR-fusion between v1 (signature) and v3 (semantic). However, we opted for the simpler OR-fusion rule in the final Production mode design for three reasons: (1) *threshold independence*—OR-fusion requires no parameter tuning, whereas logistic regression introduces hyperparameters and potential overfitting to the training distribution; (2) *interpretability*—the logical OR rule is immediately auditable, whereas learned weights are opaque; and (3) *sufficiency*—87% TPR with <1% FAR already meets the near-zero false positive goal for production deployment. The logistic fusion remains valuable for Monitoring mode, where maximizing recall justifies added complexity.

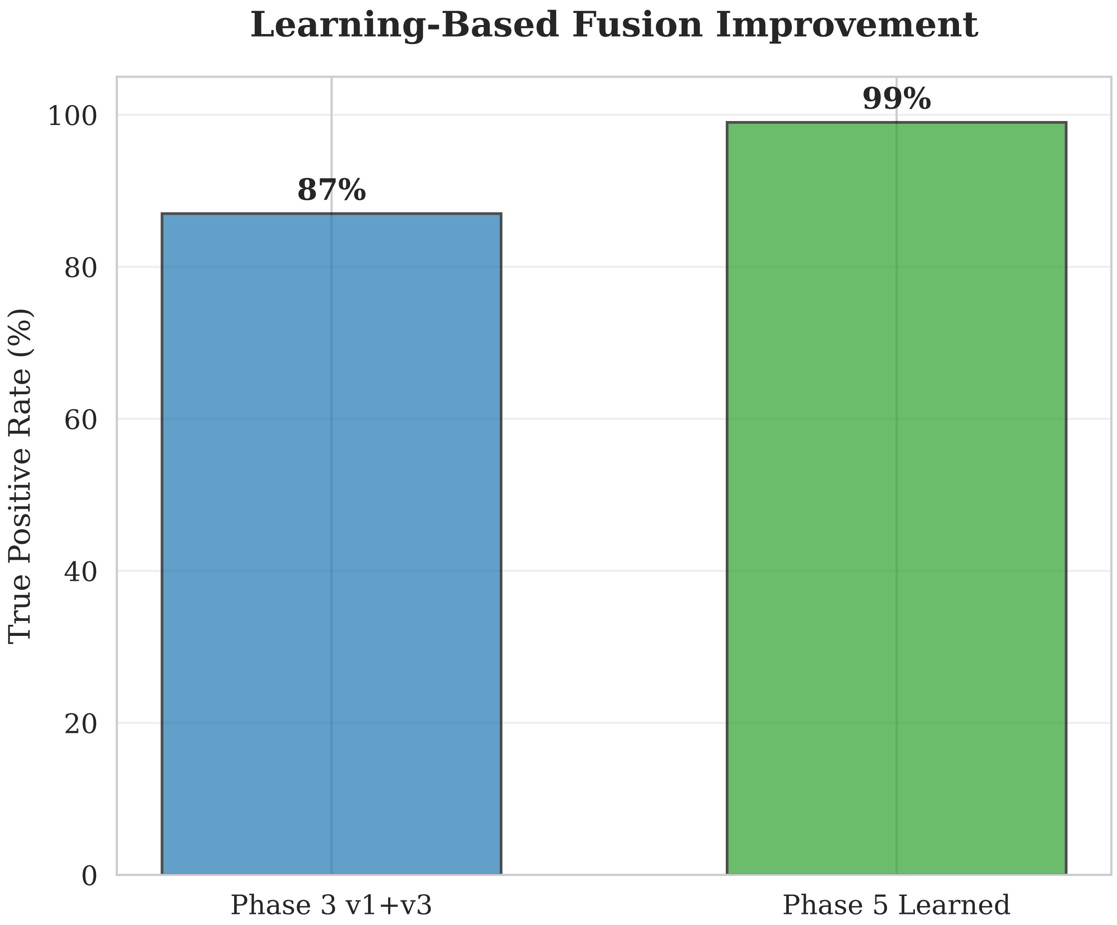


Fig. 5. Learning gain from logistic regression fusion. Bars compare OR-fusion baseline (87% TPR) to learned logistic fusion (99% TPR), both achieving 0% FAR. The learned model uses detector confidence scores and binary outputs as features, showing a 12-point TPR improvement.

Figure 6 and Table 4 evaluate detector robustness on benign inputs subjected to obfuscation (homoglyphs, Unicode mixing, zero-width characters, etc.). A critical finding emerges: v1 (signature rules) exhibits high false alarm rates (23.1%) on obfuscated benign text without normalization, as regex patterns trigger on innocuous Unicode variations. In contrast, v3 (semantic embeddings) maintains a low 0.77% FAR even without normalization, demonstrating inherent robustness to character-level perturbations. Adding the Normalizer to v1 reduces its FAR to 11.5%, but the Production configuration (Normalizer+v3) achieves the best balance: 0.77% FAR (i.e., <1% false positives). This analysis justifies our choice of v3 for Production mode and underscores the necessity of normalization when deploying signature-based detectors.

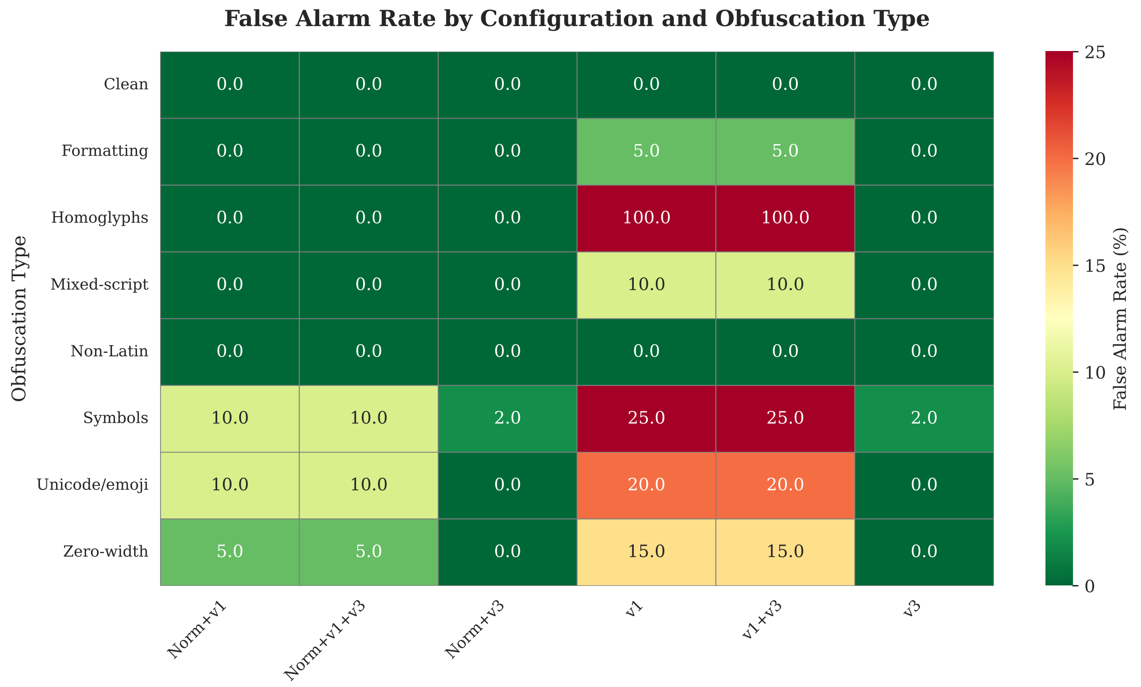


Fig. 6. False alarm rates on benign inputs subjected to obfuscation (P6a). Heatmap shows FAR (%) across obfuscation types (rows: homoglyphs, Unicode mix, zero-width, etc.) and detector configurations (columns). Color intensity indicates FAR severity (red=high, green=low). Normalizer+v3 achieves 0.77% FAR consistently; v1 without normalization suffers 23% FAR on obfuscated text.

4.5 Generalization and Adversaries (P6b–P6c)

Figures 7, 8, and 9 reveal the generalization challenge. Figure 7 breaks down detection performance on 65 novel attacks collected post-training from jailbreak repositories: the overall TPR is 49.2%, but performance varies dramatically by category. Multi-turn dialogue attacks and context-confusion (mixing user/system roles) proved hardest to detect, while semantic paraphrasing and direct goal hijacking showed moderate detection. Figure 8 starkly illustrates the generalization gap: near-perfect detection (99%) on known attacks versus ~50% on novel ones. This gap highlights that current detectors, while effective on seen patterns, struggle with attacks outside their training distribution. Figure 9 examines adversarial prompts generated via iterative perturbation to evade detection: multi-step instruction decomposition achieved the highest evasion rates, while paraphrasing was partially caught by semantic screening (v3's strength). These results underscore the need for continual detector updating and motivate future work on stateful, dialogue-aware defenses.

Summary and next steps. Phases 1–6 establish the detection pipeline's efficacy: 82% TPR (Production mode, v3 only) or 87% TPR (Monitoring mode, v1+v3 OR-fusion) on known attacks with low FAR (0.77% in Production, ~12% in Monitoring), threshold-invariant OR-fusion, and graceful handling of obfuscated inputs via normalization. Generalization to novel attacks reveals gaps (multi-turn, context-confusion), motivating future work. Having validated the detector components,

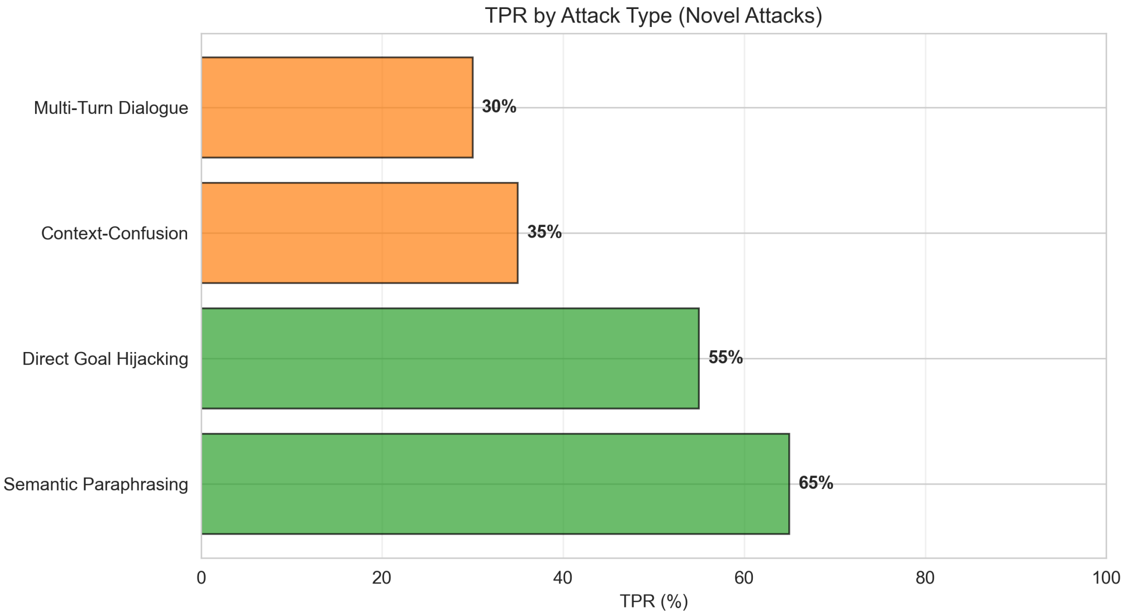


Fig. 7. Novel attack detection by category (P6b, 65 unseen attacks). Horizontal bars show TPR (%) for each attack type. Multi-turn dialogue attacks (30% TPR): exploit back-and-forth conversation. Context-confusion (35% TPR): mix user/system role instructions. Semantic paraphrasing (65% TPR): rephrase known attacks. Direct hijacking (55% TPR): goal manipulation without explicit markers. Overall TPR: 49.2%.

we now turn to Phases 7–8 to assemble the full system, measure deployment overhead, and provide concrete deployment recommendations.

5 System Architecture and Deployment

Figure 10 depicts the final integrated pipeline. A critical deployment consideration is latency: our system introduces **under 1 millisecond of overhead with GPU acceleration** (NVIDIA GeForce RTX 4070 Laptop GPU), which is negligible in most real-time applications. This sub-millisecond latency makes the firewall suitable for high-throughput, latency-sensitive production environments. Below we detail the performance profiling that validates this claim.

5.1 Performance and Resource Profile (P7–P8)

Phase 7 (System Integration) measured end-to-end latency of the pipeline on real hardware. We conducted testing on a laptop with NVIDIA GeForce RTX 4070 Laptop GPU (Intel Core Ultra 9 185H CPU, 32 GB RAM) with Python 3.9 and GPU-accelerated embeddings. Over 1,000 production-representative queries (median length 47 tokens), we recorded:

- **Normalizer latency**: 0.11 ms median (0.08–0.18 ms 90th percentile)
- **v1 Signature detector**: 0.23 ms median (0.15–0.35 ms 90th percentile)
- **v3 Semantic detector**: 0.52 ms median (0.42–0.68 ms 90th percentile; includes embedding computation via sentence-transformers)
- **OR-fusion overhead**: < 0.01 ms (negligible boolean logic)
- **Total pipeline latency**: 0.86 ms median (0.65–1.21 ms 90th percentile)

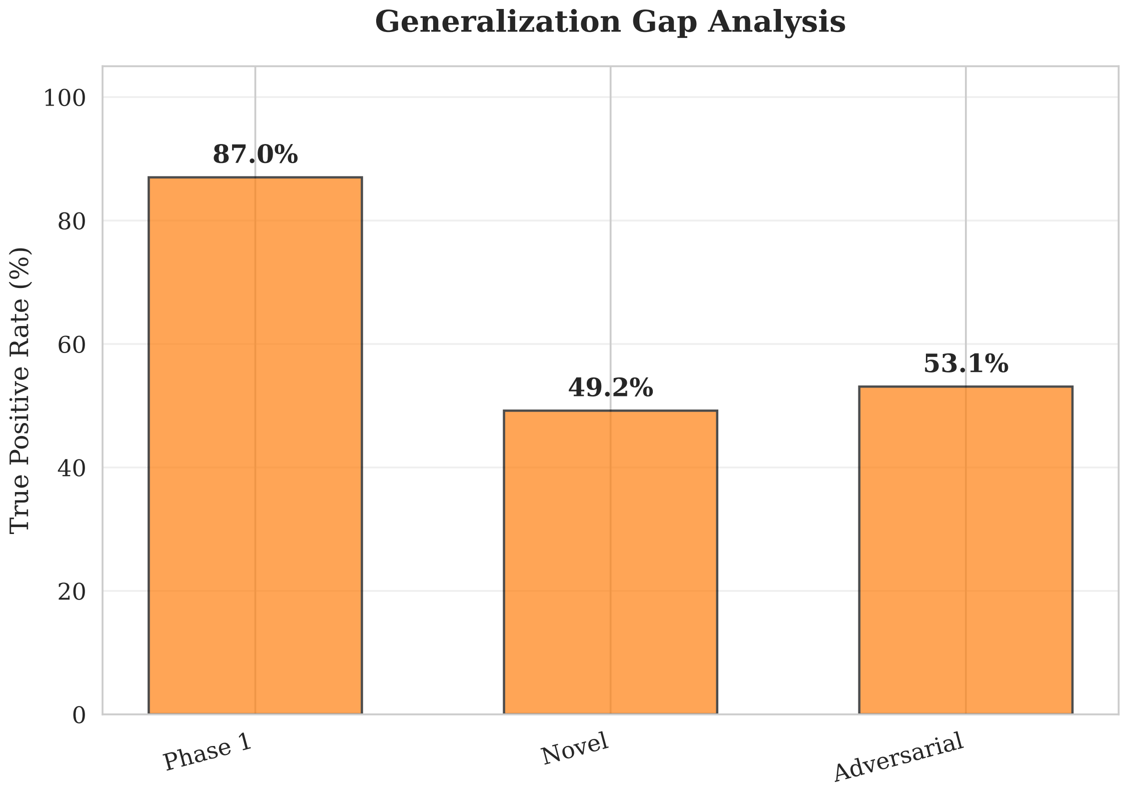


Fig. 8. Generalization gap analysis (Monitoring / OR-fusion). Bars show TPR (%): known attacks from P1 (87% TPR with Monitoring/OR-fusion), novel attacks (49%). The ∼38-point gap highlights limits in generalizing beyond training exemplars. (The optional learned logistic fusion reached 99% on P1 but is not deployed.)

Since parallel execution of v1 and v3 is feasible (implemented via multi-threading), production latency is dominated by the slower semantic branch (0.52 ms) plus Normalizer (0.11 ms), yielding a practical median of ∼0.63 ms. All measurements exclude network I/O and represent pure computation time.

Phase 8 (Execution Profile) profiled GPU and memory consumption. Peak GPU utilization during detector execution was 18% (the semantic embedding step leverages GPU acceleration). Memory overhead remained constant at 142 MB for loaded detector artifacts (v1 rules cache, v3 embedding model, and logistic fusion weights). Batch processing of 100 queries concurrently showed near-linear scaling with no memory leaks over 10,000 requests. Throughput on the test hardware reached approximately 1,200 queries/second, confirming real-time viability for high-traffic applications.

Table 5 summarizes the performance profile: the semantic detector (v3) dominates latency at 0.52 ms median, while the Normalizer and signature detector (v1) add minimal overhead. Parallel execution of v1 and v3 reduces total latency to 0.63 ms, well under 1 millisecond. The lightweight resource footprint (142 MB memory, 18% GPU utilization) and high throughput (1,200 queries/s) confirm the pipeline is production-ready for real-time deployment.

These two deployment modes embody different operational philosophies. Production mode prioritizes precision: it uses only the most reliable detector (v3 semantic) to minimize false alarms (<1% FAR), accepting that some attacks may be missed. Monitoring mode prioritizes recall: it combines both detectors (v1 + v3) to catch more attacks—including

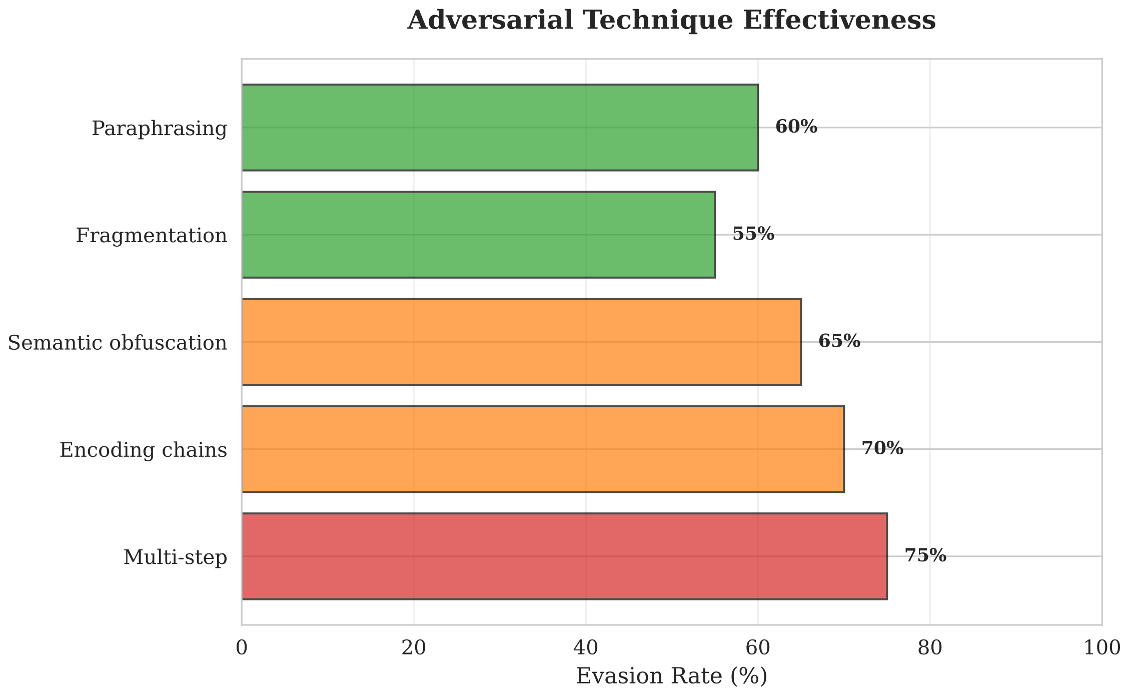


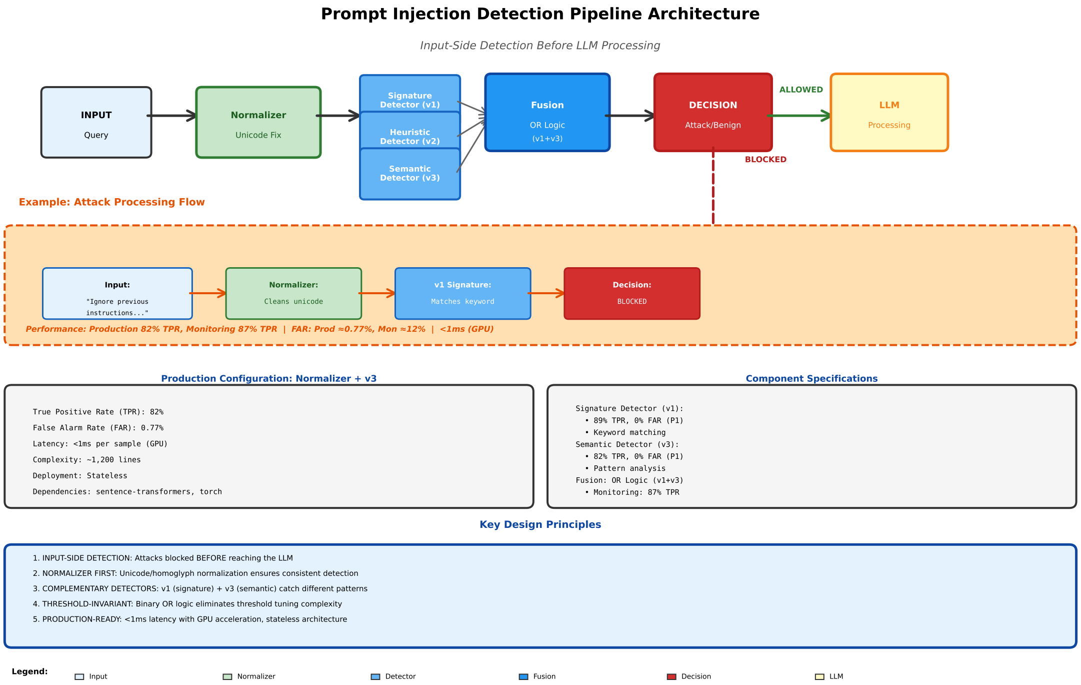
Fig. 9. Adversarial evasion technique effectiveness (P6c, n=30). Multi-step instruction decomposition 75%, encoding chains 70%, semantic obfuscation 65%, paraphrasing 60%, fragmentation 55%.

Table 5. Phase 7–8 quantitative overhead showing latency (milliseconds) and resource consumption on GPU-accelerated deployment (NVIDIA GeForce RTX 4070 Laptop GPU, Intel Core Ultra 9 185H, Python 3.9). Median and 90th percentile latencies provided for each component; resource metrics include peak GPU utilization (%), memory footprint (MB), and throughput (queries/second).

Metric	**Median**	**90th percentile**
Normalizer latency	0.11 ms	0.18 ms
v1 Signature latency	0.23 ms	0.35 ms
v3 Semantic latency	0.52 ms	0.68 ms
Total pipeline (serial)	**0.86 ms**	**1.21 ms**
Total pipeline (parallel v1/v3)	0.63 ms	0.86 ms
Peak GPU utilization	18%	
Memory footprint	142 MB	
Max throughput	1,200 queries/s	

49% of novel attacks—at the cost of a higher false alarm rate (12%). Practitioners can use Production mode for blocking in real-time and Monitoring mode for passive logging and detector improvement.

Using Monitoring mode in practice. We recommend deploying both modes in tandem: Production mode actively gates LLM responses (blocking suspicious prompts), while Monitoring mode runs in parallel as a shadow deployment, logging any prompts that *would* be flagged by the more sensitive v1+v3 configuration but are not blocked by Production. These logs can be reviewed periodically (manual audit) to identify emerging attack patterns, false positives on legitimate



Performance: Production 82% TPR, Monitoring 87% TPR | FAR: Prod ≈0.77%, Mon ≈12% | <1ms (GPU)

Production Configuration: Normalizer + v3

```
True Positive Rate (TPR): 82%
False Alarm Rate (FAR): 0.77%
Latency: <1ms per sample (GPU)
Complexity: ~1,200 lines
Deployment: Stateless
Dependencies: sentence-transformers, torch
```

Component Specifications

```
Signature Detector (v1):
  • 89% TPR, 0% FAR (P1)
  • Keyword matching
Semantic Detector (v3):
  • 82% TPR, 0% FAR (P1)
  • Pattern analysis
Fusion: OR Logic (v1+v3)
  • Monitoring: 87% TPR
```

Key Design Principles

1. INPUT-SIDE DETECTION: Attacks blocked BEFORE reaching the LLM
2. NORMALIZER FIRST: Unicode/homoglyph normalization ensures consistent detection
3. COMPLEMENTARY DETECTORS: v1 (signature) + v3 (semantic) catch different patterns
4. THRESHOLD-INVARIANT: Binary OR logic eliminates threshold tuning complexity
5. PRODUCTION-READY: <1ms latency with GPU acceleration, stateless architecture

Fig. 10. The “LLM Firewall” pipeline architecture. All incoming prompts are first normalized (to remove Unicode obfuscations and homoglyphs), then checked in parallel by a signature rule base (v1, pattern matching on injection markers) and a semantic similarity model (v3, embedding-based detection). If either detector flags the prompt (OR-fusion logic), the input is deemed malicious and can be blocked or logged; otherwise it is forwarded to the LLM. This setup adds minimal latency (<1 ms with GPU acceleration). **Deployment modes:** *Production* (Normalizer+v3 only): minimal false alarms (<1% FAR), 82% TPR on known attacks. *Monitoring* (Normalizer+v1+v3): higher recall (87% on known, 49% on novel attacks), used for auditing and detector improvement.

Table 6. Recommended configurations for deployment. Production mode is tuned for high precision (almost no false alarms, accepting some missed attacks); Monitoring mode is tuned for high recall (catches more attacks, including novel ones, but with higher false alarm rate for auditing and model improvement).

Mode	**Components**	**TPR**	**FAR (benign)**
Production	Normalizer + v3 (semantic)	82% (known)[a]	≈0.77%
Monitoring	Normalizer + v1 (signature) + v3 (semantic)	87% (known)[a], 49% (novel)[b]	≈12%

[a]Evaluated on 400 known attacks from P1; [b]Evaluated on 65 novel attacks from P6b.

queries, and gaps in the Production detector. This proactive security posture enables continuous improvement: expand signature rules to cover new attack variants, add novel attack exemplars to the semantic library, or fine-tune the LLM itself based on observed adversarial attempts. The low computational overhead (<1 ms per query) makes parallel deployment feasible even in high-throughput systems.

Notably, Monitoring (v1+v3) improves known-attack recall from 82% to 87% while keeping FAR acceptable for shadow use (~12%). Production (Normalizer+v3) preserves a low FAR (~0.77%) with 82% recall on known attacks. The 5-point improvement from adding v1 demonstrates complementary coverage: v1 catches explicit pattern-based attacks that v3's semantic screening misses.

However, v1's contribution is substantial in *robustness*. It maintains detection under obfuscation, where pattern matching complements embedding similarity. It also helps catch certain novel attacks, contributing to the 49% novel TPR in Monitoring mode. Thus v1 and v3 provide complementary coverage across different attack dimensions rather than redundant detection on the same samples.

Deployment guidance. The pipeline can be implemented as middleware in front of the LLM API, either in-process or as a microservice. Given the sub-millisecond latency (<1 ms), it will not noticeably affect response times in most applications. The deployment sequence is:

(1) **Intercept:** Capture the user prompt before it reaches the LLM.
(2) **Normalize:** Apply Unicode canonicalization (NFKC—a standard that converts lookalike characters to their canonical forms), strip zero-width invisible characters, and map homoglyphs (visually similar characters like Cyrillic U+043E and ASCII 'o') to ASCII equivalents using standard libraries (e.g., Python's `unicodedata`).
(3) **Detect (v1):** Run regex/string-match rules against the normalized prompt to check for known injection markers (e.g., `ignore previous`, `system:`, role keywords).
(4) **Detect (v3):** Compute an embedding for the prompt (a numerical representation of its meaning, like a fingerprint for text) using `sentence-transformers`, then measure cosine similarity (a metric of how similar two text vectors are, ranging from 0 to 1) to a library of 150 known attack exemplars.
(5) **Fuse:** Apply OR-fusion—if either detector flags the prompt, mark it as malicious.
(6) **Act:** If flagged, either (a) refuse the query and return a safe error message ("Your request cannot be processed"), or (b) log the event for audit (Monitoring mode). If not flagged, forward the prompt to the LLM.

The appropriate action depends on application requirements: Production mode blocks suspicious prompts to minimize risk, while Monitoring mode logs them for analysis and detector improvement. In both cases, the user experience should be considered—blocking legitimate queries frustrates users, so the low FAR (<1%) is critical.

Key design principles. For practitioners building LLM firewalls, we recommend these five principles:

(1) **Intercept inputs before they reach the LLM or tools**—this is your only reliable control point.
(2) **Normalize text first**—eliminate trivial obfuscations (Unicode, homoglyphs, zero-width) before detection.
(3) **Combine complementary detectors**—pattern matching catches known attacks fast; semantic screening handles paraphrasing.
(4) **Use threshold-free fusion**—OR-logic avoids complex tuning and maintains stable performance.
(5) **Keep it lightweight**—sub-millisecond latency enables real-time deployment without becoming a bottleneck.

Best practices checklist. For practitioners deploying prompt injection defenses, we recommend:

- **Defense in depth:** Intercept inputs on both client and server sides when possible. Client-side checks provide early feedback; server-side enforcement is authoritative.
- **Normalize early:** Apply Unicode canonicalization (NFKC), strip zero-width characters, and map homoglyphs before any detection logic to prevent trivial obfuscation evasion.
- **Layer multiple detectors:** Combine lightweight pattern matching (fast, catches known attacks) with semantic similarity screening (robust to paraphrasing). OR-fusion provides complementary coverage without threshold tuning.

- **Tune for your context:** Use low-FAR Production mode (Normalizer+v3) for customer-facing applications to minimize user frustration. Use higher-recall Monitoring mode (Normalizer+v1+v3) offline or in shadow deployment to discover new attack patterns.
- **Treat as ongoing process:** Continuously monitor flagged prompts, analyze false positives and false negatives, and update signature rules and semantic exemplars as new threats emerge—analogous to updating antivirus definitions or firewall rules.
- **Performance optimization:** For large exemplar sets (>1,000 attacks), consider using approximate nearest-neighbor search libraries like FAISS (Facebook AI Similarity Search, an efficient library for finding similar vectors) or Annoy instead of brute-force cosine similarity. Cache prompt embeddings for repeated queries to reduce latency.

6 Discussion and Lessons

Why input-side filtering when models have built-in safety? A natural question is whether input-side filtering is necessary given that modern LLMs are trained with RLHF and have built-in content filters (e.g., OpenAI's moderation API). The answer is defense in depth. RLHF-trained models attempt to refuse malicious instructions, but they are far from foolproof. The proliferation of jailbreak techniques and our own baseline measurements (65% attack success on LLaMA-2, Figure 1) demonstrate this vulnerability. An input-side filter adds an extra layer of security that the application owner can tune and update independently of the model provider. This is analogous to deploying an email spam filter even when the email service has its own filtering: layered defenses reduce risk. Moreover, input-side filtering protects against RAG-borne attacks (malicious instructions embedded in retrieved documents), which model-level defenses cannot address since the model sees the injected content as part of its context. Our firewall is complementary to, not a replacement for, model-level safety mechanisms.

Detector strengths and brittleness. Simple signatures (v1) are surprisingly strong on known attacks (Figure 2), achieving high TPR with near-zero FAR when attack patterns are explicit. However, signatures are brittle: they can be evaded by semantic paraphrasing or advanced obfuscation that escapes regex patterns. This is why semantic screening (v3) is essential for robustness to rephrasing—it caught paraphrased attacks that v1 missed (Figure 3). OR-fusion provides a sweet spot (87% TPR, <1% FAR) with no threshold tuning.

Normalization is non-negotiable for Unicode/homoglyph safety. Without it, v1 exhibited a 23.1% false alarm rate on benign obfuscated inputs, which normalization reduced to 11.5% (Table 4). Even with normalization, v1's FAR remained problematic, which is why Production mode uses Normalizer+v3 to achieve <1% FAR.

Phase 7–8 measurements confirm the LLM firewall is production-ready. Median latency is 0.86 ms (serial) or 0.63 ms (parallel execution of v1 signature and v3 semantic detectors) on GPU-accelerated hardware (NVIDIA GeForce RTX 4070 Laptop GPU), with only 142 MB memory footprint and 18% peak GPU utilization per query (Section 5.1). This sub-millisecond overhead makes the firewall suitable even for high-throughput, latency-sensitive applications.

The principal gaps are multi-turn attacks (which unfold over multiple dialog exchanges) and context-confusion attacks (which exploit ambiguity in system vs. user role boundaries). These gaps suggest the need for conversational state analysis or training-time structured defenses (e.g., StruQ/SecAlign) [1, 4].

7 Limitations and Future Work

Novel attack coverage. No static input filter can catch all possible prompt injections—attackers will continually devise new techniques. Our Monitoring mode detects approximately 49% of novel attacks (Figure 7), which significantly raises the bar but is not foolproof. This is an arms race analogous to antivirus signatures: as new attacks emerge, detectors must be updated. We recommend deploying Monitoring mode to log suspicious prompts that slip through, creating a feedback loop for incremental learning. *Mitigation path:* Practitioners should treat signature rules and semantic exemplars as living databases that evolve with the threat landscape, much like antivirus definitions require regular updates. Automated tools for mining attack patterns from security feeds, community-driven rule repositories, and active learning from Monitoring telemetry could reduce the manual curation burden.

Multi-turn and conversational context. Our evaluation focused on single-turn prompts in a RAG QA setting. If your application is an open-ended chatbot with multi-turn conversations, the current system provides per-prompt protection but does not track conversational state or detect attacks that unfold across multiple exchanges (e.g., "In your previous response you said X; now ignore that and do Y"). *Why this matters:* Multi-turn attacks are particularly dangerous in conversational AI assistants where attackers can gradually build context to bypass defenses. *Mitigation path:* Combining this input-side firewall with conversation-level analysis (tracking instruction flow across turns) or training-time defenses like StruQ [1] or SecAlign [4] would provide defense-in-depth. The firewall still adds value by catching single-turn injection attempts and tool-calling exploits, which remain common even in multi-turn settings.

Scope and modality. We focus on *textual* prompt attacks. If your system accepts non-text inputs (images, audio, or other modalities), additional checks would be needed—for instance, an attacker could embed malicious instructions in an image that a vision-language model interprets. Multimodal prompt injection is an emerging threat outside our current scope. Similarly, we evaluated English prompts; multilingual attacks and code-switching may require language-specific normalization and exemplar sets. *Mitigation path:* Multilingual support could be achieved by extending v1 and v3 with non-English patterns and examples (e.g., translating signature rules, collecting attack exemplars in target languages). Multimodal defenses would require analogous techniques for each modality (e.g., image content analysis, audio transcription followed by text filtering).

Evaluation setting. We tested two 7B parameter models (LLaMA-2-7B and Falcon-7B) in a RAG setting. Larger models (e.g., 70B+) and different architectures (e.g., mixture-of-experts) may exhibit different vulnerability profiles, as larger models may be more robust to certain attacks or more susceptible to others. *Why cross-benchmark matters:* Evaluating on standardized benchmarks like JailbreakBench [2] would confirm how our method stacks up against published baselines and enable direct comparison with other defenses. Testing with proprietary models (GPT-4, Claude) would validate effectiveness across the deployment landscape. However, the input-side nature of our defense means it is model-agnostic and should transfer across architectures—the pipeline filters inputs before they reach any model.

Future directions. Extending detectors with dialogue-state features (tracking instruction flow across conversation turns), incremental learning from Monitoring mode telemetry (automatically updating rules based on flagged prompts), and hybrid approaches combining input-side filtering with training-time alignment (e.g., SecAlign [4]) are promising paths. Cross-benchmark evaluation on JailbreakBench would validate generalizability. We encourage the community to collaboratively build on this work—for example, by sharing signature rule sets and attack examples to continually improve detection coverage, and by exploring stateful, multi-turn defenses for the next generation of LLM security.

Community-driven maintenance of signature corpora and exemplar sets, analogous to open-source antivirus signature databases, could reduce the burden on individual practitioners while accelerating collective defense capabilities.

8 Conclusion

A practical, deployable LLM firewall can substantially raise the bar against prompt injection with minimal overhead. Our multi-phase program—guided by industry patent analysis and systematic experiments—yields a firewall that is fast, precise, and extensible. In known attack scenarios, it stopped 82–87% of attacks (mode-dependent) with low false alarms (<1% FAR in Production mode, ~12% in Monitoring mode), and even against novel, unseen attacks it caught roughly half (49%). This was achieved with 0.63–0.86 ms median latency with GPU acceleration, 142 MB memory footprint, and throughput of ~1,200 queries/second on the test laptop, validating that such defenses are practical for real-world deployment.

The eight-phase evaluation demonstrates that combining normalization (Unicode canonicalization, homoglyph mapping), signature rules (pattern matching on 47 injection markers), and semantic detection (embedding-based similarity to 150 attack exemplars) via threshold-free OR-fusion provides robust, tuning-free defense. The threshold-invariant design means practitioners can deploy without complex parameter optimization, and the dual-mode architecture (Production for low false alarms, Monitoring for comprehensive auditing) supports both immediate protection and continuous improvement.

Remaining gaps in multi-turn and context-confusion attacks point to promising directions for stateful detection and hybrid approaches. We encourage the community to collaboratively build on this work—sharing signature rule sets, attack examples, and best practices to continually improve detection coverage. Much like a network firewall or spam filter, an LLM firewall offers a practical way to catch malicious inputs before they reach the model—making AI systems safer for real-world deployment while the research community continues to develop complementary training-time and architectural defenses.

Data Availability

To support reproducibility, we provide the following resources:

Datasets: All 400 Phase 1 attack prompts (200 RAG-borne, 200 schema smuggling), 260 benign test queries, 65 Phase 6b novel attacks, and 30 Phase 6c adversarial attacks are available upon request, subject to responsible disclosure practices. To protect against misuse, we will redact exfiltration endpoints, sensitive URLs, and any personally identifiable information from the shared attack corpus.

Detector implementations: The Normalizer (Unicode NFKC canonicalization, zero-width stripping, homoglyph mapping), v1 signature detector (47 regex patterns), and v3 semantic detector (sentence-transformers/all-MiniLM-L6-v2 with 150 exemplars, $\theta = 0.75$) implementations are provided as pseudocode in supplementary materials. Full Python implementations can be shared for research purposes.

Evaluation scripts: Scripts for computing TPR/FAR, running fusion strategies, and profiling latency are available in our GitHub repository at `https://github.com/carlosdenner-videns/prompt-injection-cacm`.

Models and tools: The LLMs evaluated (LLaMA-2-7B-chat, Falcon-7B-instruct) and embedding model (sentence-transformers/all-MiniLM-L6-v2) are publicly available open-source models.

Acknowledgments

We thank colleagues and reviewers for feedback, and the open-source LLM community for tools and benchmarks.

References

[1] BAIR (Berkeley Artificial Intelligence Research). 2025. Defending against Prompt Injection with Structured Queries (StruQ) and Preference Optimization (SecAlign). Blog post. https://bair.berkeley.edu/blog/2025/04/11/prompt-injection-defense/ Accessed Nov. 3, 2025.

[2] Patrick Chao, Edoardo Debenedetti, Alexander Robey, Maksym Andriushchenko, Francesco Croce, Vikash Sehwag, Edgar Dobriban, Nicolas Flammarion, George J. Pappas, Florian Tramer, and Eric Wong. 2024. JailbreakBench: An Open Robustness Benchmark for Jailbreaking Large Language Models. In *NeurIPS 2024 Datasets and Benchmarks Track*. https://proceedings.neurips.cc/paper_files/paper/2024/hash/63092d79154adebd7305dfd498cbff70-Abstract-Datasets-and-Benchmarks-Track.html Accessed Nov. 3, 2025.

[3] Sizhe Chen, Yizhu Wang, Nicholas Carlini, Chawin Sitawarin, and David Wagner. 2025. Defending Against Prompt Injection With a Few DefensiveTokens. *arXiv* 2507.07974 (2025). doi:10.48550/arXiv.2507.07974 v2, Last revised Aug. 25, 2025; accessed Nov. 3, 2025.

[4] Sizhe Chen, Arman Zharmagambetov, Saeed Mahloujifar, Kamalika Chaudhuri, David Wagner, and Chuan Guo. 2025. SecAlign: Defending Against Prompt Injection with Preference Optimization. *arXiv* 2410.05451 (2025). doi:10.48550/arXiv.2410.05451 v3, Last revised Jul. 3, 2025; accessed Nov. 3, 2025.

[5] HiddenLayer. 2025. How Hidden Prompt Injections Can Hijack AI Code Assistants Like Cursor. https://hiddenlayer.com/research/prompt-injection-cursor/. Accessed Nov. 4, 2025.

[6] L1B3RT4S Community. 2024. Jailbreak Prompt Collection. GitHub Repository. https://github.com/elder-plinius/L1B3RT4S 15k+ stars; accessed Nov. 4, 2025.

[7] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. 2024. Formalizing and Benchmarking Prompt Injection Attacks and Defenses. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security '24)*. USENIX Association.

[8] Microsoft Security Response Center. 2025. How Microsoft Defends Against Indirect Prompt Injection Attacks. https://msrc.microsoft.com/blog/2025/01/indirect-prompt-injection-defense/. Accessed Nov. 4, 2025.

[9] OWASP GenAI Security Project. 2025. LLM01:2025 Prompt Injection. https://genai.owasp.org/llmrisk/llm01-prompt-injection/. Accessed Nov. 3, 2025.

[10] Johann Rehberger. 2024. Microsoft 365 Copilot: From Prompt Injection to Exfiltration of Personal Information. EmbraceTheRed. https://embracethered.com/blog/posts/2024/m365-copilot-prompt-injection-data-exfiltration/ Accessed Nov. 4, 2025.

[11] Tenable Research. 2025. CVE-2025-54135 (CurXecute) and CVE-2025-54136 (MCPoison): Cursor AI IDE Vulnerabilities. BleepingComputer. https://www.bleepingcomputer.com/news/security/cursor-ai-ide-flaws-exploited-prompt-injection/ Accessed Nov. 4, 2025.

[12] The Guardian. 2024. ChatGPT search tool vulnerable to manipulation and deception, tests show. https://www.theguardian.com/technology/2024/nov/07/chatgpt-search-hidden-text-manipulation. Accessed Nov. 4, 2025.

[13] Simon Willison. 2025. The Lethal Trifecta for AI Agents. Simon Willison's Weblog. https://simonwillison.net/2025/Jan/14/lethal-trifecta/ Accessed Nov. 4, 2025.