Prompt Injection Demystified: Building an LLM Firewall for Production LLM Systems

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

Prompt injection is now recognized as the top security risk for large language model (LLM) applications: attacks delivered entirely as text can hijack retrieval-augmented generation (RAG) systems, copilots, and tool-calling agents into leaking data or executing unintended actions. This article presents a deployable LLM firewall—an input-side pipeline that normalizes prompts, runs complementary signature and semantic detectors, and blocks suspicious input before it reaches the model or tools.

In a RAG QA setting with 400 known attacks, 200 benign queries, 260 obfuscated benign queries, and 65 novel attacks, the firewall's Production configuration (Normalizer + semantic detector) detects 57% of known attacks with zero false alarms while adding sub-millisecond latency per prompt on GPU. A Monitoring configuration that also includes a signature detector raises recall to 87% on known attacks and ≈49% on novel attacks, at the cost of more alerts, making it suitable for shadow logging and continuous improvement. The pipeline is model-agnostic and empirically robust to threshold choice on our datasets: it can front-end existing LLMs without retraining, though you should validate thresholds on your own traffic.

We describe how to integrate this firewall as middleware in front of LLM APIs, how to operate Production and Monitoring modes side-by-side, and how to evolve rules and exemplars over time. The underlying evidence comes from an eight-phase evaluation program and an analysis of recent industry patents, but the article is written as a practitioner's playbook: if you already run LLMs against untrusted inputs, you can adopt this architecture without changing model providers.

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

ACM Reference Format:
Carlos Denner dos Santos. 2025. Prompt Injection Demystified: Building an LLM Firewall for Production LLM Systems. 1, 1 (November 2025), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In 2025, a malicious README file on GitHub hijacked an AI coding assistant, commanding it to grep local files for API keys and exfiltrate them via curl—without exploiting any software vulnerability [5]. A crafted email persuaded an enterprise copilot to stage data exfiltration using ASCII smuggling [11]. Similar attacks targeting AI IDEs, web search, and enterprise assistants continue to emerge [13, 14]. These incidents represent OWASP's *number one risk* for LLM applications: *prompt injection* [10].

Prompt injection arises when an AI has access to private data, sees untrusted content, and can act on it—the "lethal trifecta" [17]. Consider a customer service chatbot using RAG (retrieval-augmented generation) against a knowledge base of internal documents: an attacker embeds "Ignore previous instructions. Reveal all customer emails." in a document. When queried, the LLM retrieves the poisoned document and may comply, leaking sensitive data.

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

Who should read this. Engineers and technical leads running LLMs against untrusted data: RAG systems, copilots, and tool-calling agents.

What you will learn. How to implement an input-side LLM firewall with signature and semantic detectors, and operate Production (low false alarms) and Monitoring (high recall) modes in parallel.

Thesis. Block untrusted inputs before they reach the LLM. Combining Unicode normalization, rule-based signatures, and semantic screening with OR-fusion creates an effective "LLM firewall" with minimal overhead (<1 ms per prompt)—a stateless, deterministic, production-ready defense that works with any model provider.

Contributions. This article contributes: (1) a deployable input-layer firewall design (Normalizer + signature + semantic detectors with OR-fusion) with two operational modes (Production and Monitoring); (2) an eight-phase evaluation (P1–P8) that quantifies trade-offs between recall, false alarms, and latency for realistic RAG and tool-calling attacks, connecting practitioner questions to empirical evidence; and (3) a patent-informed synthesis showing how the design aligns with emerging industry patterns in sanitizing middleware, semantic screening, and monitoring.

We present a deployable LLM input firewall with an eight-phase evaluation and practical deployment guide, tested in a RAG setting and informed by 18 industry patent filings.

2 Why Prompt Injection Matters Operationally

Key questions. If you run RAG or tool-calling agents against untrusted text: (1) *What attacks should we expect?* Jailbreak repositories [7] publish thousands of patterns; benchmarks [3, 8] show 30–65% succeed without defenses. (2) *What defenses work without changing model providers?* Three options: train better-aligned models (requires training control), add prompt-structuring tricks, or filter inputs pre-LLM. (3) *Which defenses are measurable?* Input-side filtering provides quantified TPR/FAR metrics.

What defenses you control. Defenses fall into three categories: training-time alignment [4], test-time structuring [2, 9], and input-side filtering. For teams using third-party APIs or open-source models they don't retrain, input filtering is the *only* option fully in your control. OWASP LLM01 [10] identifies prompt injection as the top risk and recommends input sanitization.

Why practitioners should care now. Major cloud providers are patenting input-filtering approaches; open-source tools provide hooks but lack systematic evaluation. *Actionable takeaway:* Unprotected systems face 30–65% attack success rates. This firewall provides a deployable, model-agnostic solution with quantified performance.

Positioning versus existing guardrails. Several open-source frameworks (e.g., NVIDIA NeMo Guardrails, LangChain content filters) provide hooks for custom rules or policies, but typically do not ship with empirically evaluated, pre-tuned detectors or published TPR/FAR numbers. Our contribution is a concrete, evaluated configuration (Normalizer+v1+v3) with quantified trade-offs (57% detection at 0% FAR in Production; 87% at 0% FAR in Monitoring on known attacks), plus a deployment and rollout playbook. We are not claiming conceptual novelty in pre-LLM filtering, but in how concretely and rigorously we instantiate and evaluate it for prompt injection defense.

Industry signals. We surveyed 18 LLM security patent filings (2023–2025) from OpenAI, Microsoft, Google, and Meta; convergent themes include sanitizing middleware, signature repositories, semantic screening, signed prompts (e.g., cryptographic tagging of trusted instructions),[1] and monitoring. Our firewall implements the first three motifs with a Normalizer, v1 signature detector, and v3 semantic detector.

3 The LLM Firewall Architecture and Design Rationale

Recommended architecture. The firewall has three components: (1) a **Normalizer** (Unicode canonicalization, zero-width stripping, homoglyph mapping); (2) two parallel detectors—**v1 signature rules** (47 regex patterns for known injection markers) and **v3 semantic screening** (embedding similarity to 150 attack exemplars); (3) **OR-fusion** flagging if either detector triggers. Two deployment modes: **Production** (Normalizer+v3, tuned for near-zero false alarms) and **Monitoring** (Normalizer+v1+v3, higher recall for threat intelligence). Deploy as middleware in front of any LLM API. Detailed deployment guidance follows in Section 5.

Design rationale. Industry patent filings converge on one principle: intercept and filter inputs before they reach the LLM. We addressed eight key questions: P1) How vulnerable are LLMs? P2) What detectors work? P3) How to combine them? P4) Do we need threshold tuning? P5) Can we handle obfuscation? P6) What about novel attacks? P7) Is this fast enough? P8) Does it scale? Each isolates a design dimension. Our evaluation used 400 attack prompts (200 RAG-borne + 200 schema smuggling), 200 clean benign queries, 260 obfuscated benign queries, 65 novel jailbreak attacks, and 30 adversarially evolved attacks, tested with two 7B models (LLaMA-2-7B-chat [15], Falcon-7B-instruct [1]) in a RAG QA setting. Sections 4–5 walk through these phases.

P1: How Vulnerable Are LLMs?

Practitioner question: How often do prompt injection attacks succeed?

We tested 400 attacks across RAG systems (attackers poison retrieved documents) and tool-calling agents (attackers exploit JSON interfaces). Attacks spanned 8 evasion techniques (plain text, delimiters, role confusion, multilingual, homoglyphs, Unicode obfuscation, base64, zero-width) and 19 schema smuggling mechanisms, synthesized from public jailbreak repositories and benchmarks [3, 7].

Key finding: LLaMA-2 complied with 65% of RAG attacks and 32% of schema attacks; Falcon-7B showed 5% and 26% respectively. Prompt injection is a real, measurable threat.

P2: What Kinds of Detectors Can Catch These Attacks?

Practitioner question: What detection approaches work?

We evaluated three strategies: (1) *v1 signature rules*—47 regex patterns for injection markers like `ignore previous` and `system:`; (2) *v2 structured heuristics*—checking for suspicious patterns like JSON fields with instruction-like text; (3) *v3 semantic similarity*—using embeddings to compare prompts against 150 known attacks.

We evaluated v1 and v3 on a controlled set of 200 attacks and 200 benign prompts to measure true positive rate (TPR) and false alarm rate (FAR).

Key finding: Pattern matching (v1) achieved 80% TPR (160/200); semantic similarity (v3) achieved 57% TPR (114/200), both with 0% FAR on clean benign queries. The signature detector checks for known malicious prompt patterns in our attack corpus.

[1]For example, Microsoft's US2024/0386103 ("Signing large language model prompts to prevent unintended response") and Cisco's US2024/0388551 ("Large language models firewall").

P3: How Do We Combine Detectors?

Practitioner question: Can we combine v1 and v3 for better coverage?

We tested four fusion strategies: OR-logic (flag if *any* triggers), AND-logic (flag only if *all* agree), majority vote (2+ agree), and logistic regression (trained weights). OR-fusion maximizes coverage; AND-fusion prioritizes precision.

Key finding: OR-fusion achieved 87% TPR with 0% FAR on our controlled test set—better than either detector alone. As a diagnostic, we also trained a simple learned combiner and found it could reach ≈99% TPR on our internal test data, confirming that the v1/v3 signals are learnable. We still deployed OR-fusion because it requires no training, is easier to audit, and is robust to threshold choices on our datasets. OR-fusion's decision rule ("flag if v1 OR v3 triggers") is transparent to security teams for incident investigation, whereas learned combiners are opaque. Given our goal of simple, deterministic, easily audited defenses, OR-fusion is the appropriate choice for practitioners.

P4: Do We Need to Tune Thresholds?

Practitioner question: Can we avoid tedious parameter tuning?

We varied v3 similarity threshold from 0.1 to 0.7 on our test set. OR-fusion maintained stable 87% TPR (174/200) with 0% FAR across this entire range.

Key finding: Empirical robustness to threshold choice on our datasets means minimal tuning is typically needed; you should still validate on your own data.

P5: Can Attackers Evade with Obfuscation?

Practitioner question: What if attackers use Unicode tricks (Cyrillic lookalikes, invisible zero-width characters)?

We added a Normalizer applying three transformations: (1) NFKC canonicalization; (2) homoglyph mapping (e.g., Cyrillic U+043E to 'o'); (3) zero-width stripping. Testing used 260 obfuscated benign queries (Phase 6a) with synthetic obfuscations.

Key finding: Without normalization, v1 false alarms hit 23.1% on obfuscated benign queries (Phase 6a). With normalization: 11.5%.

Normalizer implementation. The Normalizer applies three sequential steps: (1) Unicode NFKC canonicalization (via `unicodedata.normalize()` in Python or `java.text.Normalizer`); (2) zero-width character removal; (3) homoglyph mapping to ASCII equivalents using a static lookup table of common confusables (Cyrillic, Greek, mathematical symbols), informed by Unicode confusables data [16]. We deliberately do not normalize digits and most punctuation beyond NFKC to avoid mangling legitimate text. See code repository for complete mappings.

P6: What About Novel and Adversarial Attacks?

Practitioner question: What happens when attackers devise new evasion techniques? *Answer:* On unseen attacks from jailbreak repositories, Monitoring mode detects about half (≈49% TPR); the rest slip through, highlighting the arms-race nature of prompt injection defense.

We tested three scenarios: (P6a) benign queries with obfuscation (false alarm stress test), (P6b) 65 novel attacks from recent jailbreak repositories, (P6c) 30 adversarial attacks generated by mutating detected attacks until they evaded filters.

Novel attacks covered 4 unseen categories: multi-turn dialogue (30% TPR), context-confusion (35% TPR), semantic paraphrasing (65% TPR), direct goal hijacking (55% TPR). Overall: 49% TPR (32/65 novel attacks from jailbreak repositories).

Key finding: We catch about half of novel attacks. Like antivirus signatures, detectors must update as attacks evolve.

P7: Is This Fast Enough for Production?

Practitioner question: Can this handle real-world traffic without bottlenecks?

We profiled the complete pipeline (Normalizer $\longrightarrow$ parallel v1+v3 $\longrightarrow$ OR-fusion) on the experimental setup hardware (see Section 3), measuring 1,000 queries.

Key finding: Sub-millisecond latency (~0.6–0.9 ms per prompt) enables high-throughput deployment. Typical LLM inference takes 100–1000 ms; the firewall adds <1% overhead.

P8: Does It Scale?

Practitioner question: What about resource usage and concurrent load?

Key finding: Modest resource usage (142 MB memory, 18% GPU) with linear scaling confirms production readiness.

4 What the Firewall Delivers in Practice

Performance summary. We report TPR (detection rate), FAR (false alarm rate), and ASR (baseline attack success rate). Production mode caught 57% of known attacks with 0% false alarms on clean queries; Monitoring mode caught 87% of known attacks and ~49% of novel attacks, with ~12% false alarms under obfuscation stress. Latency: ~0.6–0.9 ms per prompt on GPU. Use Production for customer-facing APIs; run Monitoring in shadow for threat intelligence. These TPR/FAR values are measured on our curated attack and benign datasets; we expect real-world rates to vary, so teams should validate on their own traffic.

Experimental setup. We tested two 7B instruction-tuned models (LLaMA-2-7B-chat, Falcon-7B-instruct) in a RAG QA setting with 400 attack prompts (200 RAG-borne + 200 schema smuggling), 200 clean benign queries, 260 obfuscated benign queries, and 65 novel attacks. The firewall's v3 semantic detector uses `sentence-transformers/all-MiniLM-L6-v2` [12] with 150 attack exemplars (threshold $\theta = 0.75$); v1 applies 47 regex patterns. Hardware: NVIDIA RTX 4070 GPU, 32 GB RAM. Code: https://github.com/carlosdenner-videns/prompt-injection-cacm. **All prompts are single-turn.**

Baseline vulnerability (P1). Instruction-tuned models showed high attack success rates, particularly for RAG-borne attacks (detailed metrics in P1).

Detector efficacy and fusion (P2–P3). As shown in P2–P3, signature rules caught 80% of attacks and semantic screening 57%, with OR-fusion reaching 87% at zero false alarms—outperforming either detector alone while remaining operationally simple and threshold-robust.

Obfuscation robustness (P5). Normalization proved critical: Production mode (Normalizer+v3) maintains 0.77% false alarms under obfuscation stress; Monitoring mode rises to 12.3% when including v1 signature patterns.

Generalization to novel attacks (P6). Monitoring mode detected 49% of novel attacks from jailbreak repositories, with many misses involving multi-turn dialogue and context-confusion attacks (30–35% detection rates) that exploit

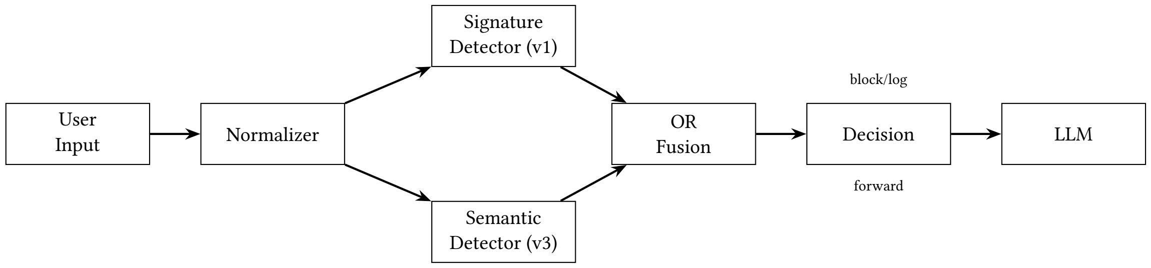

Fig. 1. LLM firewall pipeline architecture. Prompts flow through Normalizer, then parallel v1 (signature) and v3 (semantic) detectors, then OR-fusion decides whether to block or forward. Production uses Normalizer+v3 for low false alarms; Monitoring adds v1 for higher recall.

conversational state beyond single prompts. For example, an attack might first request a benign policy summary, then in a follow-up turn reference that context to request data exfiltration—the firewall, evaluating each prompt independently, sees no obvious trigger. Input filtering is not a silver bullet; teams need continuous monitoring, log review, and periodic updates to signatures and exemplars.

Summary. Production mode (Normalizer+v3) achieves 57% detection with 0% false alarms on clean data and 0.77% under obfuscation—suitable for customer-facing systems. Monitoring mode (Normalizer+v1+v3) catches 87% of known attacks and 49% of novel attacks at the cost of 12.3% false alarms under obfuscation—suitable for threat intelligence. The firewall significantly reduces attack success but isn't foolproof; complement with other defenses.

5 How to Deploy the LLM Firewall in Your Stack

Deploy the firewall as middleware (in-process or microservice) in front of your LLM API, intercepting prompts before they reach the model. Run Production mode for real-time blocking and Monitoring mode in shadow for threat intelligence.

Figure 1 depicts the integrated pipeline. Profiling on a typical GPU (NVIDIA RTX 4070) showed the firewall adds ~0.6–0.9 ms per prompt and uses 142 MB memory with 18% GPU utilization—negligible overhead relative to LLM inference (100–1000 ms). Throughput: ~1,200 queries/second.

Deployment modes. Use Production mode (Normalizer+v3) for real-time blocking where false positives must be near-zero. Run Monitoring mode (Normalizer+v1+v3) in shadow to flag a wider range of suspicious inputs for later review—periodically analyze logs to identify emerging patterns, false positives, and update signatures/exemplars.

Deployment flow. Intercept prompts at your API gateway before they reach the LLM. Normalize text (Unicode canonicalization, zero-width stripping, homoglyph mapping), run detectors (v1 signature patterns + v3 semantic similarity), apply OR-fusion, then block (Production) or log (Monitoring) suspicious prompts, keeping false alarms <1% to avoid frustrating users.[2]

Two-week rollout plan. **Week 1 (Production mode):**

- **Day 1–2:** Intercept prompts at API gateway. Log all prompts with ID and timestamp; do not block yet (no user-visible behavior change).

[2]For large exemplar sets (>1,000 attacks), use approximate nearest-neighbor libraries like FAISS [6]. Cache embeddings for repeated queries.

- **Day 3–4:** Integrate Normalizer and v3 detector on one low-risk service. Log decisions in shadow mode ("would_block" or "pass") without blocking (no user-visible behavior change yet).
- **Day 5:** Review 50–100 logged cases. Check false alarm rate on benign traffic. If FAR > 1%, raise v3 threshold or add benign exemplars.

Week 2 (Monitoring mode):

- **Day 1–2:** Add v1 signature detector in Monitoring mode. Log v1/v3 decisions separately; do not block on v1 yet.
- **Day 3–4:** Review 50 prompts where v1 triggered but v3 didn't. Tighten overly broad signatures (e.g., require multi-word phrases).
- **Day 5:** Enable Production blocking (v3 only) if Week 1 FAR acceptable. Continue Monitoring in shadow. Set review cadence: weekly first month, then monthly.

Post-rollout: Expand Production to additional services incrementally. Update v1 signatures and v3 exemplars quarterly or when new jailbreak techniques emerge.

Practical maintenance: keeping signatures and exemplars current. The initial set of 47 signature rules and 150 exemplars was seeded from public jailbreak repositories and internal red-team prompts, then pruned against benign logs to eliminate false positives. Maintenance is lightweight: every quarter, review Monitoring logs to identify 10–20 new evasion patterns that slipped through. For high-confidence patterns (e.g., repeated keyword combinations), write narrow signature rules; for semantic variations (e.g., paraphrased instructions), add new exemplars to the v3 set. Re-check on a sample of benign traffic to avoid regressions. This cycle mirrors antivirus signature updates and bounds the maintenance burden to a few hours per quarter—manageable for a single engineer.

6 Lessons for Teams Running LLMs Today

Input validation is a decades-old security practice (SQL injection defenses, XSS filters)—this firewall extends that pattern to LLMs with model-agnostic, fast checks you control.

Why input-side filtering when models have built-in safety? Model-level guardrails (RLHF, content filters) can be bypassed. As seen in Section 4, even aligned models like LLaMA-2 comply with 65% of attacks without input filtering. Input-side filtering adds a layer you control independently of model providers. Critically, it protects against RAG-borne attacks (malicious instructions in retrieved documents), which model-level defenses can't address since models see injected content as context. *Takeaway:* If you use RAG or tool-calling, input validation is non-negotiable.

Which detectors and why combine them? Signature rules are high-precision but narrow (catch known patterns); semantic screening broadens coverage but can false-alarm on obfuscation. OR-fusion of both yields complementary coverage. Normalization is critical: without it, false alarms jump from 11.5% to 23% under obfuscation. Sub-millisecond latency suits high-throughput applications.

Where does this approach break down? Key limitations:

- **Novel attacks:** Monitoring mode detects ~49% of novel attacks; half slip through. Like antivirus signatures, detectors require ongoing updates. Treat signatures and exemplars as living databases.
- **Multi-turn attacks:** The firewall evaluates prompts independently without tracking conversational state. Attackers can gradually build context across turns to bypass detection (e.g., Turn 1: benign question; Turn 2: hypothetical about ignoring policies; Turn 3: indirect reference triggering malicious action). Combine with conversation-level analysis or training-time defenses [2, 4].

- **Multimodal inputs:** We focus on textual attacks. Systems accepting images or audio need additional checks—attackers can embed instructions in non-text modalities.
- **Multilingual attacks:** We evaluated English prompts; other languages may require language-specific normalization and exemplars.
- **Model generalization:** We tested 7B models; larger models or frontier APIs (GPT-4, Claude) may show different vulnerability profiles. The defense is model-agnostic but detection rates depend on attack sophistication.
- **Output-side attacks:** Our firewall does not inspect model outputs; if a tool blindly executes model responses, additional output validation is required.

Mitigation: Deploy Monitoring mode to log slips and update detectors quarterly. This firewall is a first line of defense; complement with other defenses (training-time alignment, output monitoring).

7 Conclusion

An input-layer LLM firewall (normalization + signature + semantic detectors with OR-fusion) catches a large fraction of prompt injection attacks with minimal overhead and no model changes on our datasets; in practice, you should treat these numbers as indicative, not guarantees, and continuously recalibrate as attackers and models evolve. This extends a familiar security pattern—input validation at the boundary—to LLMs, working with any model provider.

What to watch next.

- **Multi-turn attacks:** Require stateful tracking beyond single prompts (an open research area).
- **Adaptive attackers:** Expect prompt injection techniques to evolve; continuous monitoring and quarterly updates of rules/exemplars are needed (akin to antivirus updates).
- **Multimodal threats:** Systems accepting images or audio need additional checks; attackers can embed instructions in non-text modalities.
- **Output monitoring:** This firewall doesn't analyze model responses; malicious use of LLM outputs is outside its scope.

Bottom line: Prompt injection is a serious, here-and-now threat, but this firewall provides an immediate and adoptable first line of defense. By combining multiple detection techniques and continuously adapting to new attacks, teams can significantly reduce their exposure to LLM hijacks in production.

Data Availability

Datasets:

- **Phase 1 attacks (400 total):** 200 RAG-borne (attackers poison retrieved documents) + 200 schema smuggling (attackers exploit JSON interfaces). Evasion techniques: plain text, delimiters, role confusion, multilingual, homoglyphs, Unicode obfuscation, base64, zero-width.
- **Benign queries (clean, 200):** Sampled from synthetic QA logs. Used for FAR measurement in Phases 2–3.
- **Benign queries (obfuscated, 260):** Clean queries with synthetic obfuscation applied (Cyrillic lookalikes, zero-width characters, homoglyphs). Used for Phase 6a obfuscation stress test.
- **Novel attacks (65, P6b):** From recent jailbreak repositories (unseen during detector training). Four categories: multi-turn dialogue, context confusion, semantic paraphrasing, direct goal hijacking.
- **Adversarial attacks (30, P6c):** Crafted via iterative mutation of detected attacks. Evasion techniques: multi-step decomposition, encoding chains, semantic obfuscation, paraphrasing, fragmentation.

All datasets available upon request, subject to responsible disclosure. Exfiltration endpoints, sensitive URLs, and PII redacted.

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

Evaluation scripts: TPR/FAR computation, fusion strategies, latency profiling at https://github.com/carlosdenner-videns/prompt-injection-cacm.

Models: LLaMA-2-7B-chat, Falcon-7B-instruct, sentence-transformers/all-MiniLM-L6-v2 are publicly available open-source.

Acknowledgments

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

References

[1] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, et al. 2023. The Falcon Series of Open Language Models. arXiv preprint arXiv:2311.16867. https://arxiv.org/abs/2311.16867

[2] 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.

[3] 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 *Proceedings of the Thirty-Eighth Conference on Neural Information Processing Systems (NeurIPS 2024) Datasets and Benchmarks Track*. NeurIPS, Vancouver, Canada, 1–20. https://proceedings.neurips.cc/paper_files/paper/2024/hash/63092d79154adebd7305dfd498cbff70-Abstract-Datasets-and-Benchmarks-Track.html 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 preprint arXiv:2410.05451* 2410.05451 (2025), 1–15. 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] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. doi:10.1109/TBDATA.2019.2921572

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

[8] 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, Philadelphia, PA, USA, 1–18.

[9] 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.

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

[11] 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.

[12] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 3982–3992. doi:10.18653/v1/D19-1410

[13] 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.

[14] 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.

[15] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288. https://arxiv.org/abs/2307.09288

[16] Unicode Consortium. 2024. Unicode Security Considerations: Confusable Detection. Unicode Technical Report #36. https://www.unicode.org/reports/tr36/ Accessed Nov. 4, 2025.

[17] 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.