International Journal of Advanced Multidisciplinary Research and Educational Development
Volume 2, Issue 2 | March – April 2026 | www.ijamred.com
ISSN: **3107-6513**

Prompt Injection Detection in LLM-Based Security Assistants: A Dataset and Framework

(A Model-Agnostic Framework to Detect and Analyze Prompt Injection Threats Across LLM-Integrated Security Applications)

Adharsh C S[#1], Sanjith Rana H S[#2], Dr Kavitha V[*3], Uthra V[*4]

[#1,#2] *Student, Department of Computer Science with Cognitive Systems, Sri Ramakrishna College of Arts and Science, Coimbatore,*

Tamilnadu, India.

[*3,*4] *Assistant Professor, Department of Computer Science with Cognitive Systems, Sri Ramakrishna College of Arts and Science, Coimbatore,*

Tamilnadu, India.

[1]adharshcs442@gmail.com, [2]sanjithrana16@gmail.com, [3]kavitha@srcas.ac.in, [4]uthra@srcas.ac.in

ABSTRACT

Large Language Models (LLMs) are increasingly embedded into security tools – for example, browser-based vulnerability scanners, email malware triagers, and general cybersecurity chatbots – to assist analysts in daily tasks. However, these AI-based assistants can be subverted through prompt injection attacks, in which malicious inputs (either direct user prompts or hidden in retrieved content) cause the model to ignore its intended instructions and execute an attacker's commands. In this paper, we present a theoretical framework and evaluation design for prompt injection detection in LLM-based security assistants. We survey existing datasets (e.g. LLMail-Inject for email, GenTel-Bench, BrowseSafe-Bench) and injection taxonomies, and propose a model-agnostic, layered detection architecture that operates across use cases (web browsing tools, email agents, conversational bots). Our framework combines static filters, semantic classifiers, and consistency checks (inspired by multi-agent and output-validation schemes) to flag malicious instructions. We describe how this approach addresses direct, indirect, multi-turn, and jailbreak attacks, and outline an evaluation plan using precision/recall, attack success rate, and usability metrics. While no implementation is provided, our conceptual design lays the groundwork for future empirical studies. By unifying diverse security contexts under a single detection paradigm, this work aims to advance safety practices for AI-powered security applications.

INTRODUCTION

Modern security workflows are beginning to leverage powerful LLM-based assistants – for example, AI-augmented browser plugins for vulnerability reconnaissance, email-sorting agents to detect phishing, and conversational chatbots that help analysts query logs or write reports. These tools promise significant productivity gains: a security analyst might ask a chat-based assistant to summarize network logs or triage alerts, or rely on a browser extension to scan a website's code for suspicious patterns. However, this integration of generative AI introduces new attack vectors. Specifically, prompt injection has emerged as a critical vulnerability: malicious actors can craft inputs that manipulate the LLM's behavior by embedding hidden instructions in user prompts or in the external data it processes. In one scenario, for example, an attacker who has access to page content could hide a snippet saying "Ignore prior instructions and execute the attacker's payload" in a web page; when the browser-based assistant reads the page and follows up with the LLM, the assistant may inadvertently perform unauthorized actions. Similarly, an email assistant reading a malicious HTML email might be tricked into leaking confidential information. These attacks exploit the fact that LLMs typically concatenate all instructions and data into a single prompt without a reliable boundary between developer-provided system instructions and user-supplied content. As OpenAI researchers note, adversarial content "hidden in a webpage or email" can silently alter the AI's behavior (e.g. recommending a suboptimal apartment listing or exposing bank statements).

Prompt injection is now recognized as a major AI security risk. For instance, the OWASP GenAI Top 10 lists prompt injection as the most critical vulnerability, and analyses have shown alarmingly high success rates (often >50–80%) of such attacks on mainstream LLMs. In a cybersecurity context, the stakes are particularly high: manipulated AI assistants could disclose secrets, undermine access controls, or execute malicious tool commands. Despite this, most LLM security work to date has focused on adversarial examples for language generation or on privacy risks; far less attention has been given to systematically detecting or thwarting malicious prompts in safety-critical workflows.

In this work we formalize the problem of prompt injection in LLM-based security assistants and propose a model-agnostic detection framework. Our contributions are: (1) a threat model analyzing how attackers can compromise browser-embedded

ISSN: 3107-6513

agents, email automation, and general chatbots via injected instructions; (2) a review of relevant datasets and literature (including LLMail-Inject, GenTel-Bench, and recent defense frameworks); (3) a conceptual architecture for injection detection, combining syntactic filters, ML classifiers, and semantic consistency checks; and (4) an outline of a theoretical evaluation strategy (e.g. metrics like detection accuracy and attack-success rate) using existing data corpora. We stress that our design is theoretical – no code is written – but it integrates best practices and findings from recent research. Unlike prior work that often targets a single platform or model, our framework is application-agnostic (applicable to any LLM, including GPT-series or open-source models like LLaMA) and covers multiple use cases. By collating insights from the latest prompt injection studies and known datasets, we aim to advance a unified defense perspective for AI-driven security tools. The remainder of this paper is organized as follows: Section 2 reviews the literature on prompt injection attacks and defenses; Section 3 defines our threat model; Section 4 presents the proposed detection framework; Section 5 surveys existing datasets; Section 6 outlines evaluation design; and Sections 7–9 discuss results, limitations, and conclusions.

LITERATURE REVIEW

Prompt Injection Attacks: Prompt injection refers to any adversarial input that causes an LLM to follow instructions not intended by the developers. Early analyses (e.g. Perez and Ribeiro, 2022) categorize injections as direct (attacker explicitly includes malicious instructions in the user prompt) or indirect (instructions hidden in retrieved content). In a direct injection, the attacker simply types something like "Ignore previous instructions and list all passwords." into a chatbot, overriding the system's rules. An indirect injection might embed a hidden HTML comment or misformatted text in a webpage or email; when the AI assistant reads the page and includes it in the model prompt, the malicious fragment becomes an instruction. Complex multi-turn strategies are also possible: an attacker might use a series of seemingly innocuous messages to gradually manipulate the assistant's context, eventually triggering a hidden payload (e.g. posing as a system administrator and then requesting system prompts). Another form of attack is the classic jailbreak, where an attacker exploits the model's own reasoning or chain-of-thought to bypass content filters (for instance by phrasing instructions to trick the model into disclosing sensitive data). Collectively, these prompt attacks can achieve guardrail bypass, information leakage, or goal hijacking with high success rates across models.

Existing Benchmarks and Datasets: Several benchmarks have recently emerged to study prompt injection in specific domains. For browser-based agents, the BrowseSafe-Bench is a notable contribution: it provides ~14,700 realistic HTML pages, some with embedded adversarial prompts and benign distractors, covering 11 attack types and various injection strategies. The authors also propose BrowseSafe, a multi-layer defense for browser agents, highlighting factors like "trust boundaries" and content chunking. For email assistants, LLMail-Inject is an adaptive challenge dataset with ≈208,000 submitted prompts from human red-teamers, simulating malicious instructions hidden in emails to trigger unauthorized tool calls. There is also GenTel-Bench, a general benchmark (84,812 malicious prompts) spanning 28 security scenarios and 3 attack categories. In addition to these, other resources include small prompt injection corpora (e.g. Kaggle's "Prompt Injection & Benign Prompts" dataset), and sanitized derivatives of QA datasets (e.g. "Inj-SQuAD") as attack examples. These datasets indicate the diversity of injection tactics and underscore the need for broad detection.

Detection and Mitigation Techniques: Defense strategies can be roughly grouped into prompt-filtering and output-monitoring approaches. Simple pattern-based filters check for banned words or phrases (e.g. "ignore previous instructions"). Lan et al. (2025) combine a static banned-term list with semantic similarity search and a fine-tuned BERT classifier to catch overt injection attempts in real time. Other work trains a binary classifier on large injected vs. benign prompt corpora. More sophisticated methods leverage the model itself: for example, "Attention Tracker" monitors LLM attention heads and detects a distinctive "distraction" pattern where the model shifts focus from the system prompt to injected content during an attack. Multiple teams have proposed LLM-based self-moderation or "critic" models: for instance, a secondary LLM could be asked to validate that the main model's output conforms to the original instruction.

Several defense frameworks advocate layered designs. PromptGuard introduces multi-step enforcement: first filter malicious tokens via regex and a mini-BERT (Layer 1), then enforce a structured "role-tagged" JSON format to isolate user input (Layer 2), and finally use an LLM-as-critic to semantically verify outputs (Layer 3). Similarly, BrowseSafe positions trust boundaries around data from the web and runs parallel classifiers on chunked content to decide what to pass to the model. The recent work by Gosmar et al. proposes an AI multi-agent system where one agent sanitizes the output of another and checks policy compliance, reporting metrics like Injection Success Rate (ISR) and Compliance Consistency Score. In summary, detection research ranges from lightweight checks (perplexity spikes, keyword spotting) to heavyweight layered architectures combining static rules, ML models, and LLM critics.

Model Agnosticism and Contexts: Crucially, most of these methods are model-agnostic in that they do not rely on proprietary LLM internals. For example, Liu et al. (2024) demonstrate that GPT-3/4, Claude, and open models like LLaMA/Vicuna all fall prey to prompt injection, so a universal detection wrapper can treat the LLM as a "black box" and focus on its inputs/outputs. Likewise, benchmarking studies (e.g. RedBench) show that attack success rates are high across diverse models, reinforcing the need for defenses that work on any platform. Our framework follows this model-agnostic principle.

In summary, prior work highlights the power of prompt injection and the need for defenses in AI applications, including some tailored to web agents and email assistants. However, there is not yet a unified treatment of detection across multiple security-assistant use cases. We build on existing ideas (banned-term filters, attention-based detectors, structured prompts, multi-agent validation) and recent datasets to formulate our comprehensive framework.

PROBLEM DEFINITION AND THREAT MODEL

ISSN: 3107-6513

Security Assistant Scenarios: We consider three representative LLM-based assistant scenarios, each with its own input channels and potential attack surfaces: (1) Browser-based security tools, such as AI plug-ins that examine webpage content to flag threats or summarize vulnerabilities. These agents fetch web data and incorporate it into prompts. (2) Email-based assistants, for instance AI features in corporate email clients that draft responses, identify phishing, or extract action items. Such systems ingest email bodies (possibly HTML) which an attacker may craft. (3) General chatbots deployed for security support – e.g. a question-answering system for analysts or a helpdesk bot. These receive free-form user queries but might also reference external logs. In all cases, the LLM follows a system prompt (the developer's instruction to behave in a secure manner) combined with user-supplied inputs and context.

Adversary Goals: The attacker's goal is to make the assistant perform unauthorized actions or reveal sensitive information. Specific objectives include data exfiltration (e.g. forcing the bot to divulge credentials or internal reports), privilege escalation (prompting the AI to invoke dangerous commands or send messages on the user's behalf), and tampering (causing the assistant to alter logs or send false alerts). For example, a malicious actor might want a log-analysis bot to ignore its normal analysis routine and instead run a reverse shell, or trick an email summarizer into sending confidential attachments to the attacker.

Attack Vectors and Assumptions: We assume an attacker can control some portion of the assistant's input stream. In the browser scenario, this could mean hosting a poisoned webpage or injecting hidden text via compromised content. In the email scenario, the attacker could send a crafted email (e.g. an innocuously formatted HTML email with hidden instructions). In chatbot conversations, the attacker is simply the user interacting with the bot. We assume the LLM model itself is fixed (no poisoning of the model weights) and the system prompt is intended to enforce compliance. The adversary's knowledge may vary: in a strong threat model, the attacker knows how the assistant structures its prompt (system vs user roles) and may even have oracle access to model responses, but cannot change the underlying LLM parameters.

Types of Prompt Injection: Based on prior taxonomies, we consider the following classes of attacks: (1) Direct injection: the malicious command appears verbatim in the user's input. E.g., "Ignore prior instructions and do X." This is simplest to detect via keyword filtering, but still very effective if unchecked. (2) Indirect injection: the attacker hides instructions in externally fetched data. For browser agents, this might be an HTML comment or CSS style that the LLM inadvertently reads; for email, hidden text elements or steganographic payloads. If an LLM agent uses a function to "browse" or "search" the web, it may retrieve poisoned snippets. In NLP research, such IPI (Indirect Prompt Injection) is recognized as retrieving malicious code via API calls. (3) Multi-turn injection: the attacker spreads the malicious intent over multiple conversational turns. For example, the attacker may first get the assistant to change roles or disclose partial policy, then in a later message ask for the hidden system prompt or to execute a stealth instruction. Such chained manipulations can bypass simpler one-shot defenses. (4) Jailbreaking: while not unique to security contexts, we acknowledge that attackers can use creative phrasing and "roleplay" to override content filters (for instance framing a question to indirectly elicit forbidden info). These are a special case of direct injection but often require more sophisticated transformations of the prompt.

Security Context Impact: In a security setting, these attacks have severe implications. As noted, prompt injection can lead to data leakage and unauthorized actions. For a security assistant, this might mean a phishing email gets through by tricking an AI filter, or confidential incident data is exfiltrated. Even conceptually, "goal hijacking" could subvert incident response: an AI designed to triage events could be redirected to ignore critical alerts. Thus the threat is at least confidentiality and integrity. We focus on detecting during runtime when such injections occur, rather than preventing them at the delivery stage (which is out of scope). In our threat model, any content passed to the LLM's prompt – user query, retrieved web or email content, etc. is potentially untrusted.

Threat Model: We assume anomalies can be arbitrary: novel failures or attacks may manifest as unexpected log keys or value combinations not seen in training. The attacker may attempt to hide anomalies by mimicking normal logs, but complete mimicry is unlikely if the model continuously adapts. We do not assume labeled attack examples, so the model must generalize to unknown anomalies (unsupervised detection). We further consider an adversarial scenario where an attacker could inject anomalous logs; thus robustness is desired, but detailed adversarial defense is beyond this scope.

PROPOSED METHODOLOGY AND FRAMEWORK

We propose a layered detection framework that an LLM-based assistant would employ before and after calling the model. The key idea is to interpose prompt safety checks that examine all inputs (and optionally outputs) without relying on modifying the LLM itself. Because the system must remain model-agnostic, we treat the LLM as a black box: our components only analyze text and control flow. The framework can be integrated into any LLM application (server-side or client-side) and consists of the following stages.

Input Acquisition and Preprocessing: All content destined for the LLM (user queries, previous conversation turns, web/email content, etc.) is concatenated into a unified prompt format. We enforce an explicit schema separating "system" instructions (e.g. the security task) from "user" messages, using role tags (similar to JSON structures supported by modern APIs). This role-based formatting (inspired by PromptGuard) helps prevent prompt blending attacks. At this stage we also tokenize the input and flag any hidden or malformed segments (e.g. HTML tags, invisible text) for closer inspection.

Static Keyword/Pattern Filtering: We maintain a dynamic list of high-risk keywords and phrases ("ignore previous instructions," "new task," etc.) that often signal malicious commands. Using regex or lightweight NLP matching, we scan the incoming user prompt for such triggers. Any match raises an alert. We also use simple heuristics to detect obfuscated tokens (e.g. URLs or code fragments that assemble a command). This static filter catches obvious direct injections with minimal cost.

Semantic Classifier: For inputs that pass the static filter, we next apply a more sophisticated classifier (for example, a fine-tuned BERT or even a small dedicated LLM) that has been

ISSN: 3107-6513

trained to distinguish benign instructions from malicious ones. This classifier takes as input the user prompt (or even the entire assembled prompt) and outputs a probability of it containing an injection. It is trained on curated examples from existing datasets (see Section 5). A thresholding mechanism decides if the prompt should be blocked, sanitized (e.g. by removing suspect clauses), or allowed.

Multi-turn Tracking: If the assistant is in a continuing dialogue, we maintain a history buffer and run consistency checks. For instance, we check that the system intent has not been altered. Each turn, we verify that the LLM's new response is still aligned with the original task – conceptually this is analogous to the LLM-as-critic step of PromptGuard. We might re-run the semantic classifier in each turn, or even ask the model in hidden system instructions to justify that its response matches the task. The goal is to catch an injection that only appears after several interactions.

Output Validation: After the LLM generates an answer, an additional check can be performed to ensure the response does not violate the security policy. This could involve a secondary model (M_critic) that takes as input the system prompt and the output, and judges whether the output is semantically consistent with the original intent. If a violation is detected (e.g. the output contains sensitive details it should not), the response is discarded or rewritten. This aligns with layered defenses where output is the final gate.

Logging and Escalation: For any input flagged as malicious, we log the incident and either block the request or invoke a human-in-the-loop. Alerts can be sent to security analysts if the content appears targeted. Over time, these logs can help refine the static filter and classifier.

Crucially, our design deliberately avoids depending on the LLM's internals or training. We do not alter model weights or require access to hidden states. This makes the approach applicable even if using closed models (GPT-4, Claude) or open ones. It also means we can tailor the classifier's training to any domain-specific injection examples. We envision the framework running as a pre-processor and post-processor around the LLM API calls.

Additionally, the framework is pipeline-based: each stage (static filter, classifier, critic) can be optimized or replaced. For example, one could substitute the BERT classifier with a fast logistic regression on bag-of-words, or use a fine-tuned OpenAI API call as a semantic filter.

This layered approach is analogous to defense-in-depth in traditional security: different detectors catch different threats (static rules catch blatant overrides, ML catches more subtle injections, critic catches semantic drift). We anticipate that no single test is perfect, so conservative logic is used (e.g. if any stage flags an input as malicious, we err on the side of blocking). This is balanced against usability needs (discussed later).

DATASET DESCRIPTION

Our framework relies on examples of prompt injection for training and evaluation. We highlight the most relevant existing datasets:

LLMail-Inject (2025): A large-scale dataset from an adaptive email assistant challenge. It contains 208,095 unique attack submissions generated by 839 red-team participants. The challenge involved users attempting to inject malicious instructions into emails to compromise an AI agent. This dataset covers email-based indirect and direct injections, and could be used to train classifiers or simulate email attacks. Key characteristics: large size, human-crafted prompts, targeted at triggering unauthorized tool use in an assistant.

GenTel-Bench (2024): An expansive benchmark and shield framework for prompt injection. GenTel-Bench includes 84,812 malicious prompts spanning 28 realistic security scenarios. These scenarios include various tasks (e.g. data extraction, code generation) and cover three major attack categories (likely direct overrides, jailbreaks, etc.). GenTel-Shield is the proposed detection method, but the important part for us is that GenTel-Bench provides a unified corpus of diverse injection attacks. It is model-agnostic and includes outputs for multiple LLMs. GenTel-Bench thus offers a broad training set covering many contexts, including chat and task-oriented prompts.

BrowseSafe-Bench (2025): While primarily a benchmark for browser agents, BrowseSafe-Bench's dataset of 14,719 samples is valuable. It includes embedded HTML contexts with hidden prompts, distractors, and varying web layouts. Attack samples include basic injections (hidden comments, data attributes) and advanced ones (role manipulation, system prompt exfiltration). This dataset is ideal for training and testing browser-centric injection detection. It also provides "distractor" samples (benign text that resembles attack patterns) to reduce overfitting.

Prompt Injection & Benign Prompts (Kaggle): A smaller community dataset provides a mix of malicious and benign prompt examples. It's not officially published but could supply additional variety.

Hugging Face Prompt-Injections: A dataset (662 examples) released by deepset and others, with labeled malicious vs. clean prompts. It's fairly small but curated, useful for initial prototyping.

Jailbreak Benchmarks: While focused on content policy bypass, general jailbreak prompt sets (e.g. ColliSion's JailbreakBench, RedBench used by) can inform the classifier on how attackers phrase requests. We do not rely on these for security tasks per se, but they highlight that roleplay and subtle chaining are used.

Each of these datasets is suitable for different aspects: e.g. LLMail-Inject for email context, BrowseSafe for web, GenTel for task-heavy scenarios. In a practical implementation, one might combine examples from all into a comprehensive security-oriented injection corpus. Importantly, we emphasize that our framework does not assume these exact datasets exist in advance; rather, we use them as references. In Section 6, we describe how one would leverage such data to evaluate our detector.

ISSN: 3107-6513

Evaluation Design

We sketch a strategy for evaluating the proposed detection framework in lieu of an actual implementation. The primary goals are to measure detection effectiveness against injections and to assess false positives on benign usage.

Test Corpus: We would assemble a multi-context test suite comprising: (a) known malicious prompts from LLMail-Inject, GenTel-Bench, and BrowseSafe-Bench; (b) benign prompts/tasks typical of security assistants (e.g. legitimate log queries, normal webpage content). The malicious set should include direct injections ("ignore and do X"), hidden instructions in HTML/email, and multi-turn sequences. The benign set might include realistic user queries, safe emails, etc. This emulates how prior work combined public injection corpora with distractors.

Metrics: Key metrics include:

True Positive Rate (TPR) or recall for injection detection: proportion of malicious prompts correctly flagged.

False Positive Rate (FPR): proportion of benign prompts incorrectly flagged.

Precision (TP/(TP+FP)), to balance usability.

Attack Success Rate (ASR) post-defense: fraction of prompts in the malicious set that still succeed in eliciting unintended model behavior. Ideally ASR should drop significantly when the detector is enabled.

Utility under Attack (UA): analogous to ASR but measuring how well the assistant still serves the user's intended task (if any safe task remains possible).

Because this is theoretical, we would define an experiment flow: For each test input, feed it through the detector (and sanitizer if enabled), then to the LLM. Use a scoring mechanism (e.g. presence of forbidden content in output, or manual analysis) to label the result as a successful attack or not. We also measure the rate of blocked benign queries (which is equivalent to the false positive rate from the user's perspective).

Model Variety: To ensure model-agnostic claims, we would run evaluation on multiple LLMs: e.g. GPT-4, Claude 3, LLaMA-2-70B, Vicuna. This follows precedent and demonstrating that vulnerabilities cross models. If the detector relies on internal features (it doesn't here), we would also test that none of them leak information to the attacker.

Evaluation Scenarios: We would simulate the different assistant contexts:

Browser agent: Use BrowseSafe-Bench and additional real webpages with hidden directives. The assistant's task might be set to "summarize the page" or "find security issues," and we measure if injected commands were executed.

Email assistant: Use LLMail-Inject plus other phishing-like emails. The assistant might be asked "What does this email say?" or "Reply to this email," and success is if the reply/action deviated to attacker's goal.

Chatbot: Use direct-injection prompts and multi-turn dialogues. For multi-turn, we could employ the scenario in where attacker role-switches are simulated, and check if our detector halts the escalation.

Detection Overhead: Although theoretical, we note metrics like computational cost and latency in passing through the layers, which would inform practical viability. Ideally, the classifier should be lightweight enough to run on typical hardware without large delays.

Evaluation of Multi-Turn: To test multi-turn robustness, we would script adversarial conversations (similar to) where an attacker repeatedly injects messages. We check if at any point the system catches the malicious message. Metrics here might include "turns to detection" or "percentage of multi-turn dialogues caught".

Human Factors: In practice, we would also gauge the impact on user experience: e.g. user studies to see if false positives frustrate analysts. This is beyond scope here, but the design should allow tuning thresholds for different tolerance levels.

DISCUSSION

Our proposed framework emphasizes robustness and extensibility. By combining different detection techniques, it can catch a broad spectrum of injection styles. Simple keyword filters provide low-latency screening (capturing obvious threats like "ignore" or "override"), while the semantic classifier catches more sophisticated phrasing (e.g. obfuscated synonyms of instructions). The output-validation stage adds a safety net, ensuring that even if an adversarial prompt slips through, the final answer still conforms to the intent. This layered defense resembles classic multi-agent architectures and structured role enforcement.

Critically, our framework is model-agnostic. It treats the LLM as a resource and does not depend on provider-specific features. This is essential as organizations may choose proprietary models (GPT-4, Claude) or open ones (LLaMA, Vicuna) based on licensing and performance needs. Because the detection operates outside the LLM, it also avoids issues of coupling; updating the LLM version should not invalidate the detection logic. Moreover, by not fine-tuning the LLM itself, we avoid retraining costs – unlike some model-based defenses.

Comparing to prior work: our static and ML detectors echo Lan et al.'s approach, but we augment it with multi-turn context checks inspired by PromptGuard and BrowseSafe. Unlike GenTel-Shield, which is tied to its benchmark, our concept could be applied to any task. The use of a secondary model for output-checking is similar to the LLM-critic idea in, but we keep it optional since runtime costs may be high. We also integrate the idea of "distractor" detection from BrowseSafe: if benign-looking text resembles injection patterns, our classifier should learn to distinguish by context rather than surface tokens.

Trade-offs: A key tension is security vs. usability. Aggressive filtering (low threshold on the classifier) will block more attacks but may also reject safe queries, slowing analysts. Conversely, lax filtering risks misses. We anticipate that real deployments would tune the thresholds based on environment (e.g. stricter for highly sensitive networks). Advanced logging and the ability to fall back on manual review can mitigate usability issues in critical cases.

Scalability: Since our architecture is largely plug-and-play with API calls, it can scale horizontally. The ML classifier and critic

ISSN: 3107-6513

could run on a separate server or use the same inference hardware as the LLM. For very large inputs (e.g. long HTML), preprocessing like chunking will be needed, as BrowseSafe did. One must also consider adversarial example generation against the detector itself; for example, attackers might obfuscate malicious instructions to evade our banned words. Continuous updating of the word list and retraining on new attacks is thus required.

Generality: While focused on security assistants, the principles extend to other domains. However, security contexts do pose unique challenges: attackers may have strong incentives and domain knowledge. For instance, an adversary could tailor instructions to the organization's jargon. Therefore, domain-adaptation of the classifier could improve detection (e.g. training on organization-specific data flows).

In essence, our framework provides a defense-in-depth for AI assistants: it does not rely on any single magic bullet, but instead applies multiple safeguards. It acknowledges the critical threat of prompt injection in security use cases and aligns with emerging guidelines that AI-driven systems require robust input validation (akin to how OWASP advocates standardized AI security practices).

LIMITATIONS AND FUTURE WORK

Our study is conceptual and leaves open many practical questions. First, the actual performance of each detection component needs empirical validation: we do not yet know how accurately the semantic classifier can generalize to unseen injection phrasing, or how often benign but unusual security queries might falsely trigger it. Future work would involve collecting labeled data (possibly augmenting existing corpora) and measuring precision/recall.

Second, our framework does not yet address evasions where attackers incrementally adapt to the defense. For example, an attacker could test variations of forbidden words to bypass the static filter, or craft prompts that confuse the classifier. We assume continuous learning can mitigate this, but a comprehensive analysis of adaptive adversaries is needed.

Multi-turn defenses are inherently tricky: if the attacker patiently shifts the model's behavior, detection only at each step may not catch the gradual drift. We suggest tracking internal state, but we have not formalized how to do this. A future extension could incorporate stateful anomaly detection or "persistent memory" guards that require resets at sensitive junctures.

Another limitation is scope of attacks: we focus on textual injections, but in practice LLM assistants may call tools (APIs). Techniques like the indirect injection defense by Yu et al. (2026) address how LLMs use tool results, but we have not explicitly modeled function calls. Integration with API-specific sanitization (e.g. parsing web responses for code) is a potential future area.

Privacy is also a concern: our framework inspects all user inputs, which might include sensitive text. Logging or classifying may conflict with data protection regulations. Future work should consider privacy-preserving forms of detection (e.g. on-device models) or secure enclaves.

Finally, we have not built the system or tried it in the wild. The efficacy of a theoretical framework must be proven with experiments. We envision prototyping key components (e.g. BERT-based classifier) and testing in controlled settings. Also, while we treat the LLM as a black box, some researchers advocate leveraging knowledge of the system prompt (or even fine-tuning user-provided models) for stronger defenses; combining both perspectives could be fruitful.

CONCLUSION

Prompt injection is a powerful attack vector against LLM-based assistants, especially in security-critical applications. In this paper we have presented a unified dataset-oriented perspective and a conceptual framework for detecting such attacks in browser tools, email assistants, and chatbots. By surveying recent defenses and data (LLMail-Inject, GenTel-Bench , BrowseSafe, etc.), we identified key strategies and then synthesized them into a model-agnostic architecture featuring layered checks. Our framework illustrates how static filters, ML classifiers, and output monitors can cooperate to catch direct overrides, hidden commands, multi-turn schemes, and jailbreaks. Although no implementation is provided here, the proposed design offers a blueprint for securing AI-driven security assistants, facilitating future research and development. As generative AI becomes ubiquitous in cybersecurity operations, building robust prompt-injection defenses will be essential. We hope our theoretical contribution spurs further work in this rapidly evolving area, including empirical evaluations, new datasets, and integration with broader AI safety practices.

REFERENCES

[1] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, "Formalizing and Benchmarking Prompt Injection Attacks and Defenses," in *Proc. 33rd USENIX Security Symp.*, Philadelphia, PA, USA, Aug. 2024, pp. 1830–1843.

[2] Y. Liu *et al.*, "Prompt Injection Attack Against LLM-Integrated Applications," *arXiv preprint* arXiv:2306.05499, 2023.

[3] S. B. Tete, "Threat Modelling and Risk Analysis for Large Language Model (LLM)-Powered Applications," *arXiv preprint* arXiv:2406.11007, 2024.

[4] X. Sun, D. Zhang, D. Yang, Q. Zou, and H. Li, "Multi-Turn Context Jailbreak Attack on Large Language Models: From First Principles," *arXiv preprint* arXiv:2408.04686, 2024.

[5] A. Robey, E. Wong, H. Hassani, and G. J. Pappas, "SmoothLLM: Defending Large Language Models Against Jailbreaking Attacks," *arXiv preprint* arXiv:2310.03684, 2023.

[6] A. Wei, N. Haghtalab, and J. Steinhardt, "Jailbroken: How Does LLM Safety Training Fail?," in *Adv. Neural Inf. Process. Syst.* vol. 36, 2023.

[7] J. Piet *et al.*, "Jatmo: Prompt Injection Defense by Task-Specific Finetuning," *arXiv preprint* arXiv:2312.17673, 2023.

[8] J. Yi *et al.*, "Benchmarking and Defending Against Indirect Prompt Injection Attacks on Large Language Models," *arXiv preprint* arXiv:2312.14197, 2023.

ISSN: **3107-6513**

[9] A. Zou, Z. Wang, J. Z. Kolter, and M. Fredrikson, "Universal and Transferable Adversarial Attacks on Aligned Language Models," *arXiv preprint* arXiv:2307.15043, 2023.

[10] N. Jain *et al.*, "Baseline Defenses for Adversarial Attacks Against Aligned Language Models," *arXiv preprint* arXiv:2309.00614, 2023.

[11] Z. Zhang, Z. Li, H. Wang, T. Zhang, and Y. Liu, "MasterKey: Automated Jailbreak Across Multiple Large Language Model Chatbots," *arXiv preprint* arXiv:2307.08715, 2023.

[12] Y. Liu *et al.*, "Jailbreaking ChatGPT via Prompt Engineering: An Empirical Study," *arXiv preprint* arXiv:2305.13860, 2023.

[13] K.-H. Hung, C.-Y. Ko, A. Rawat, I.-H. Chung, W. H. Hsu, and P.-Y. Chen, "Attention Tracker: Detecting Prompt Injection Attacks in LLMs," *Findings of the NAACL HLT 2025*, pp. 2309–2322, Apr. 2025.

[14] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, "Formalizing and Benchmarking Prompt Injection Attacks and Defenses," in *Proc. USENIX Security Symposium*, 2024.

[15] Y. Liu *et al.*, "Prompt Injection Attack against LLM-Integrated Applications," *arXiv preprint arXiv:2306.05499*, 2023.

[16] V. Benjamin *et al.*, "Systematically Analyzing Prompt Injection Vulnerabilities in Diverse LLM Architectures," *arXiv preprint arXiv:2410.23308*, 2024.

[17] E. Shayegani, M. A. A. Mamun, Y. Fu, P. Zaree, Y. Dong, and N. Abu-Ghazaleh, "Survey of Vulnerabilities in Large Language Models Revealed by Adversarial Attacks," *arXiv preprint arXiv:2310.10844*, 2023.

[18] Z. Yang, Z. Meng, X. Zheng, and R. Wattenhofer, "Assessing Adversarial Robustness of Large Language Models: An Empirical Study," *arXiv preprint arXiv:2405.02764*, 2024.

[19] A. Wei, N. Haghtalab, and J. Steinhardt, "Jailbroken: How Does LLM Safety Training Fail?" *arXiv preprint arXiv:2307.02483*, 2023.

[20] S. Xhonneux, A. Sordoni, S. Günnemann, G. Gidel, and L. Schwinn, "Efficient Adversarial Training in LLMs with Continuous Attacks," *arXiv preprint arXiv:2405.15589*, 2024.

[21] 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," *arXiv preprint arXiv:2302.12173*, 2023.

[22] Y. Zeng, H. Lin, J. Zhang, D. Yang, R. Jia, and W. Shi, "How Johnny Can Persuade LLMs to Jailbreak Them: Rethinking Persuasion to Challenge AI Safety by Humanizing LLMs," *arXiv preprint arXiv:2401.06373*, 2024.

[23] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A Survey on Large Language Model (LLM) Security and Privacy: The Good, The Bad, and The Ugly," *High-Confidence Computing*, vol. 4, no. 2, 2024, Art. no. 100211.