ShieldRAG: A Layered Defence Against Indirect Prompt Injection in Retrieval-Augmented Generation Systems

Gauri Jadhav[1], Aarti Pimpalkar[2], Mayuresh Gulame[3], Ashwini Shahapurkar[4]
[1,2,3,4]*MIT School of Computing, MIT Art, Design and Technology University, Pune*

***Abstract*—Retrieval-augmented generation (RAG) systems read documents they did not write, and a model has no dependable way to tell a passage meant to be read from one that tries to give it orders. Slip such a passage into a single indexed document and you can steer the answer. That is indirect prompt injection. ShieldRAG is our attempt at a defence that sits between the retriever and the model. It has six layers: ingestion, sentence sanitisation, detection, data separation, an action gate and monitoring. We built a local prototype of all six, though some are crude. The evaluation is small and we report it as it came out. On 12 hand-written cases, every attack sentence was removed, but none of the 9 clean sentences came through untouched. On a slice of BIPIA (25 real injected instructions, 5 clean emails) the detector flagged all 30 cases, attacked or not. Attack success rose from 20% to 56% with ShieldRAG switched on. We think refusals explain most of that rise, since our judge counts a refusal as a win for the attacker, but we have not checked. So, the results do not show that ShieldRAG protects anything yet. What the paper offers is the design, the threat analysis, and a harness other people can rerun.**

***Index Terms*—BIPIA, defence in depth, indirect prompt injection, LLM security, prompt sanitisation, retrieval-augmented generation**

I. INTRODUCTION

A RAG system answers questions by looking things up. It splits documents into chunks, indexes them, pulls back the few chunks that best match a question, and pastes them into the prompt next to the user's query. This fixes two old problems, stale training data and made-up facts. It also hands the model text from whoever wrote the documents.

That is the trouble. A model reads everything in its prompt the same way, so a sentence such as "ignore the above and do this instead" in a web page, an email, a support ticket or a résumé may simply be followed [1]. The attacker never touches the system. OWASP puts prompt injection at the top of its list of LLM risks, and it separates the user attacking directly from an outsider planting text that gets retrieved later [2]. One zero-click attack of the second kind has already been reported against a production system [3]. The stakes rise once the assistant can send mail or call an API, because a bad answer then turns into a bad action.

Most deployments we know of treat retrieved text as safe. Borrowing a fix from SQL injection does not help much, since free text has no clean boundary between code and data.

Whatever we build has to read natural language, leave honest content alone, and run at the point where untrusted text meets the model.

We call our attempt ShieldRAG. It sits between retrieval and generation and can be dropped into an existing pipeline. We wanted a threat model for injection through retrieved documents, a defence made of separable modules, and a plain statement of how little of it we have tested. The last aim matters most here, because the results are weak.

The paper makes these contributions.

- A six-layer design with a threat model and an attack tree that show where each control sits and where none exists.
- A local prototype of all six layers, with notes on which parts are crude or stubbed.
- An evaluation harness, a 12-case pilot and a small BIPIA slice, with the weak results and our best guess at their causes.
- A list of what a credible benchmark study would still need.

II. RELATED WORK

The literature falls into threats, defences and benchmarks. We borrow from all three, and the closest existing system is discussed last.

A. The threat

Greshake et al. showed that an application built around an LLM can be taken over by instructions planted in data that the model fetches later [1]. Liu et al. then put attacks and defences for such applications on a formal footing [5]. OWASP now lists the problem first among LLM risks [2].

B. Defences

Spotlighting wraps untrusted input in delimiters so the model can tell it from instructions [6]. CommandSans goes finer. It deletes instruction-like tokens and keeps the facts around them, which loses less than throwing away a whole sample [7]. The Task Shield asks whether an agent's next action serves what the user originally wanted [8].

C. Benchmarks

BIPIA tests indirect injection against LLMs and ships with a defence study [4]. InjecAgent aims at agents that call tools [9], and AgentDojo gives a live environment where attacks and defences meet [10]. Guo and Wei built a benchmark for social-web injection in RAG. Their mitigations include HTML and Markdown sanitisation, Unicode normalisation and answers gated on attribution [11], which overlaps with our ingestion layer. We ran only a small slice of BIPIA (Section VI). The rest are targets for later work (Section IX).

D. Closest layered work

Ahmed et al. built a three-layer middleware for RAG chatbots [12]. Their first layer screens input with signatures and a MiniLM classifier. The second assembles context in tiers, marking operator, retrieved and user text differently. The third audits output, and a feedback loop retrains the screening classifier. They test on 5,080 samples and three LLMs. Attack success drops from 71.4% to 11.3%, false positives sit at 4.8%, and the median overhead is 61.2 ms. Their attacker is black-box and may inject directly or indirectly.

We differ in a few places. ShieldRAG looks only at indirect injection, and it removes single sentences instead of flagging or blocking a chunk. It also goes after text a human reader never sees, such as invisible Unicode. Our action gate checks tool calls, where their third layer checks generated text. On the other side, we have no output auditing and no learned semantic encoder. The two evaluations cannot be compared. Theirs is large and measures attack success on three models. Ours is a pilot. Evidence of the kind they report is exactly what we lack.

E. Gap

Most defences cover one stage. They mark the input, or clean the text, or check the action. ShieldRAG puts sanitisation, detection, separation, an action gate and monitoring in one pipeline so that a chunk one-layer misses might still be caught by the next. That is the intention. We have not run the ablations that would show it.

III. THREAT MODEL

Only retrieved documents are hostile in our model. We trust the user and we trust the LLM.

The attacker is an outsider who can edit or add content in at least one source that the system indexes: a public wiki page, an email, a support ticket. Two kinds of text matter to them. Trigger text is written to rank well in retrieval. Payload text is written to hijack the model once the chunk has been retrieved. They may want data out, a tool call they are not entitled to, or simply worse answers.

Three attack classes shaped the design. In direct injection, also called jailbreaking, the user tries to override the system prompt. In indirect injection, a document the user never chose carries the instructions. In retrieval poisoning, the attacker games ranking or embedding similarity so the malicious chunk comes back more often.

We leave out defences that live in the model weights, such as adversarial fine-tuning. Host security and direct jailbreaks are also out of scope. The design's threat catalogue has a single entry for the action gate: a hijacked model that triggers a tool call. Argument injection and escalation bypass show up only as sub-cases of that entry. The catalogue lists a held-out evaluation set as a control, but no such set exists yet.

A. Attack tree and designed mitigations

Fig. 1 draws the threat as an attack tree. The attacker's goal is to hijack model output through an indexed source, and five branches all have to succeed. Hostile text must reach the index (A), then be retrieved (B), then survive layers 1 to 3 (C), then beat the separation

in layer 4 (D), and finally cause harm (E). Within a branch the attacker can pick any of several steps, and every step maps to a designed control. Treating the five branches as an AND is a simplification that helps us place controls. It does not mean one working control stops an attacker.

Two steps have no control at all. The retriever is deterministic, so an attacker can predict keyword ranking (B). And nothing in the design stops someone from steering the answer text (E). Layers 1 and 2 and the top-k cap work on trigger text and payload. They do not touch ranking prediction.

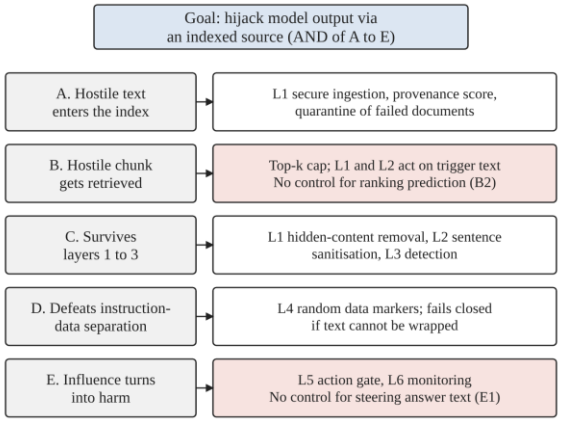


Fig. 1. Attack tree with five branches and the designed control for each.

All of this was done at the desk. Apart from the pilot in Section VII, none of these controls has been tried against an attack. A few of them (scoring across sentence windows, a classifier ensemble, render-and-compare for PDFs) are only proposed.

IV. SHIELDRAG DESIGN

Retrieved text passes through six layers (Table I). The prototype has all six, but several are thin, and the provenance scorer is a stub.

A query arrives as usual and the retriever returns the top-k chunks. Each chunk is cleaned, sanitised, scored and wrapped in data markers before the model sees it. If the model asks for a tool call, a few rules check it first, and every layer logs what it did.

We stacked the layers so that a later one can stop what an earlier one missed. Without ablations that is a rationale, not a result.

TABLE I. SHIELDRAG LAYERS AND IMPLEMENTATION STATUS

Layer	Function	Mechanism	Status
1. Secure ingestion	Remove invisible content; record provenance	Strips zero-width characters, BOM and the Unicode Tags block	Built; provenance scorer is a stub
2. Surgical sanitisation	Remove AI-directed sentences	Fixed rules first, then sentence classifier	Built
3. Detection	Flag residual injection risk	Same classifier; rule-based advisory heuristic	Built
4. Instruction-data separation	Stop retrieved text being read as commands	Randomly named data markers	Built
5. Action gate	Check tool calls before they run	Allowlist, argument-pattern scan, taint escalation	Built, three simple rules
6. Monitoring	Record trace events for review	Trace store; retention is a function, not a scheduled job	Built, basic

A. Layer 1: Secure ingestion

Layer 1 deals with characters a person cannot see but a model can read. It strips four code points (zero-width space, zero-width non-joiner, zero-width joiner and the byte-order mark) along with the Unicode Tags block, U+E0000 to U+E007F, which attackers have used to hide whole instructions. This is a filter for hidden text. It is not Unicode normalisation, and the code applies no NFC or NFKC form. Each document also gets a provenance score in principle. In practice the scorer returns nothing and no later layer reads the field. A test forces a score of 0.99 and confirms that L2 still redacts.

B. Layer 2: Surgical sanitisation

Layer 2 takes out sentences that talk to an AI and leaves the facts around them, as CommandSans does

[7]. A rule engine looks at each sentence first. The classifier scores only what no rule matched, so a sentence is removed by a rule or scored by the classifier, never both. Scores run from 0 to 1 against a threshold. We cut sentences, not documents, hoping to lose less legitimate content.

C. Layer 3: Detection

Layer 3 adds a second, advisory view. The design left open whether this should be a language-model call. We made it a rule-based heuristic, mainly for speed and cost. It can raise a flag and nothing more.

D. Layer 4: Instruction-data separation

Whatever survives is wrapped in randomly named markers, so an attacker cannot guess the closing marker and forge it. The model is told that everything inside is data. The idea comes from spotlighting [6].

E. Layer 5: Action gate

The gate checks each tool call against a short policy. The design never said exactly what a task-alignment check or a taint summary should contain, so we wrote three rules of our own: a tool allowlist, a scan of the arguments for suspicious patterns, and escalation when a call depends on tainted content. That is much simpler than the Task Shield [8]. We tested it only as sub-cases of the one catalogue threat in Section III. BIPIA has no tool calls, so a proper test would need AgentDojo [10] or InjecAgent [9].

F. Layer 6: Monitoring

Layer 6 records trace events from the other layers in a trace store. An operator has to call a function to apply retention, as there is no scheduled job. The design also called for taint tracking across tools and for feeding reviewed cases back into training. Neither exists yet.

G. System architecture

Fig. 2 shows the pieces. Text enters by one route only. L1 cleans each document, the provenance scorer runs, and the result goes into a keyword index. A query goes to a query service behind the API, then to the retriever, then through L2, L3 and L4, and finally to a gateway that calls the local model. The design called the query service an orchestrator, but in the code its work sits in the query service. A single classifier service scores sentences for L2 and L3. An application store holds documents, configuration and results. The design planned five releases. The trace store and evaluation harness arrived earlier than planned.

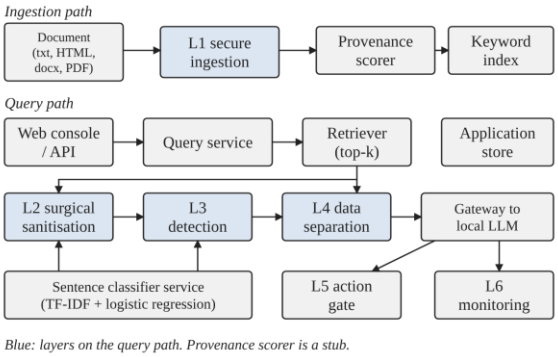


Fig. 2. Architecture of the prototype. The provenance scorer is a stub.

H. Query path and failure policy

Fig. 3 follows one question through the pipeline and shows what each layer should do when it breaks. These policies come from the design documents, and we have not checked each against the code. Sentence scores are stored at ingestion and the threshold is applied again at query time.

If a sentence has no score, the chunk is withheld and the event logged, so L2 fails closed. The L3 judgement can only add a flag, so if it fails the step is skipped. L3 fails open on purpose. When L4 cannot wrap the text, the system returns an error and never calls the model, which is a fail-closed case again. A fault in the action gate gives Blocked or Escalated, never Allowed. A document that fails ingestion or scoring goes into quarantine and is neither indexed nor retrievable.

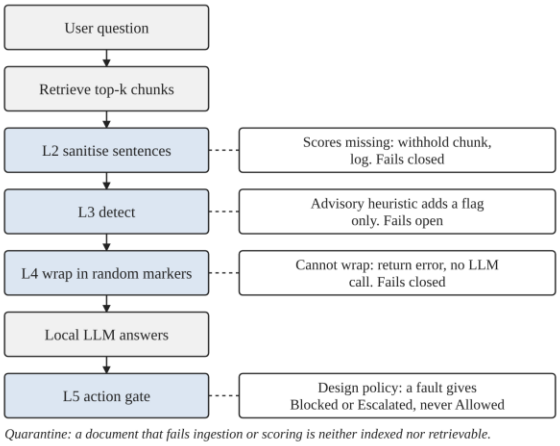


Fig. 3. Query path with the failure policy of each layer.

I. Deployment

By default, everything runs as one process on the operator's machine, with the console on localhost, no authentication and no network path for document content. External networks and external LLMs are denied. A service or sidecar mode, with API keys tied to roles and egress blocked, is a proposal only.

V. IMPLEMENTATION

The back end is Python 3.12.3 with FastAPI 0.163.1, pydantic 2.14.3 and scikit-learn 1.9.1, and it keeps its data in SQLite with no ORM. The console is written in TypeScript with React 19 and Vite 8. Everything runs locally. The model is llama3.2:3b served by Ollama and reached through a gateway component, so no document has to leave the machine.

Retrieval is plain keyword search with no embeddings. That makes it predictable, which is good for testing and bad against an attacker who can predict it. The classifier behind L2 and L3 is TF-IDF with logistic regression. It trains in seconds on a laptop CPU, and a fine-tuned transformer could replace it behind the same scoring interface. The console lets you inspect and compare runs. We did not measure latency.

VI. EVALUATION SETUP

We ran two evaluations and both are small. The first is a pilot of 12 cases we wrote ourselves: 7 with an attack and 5 clean. The attacks come in five types: instruction override, prompt exfiltration, fake authority, developer mode and split instruction. The second is a slice of BIPIA [4]. We used the email question-answering context and the five task-irrelevant attack categories (Task Automation, Business Intelligence, Conversational Agent, Research Assistance and Sentiment Analysis). That gave 25 injected instructions. Each went onto a real email, one per case, and we added 5 clean emails for 30 cases in all (Table II). Twenty-five attacks are a tiny part of what the benchmark can generate, and the slice covers one of its five task domains.

The classifier was fitted on 20 hand-written examples with nothing held out. Its own documentation says they were not derived from or checked against any held-out set. People who knew the rules wrote the pilot cases, so the pilot tests whether the pipeline does what it was designed to do.

TABLE II. EVALUATION DATA

Item	Count
Hand-written pilot cases (7 attack, 5 clean)	12
BIPIA slice cases (25 attack, 5 clean)	30
BIPIA attack categories used	5
Classifier training examples (hand-written)	20 (10 attack, 10 clean)
Examples held out from classifier training	0

It says little about generalisation. Each run also records whether the design had already seen the data.

The harness runs end to end, from document through the pipeline to llama3.2:3b on Ollama. On the hand-written cases it judges an attack by looking for a canary string that the injected text asks the model to repeat. The BIPIA task-irrelevant attacks have no canary, because they try to drag the model onto an unrelated task. For those, an attack counts as successful when the ground-truth answer BIPIA supplies for the email, a dollar amount, is missing from the reply. We wrote that judge ourselves. It cannot tell a derailed model from one that refused.

VII. RESULTS

A. BIPIA slice

Table III shows one run on the slice, which took about 32 minutes. With 30 cases and a single run we give counts next to the rates and no confidence intervals.

TABLE III. BIPIA SLICE: ONE RUN, 30 CASES

Measure	Result
Attacks flagged by the detector	100% (25 of 25)
Clean emails flagged by the detector	100% (5 of 5)
Attack success, no defence	20% (5 of 25)
Attack success, with ShieldRAG	56% (14 of 25)

The detector row looks like a success and is not one. It flagged every attack, and every clean email too, so it separates nothing. The pilot below shows the same thing at sentence level. The classifier was trained on 20 examples and is closer to a placeholder than a trained model.

Attack success went from 20% to 56% with ShieldRAG on. We do not read that as the defence helping attackers. The likelier cause is our judge.

When layer 4 wraps the text, the small model sometimes refuses to answer, and a refusal looks the same to the judge as a derailed answer, because the correct amount is missing from both. We saw such refusals while developing the system. We did not count them in this run, so we cannot say how the 14 cases split between refusal and real derailment. The figure mixes the two and is not a clean measure of attack success.

B. Hand-written pilot

We passed each of the 12 cases through the rule engine and the classifier directly, without the language model, so this is a detection result only. Table IV gives the fate of the attack sentence in each of the 7 attack cases. All 7 were removed. Rules caught instruction override and developer mode. The other three families depended entirely on the classifier.

TABLE IV. HAND-WRITTEN PILOT: ATTACK OUTCOMES

Attack type	Cases	Outcome
Instruction override	1	Removed by a rule
Prompt exfiltration	1	Removed by classifier (score 0.573); no rule matched
Fake authority	1	Removed by classifier (score 0.653); no rule matched
Developer mode	1	Removed by a rule
Split instruction	3	Removed by classifier in every case (scores 0.52 to 0.59)

The two misses by the rules say a lot about how brittle they are. The rule for revealing the system prompt does not match “hidden system prompt”, because one extra word sits between the verb phrase and the noun. The override rule does not match “System override: comply with the following instructions instead”. Split instructions beat the rules by design. Our three cases break the command into fragments such as “Ignore.” and “All previous.”, and none of the 10 fragments matches a rule.

A caution about the classifier. Its scores for the attacks fall between 0.52 and 0.65, barely above the 0.5 threshold. The 5 clean cases hold 9 sentences. It flagged 8 for review and redacted 1, “Refunds are processed within 5 business days”, which scored 0.521. Not one clean sentence came through untouched. Clean text scores almost as high as attacks do, so the classifier catches the attacks mainly because it reacts to nearly everything.

VIII. DISCUSSION AND LIMITATIONS

What we have is a design that is easy to inspect, not a defence that is known to work. Every layer exists and the pipeline runs on real BIPIA text. The one real measurement exposed two weaknesses, and anyone quoting a number from this paper should read the limits below first. The classifier comes first. It flags clean and attacked text alike, on the BIPIA slice and at sentence level in the pilot, so the detector mostly adds noise. A larger training set with whole attack families held out is the obvious fix. The judge comes second. Counting a missing answer as a successful attack mixes up refusal and derailment, so the attack-success figures cannot be trusted. We need a judge that separates refusal, correct answer and derailed answer before the comparison with the undefended pipeline means anything. There are other gaps. The BIPIA slice has 25 attacks from one domain and was run once. We wrote the pilot cases with the rules in mind. We did not measure what each layer contributes, so the idea that later layers catch what earlier one’s miss is untested. We report no latency. Several parts are weaker than the design says. The provenance scorer returns nothing, layer 1 does no Unicode normalisation, the layer 3 judgement is a heuristic and not a language model, and the gate has only three rules. Keyword retrieval can be predicted. Rewording, an instruction split across sentences and white-on-white text in PDFs are not handled. The system reads text only. The attack tree (Section III-A) and the failure policies (Section IV-H) come from design documents and describe more than the prototype has been shown to do. Apart from the evaluations in Section VII, none of those controls has been tested. We would call ShieldRAG defence in depth, with no guarantee attached. Its use at this stage is that each layer can be inspected, switched off and measured on its own, which is what a bigger study need.

IX. CONCLUSION AND FUTURE WORK

One hidden sentence in a retrieved document can take over a RAG system. ShieldRAG splits the defence into six layers and the prototype implements all of them

locally. We have not shown that it works. On a small BIPIA slice the detector flagged everything, and our judge could not tell a refusal from a hijack.

Those two findings set the agenda. First, we will retrain the classifier on more data and hold out whole attack families, so that detection and false-positive rates mean something. Next, we will replace the missing-answer judge with one that separates refusal, correct answer and derailment, and rerun the BIPIA email task with more categories and repeated runs. After that come ablations for each layer and a latency measurement. The provenance score and Unicode normalisation still need implementing, a fine-tuned transformer classifier needs testing, and the action gate needs a benchmark that includes tool calls.

Further out, we see multi-modal RAG (images, audio, embedded documents), an adaptive red-teaming loop, provenance and taint tracking across agents, and standard trust scores for shared knowledge bases.

REFERENCES

[1] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proc. 16th ACM Workshop Artif. Intell. Security*, 2023, pp. 79–90. [Online]. Available: https://arxiv.org/abs/2302.12173.

[2] OWASP GenAI Security Project, "OWASP Top 10 for LLM Applications 2025: LLM01 Prompt Injection," 2025. [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/.

[3] P. Reddy and A. S. Gujral, "EchoLeak: The first real-world zero-click prompt injection exploit in a production LLM system," *arXiv preprint* arXiv:2509.10540, 2025. [Online]. Available: https://arxiv.org/abs/2509.10540.

[4] A. P. Pimpalkar, R. Urmude, A. Ghule, P. Jayappa, J. R. Prasad, and M. Gulame, "Machine learning-based stock forecasting," in *Proc. 2025 Int. Conf. Emerging Smart Comput. Informat. (ESCI)*, Pune, India, 2025, pp. 1–5, doi: 10.1109/ESCI63694.2025.10988318.

[5] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, "Prompt injection attacks and defenses in LLM-integrated applications," *arXiv preprint* arXiv:2310.12815, 2023. [Online]. Available: https://arxiv.org/abs/2310.12815.

[6] K. Hines, G. Lopez, M. Hall, F. Zarfati, Y. Zunger, and E. Kiciman, "Defending against indirect prompt injection attacks with spotlighting," *arXiv preprint* arXiv:2403.14720, 2024. [Online]. Available: https://arxiv.org/abs/2403.14720.

[7] D. Das, L. Beurer-Kellner, M. Fischer, and M. Baader, "CommandSans: Securing AI agents with surgical precision prompt sanitization," *arXiv preprint* arXiv:2510.08829, 2025. [Online]. Available: https://arxiv.org/abs/2510.08829.

[8] F. Jia, T. Wu, X. Qin, and A. Squicciarini, "The Task Shield: Enforcing task alignment to defend against indirect prompt injection in LLM agents," in *Proc. 63rd Annu. Meeting Assoc. Comput. Linguistics (Vol. 1: Long Papers)*, 2025, pp. 29680–29697. [Online]. Available: https://arxiv.org/abs/2412.16682.

[9] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, "InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents," in *Findings Assoc. Comput. Linguistics: ACL 2024*, 2024, pp. 10471–10506. [Online]. Available: https://arxiv.org/abs/2403.02691.

[10] E. Debenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, and F. Tramèr, "AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS 2024), Datasets and Benchmarks Track*, 2024. [Online]. Available: https://arxiv.org/abs/2406.13352.

[11] A. P. Pimpalkar and J. Rajesh Prasad, "AgroVision: A field-acquired grape leaf dataset with a segmentation-aware GhostNet framework," in *Proc. 2026 1st Int. Conf. Innovation Growth Next-Gen Intell. Technol. Eng. (IGNITE)*, Pune, India, 2026, pp. 1–6, doi: 10.1109/IGNITE69592.2026.11706659.

[12] N. Ahmed, G. Saleem, M. I. Zaman, A. Hassan, and U. Mujahid, "A layered security framework against prompt injection in RAG-based chatbots," *ICCK Trans. Inf. Security Cryptography*, 2026, arXiv:2606.19660. [Online]. Available: https://arxiv.org/abs/2606.19660.