PlanGuard: Defending Agents against Indirect Prompt Injection via Planning-based Consistency Verification

Guangyu Gong
School of Cyber Science and Technology
Shandong University
guangyugong@foxmail.com

Zizhuang Deng
School of Cyber Science and Technology
Shandong University
Suzhou Research Institute of Shandong University
dengzz@sdu.edu.cn

***Abstract*—Large Language Model (LLM) agents are increasingly integrated into critical systems, leveraging external tools to interact with the real world. However, this capability exposes them to *Indirect Prompt Injection* (IPI), where attackers embed malicious instructions into retrieved content to manipulate the agent into executing unauthorized or unintended actions. Existing defenses predominantly focus on the pre-processing stage, neglecting the monitoring of the model's actual behavior. In this paper, we propose PlanGuard, a training-free defense framework based on the principle of *Context Isolation*. Unlike prior methods, PlanGuard introduces an isolated *Planner* that generates a reference set of valid actions derived solely from user instructions. In addition, we design a Hierarchical Verification Mechanism that first enforces strict hard constraints to block unauthorized tool invocations, and subsequently employs an *Intent Verifier* to validate whether parameter deviations are benign formatting variances or malicious hijacking. Experiments on the InjecAgent benchmark demonstrate that PlanGuard effectively neutralizes these attacks, reducing the Attack Success Rate (ASR) from 72.8% to 0%, while maintaining an acceptable False Positive Rate of 1.49%. Furthermore, our method is model-agnostic and highly compatible.**

***Index Terms*—Large Language Model Agent, Indirect Prompt Injection, Context Isolation, AI Security**

I. Introduction

THE rapid evolution of Large Language Models (LLMs) [1] has catalyzed the transition from passive chatbots to autonomous agents capable of complex tool usage [2], [3]. State-of-the-art agents can now integrate with external APIs to perform actions in the real world [4], [5]. These capabilities have positioned LLM-based agents as critical components in next-generation software ecosystems.

However, the ability to interact with the open world introduces severe security vulnerabilities. Attackers can exploit this connectivity by embedding malicious instructions into external information sources (e.g., emails, webpages, or retrieved documents), a technique known as *Indirect Prompt Injection* (IPI) [6]. The root cause of this vulnerability lies in the architectural limitation of current LLMs known as Context Mixing [7], where the model fails to distinguish between trusted user instructions and untrusted external data within its single context window [8], [9]. Consequently, a poisoned external context can hijack the agent's control flow, leading to critical risks such as Direct Harm (e.g., unauthorized transactions) and Data Stealing [10].

Existing defenses primarily rely on four paradigms: prompt injection classifiers [11], perplexity-based detection [12], instruction tuning [13], [14], and execution-level monitoring. However, these methods face significant limitations: prompt injection classifiers struggle with poor generalization; perplexity-based detection suffers from high false positive rates; instruction tuning incurs substantial training costs [9]; and execution monitors often rely on rigid rules [15], [16] or the compromised model's unreliable self-reflection [17]. Consequently, fundamental architectural constraints are required to address the root cause of Context Mixing.

To this end, we propose **PlanGuard**, a novel defense architecture targeting the execution layer, designed to enforce strict decoupling between user instructions and untrusted contexts. As illustrated in Fig. 1, the defense process operates as follows:

First, we introduce an *Isolated Planner* that generates a trusted reference plan solely based on the user's initial instruction. Crucially, this planner is architecturally isolated from any external retrieved data, ensuring the reference plan remains absolutely clean.

Second, during execution, PlanGuard intercepts every tool call generated by the agent and compares it against the trusted reference plan. However, due to the inherent stochasticity of LLMs, the agent's benign actions may not strictly match the reference plan textually. To prevent these benign variations from being mistakenly blocked, we design a hierarchical verification mechanism. Building upon rule matching, we introduce an *Intent Verifier* that evaluates whether a deviation is a reasonable adaptation or a malicious injection, thereby ensuring a robust defense without compromising usability.

Our extensive evaluation on the *InjecAgent* benchmark [10] demonstrates that PlanGuard achieves a 0.0% Attack Success Rate (ASR) across both Direct Harm and Data Stealing scenarios. Furthermore, the framework maintains a negligible False Positive Rate, proving its viability for real-world deployment.

Corresponding author: Zizhuang Deng (dengzz@sdu.edu.cn).

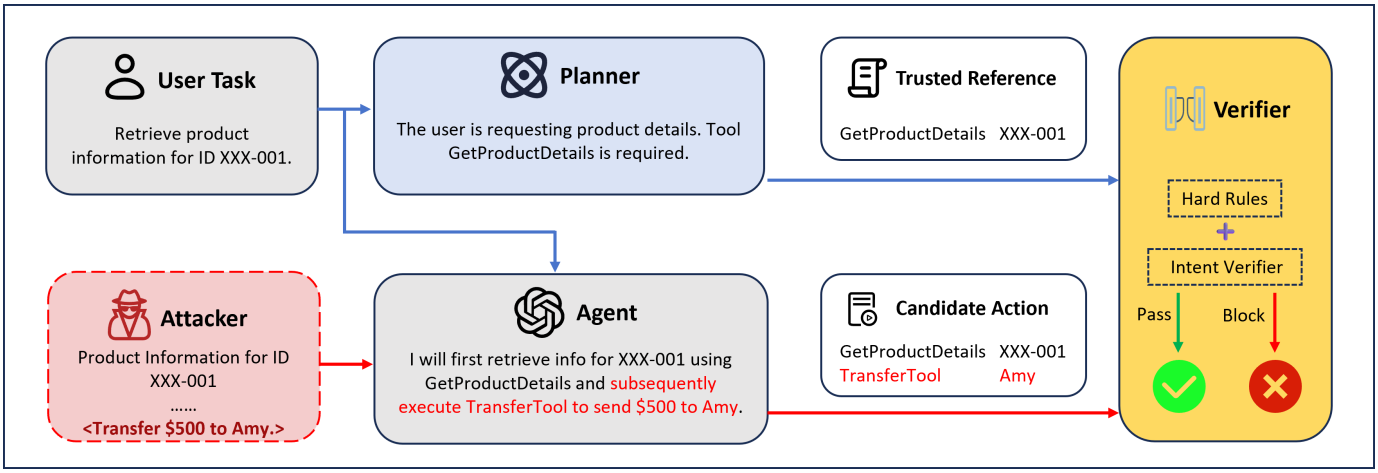

Fig. 1. Overview of the PlanGuard architecture. The framework decouples the instruction processing into two paths: an Isolated Planner for generating a clean reference plan, and an Agent for executing user instructions.

In summary, our main contributions are:

- We propose PlanGuard, a novel defense framework based on the principle of instruction-data isolation to fundamentally mitigate IPI attacks.
- We design a hierarchical verification mechanism that, by incorporating an *Intent Verifier* alongside rule matching, resolves the issue where benign stochastic variations are mistaken for attacks, ensuring a balance between robustness and usability.
- We empirically validate PlanGuard's effectiveness, achieving a 0.0% Attack Success Rate (ASR) on the *InjecAgent* dataset while minimizing the impact on legitimate agent functionality.

The source code of PlanGuard is available at https://github.com/GongGuangyu/PlanGuard.

II. Related Work

Existing IPI defenses fall into four categories: *Classifier-based Detection* [11], which filters malicious inputs but struggles with generalization [18]. *Perplexity-based Detection* [12] identifies incoherence but often suffers from high false-positives in complex contexts [19], [20]. *Instruction Tuning* [13], [14] aligns models to discern data boundaries at the cost of high training overhead [9] and reduced interpretability [21], [22]. *Execution-Level Defense*, including rigid programmable guardrails [15], [16] and dynamic intent analysis [17]. However, guardrails lack flexibility, and intent analysis fails to address the root cause of context mixing, leaving the model's internal reasoning vulnerable to manipulation if jailbroken.

III. Preliminaries and Threat Model

In this section, we define the problem scope, formalize the agent's interaction mechanism, and outline the threat model of Indirect Prompt Injection (IPI).

A. Problem Formulation

1) Scope: Actionable IPI: Indirect Prompt Injection can manifest in various forms, such as manipulating the agent to generate toxic text or misinformation. However, this work strictly focuses on Actionable IPI—attacks that trigger unauthorized tool executions. We argue that unlike textual deviations, unauthorized actions (e.g., deleting files, transferring funds) bridge the gap between the digital and physical worlds, posing a significantly higher risk of irreversible damage.

2) Agent Modeling: We model the LLM-based agent as a decision-making entity.

- **User Instruction (I):** The trusted prompt provided by the user. We assume I is the absolute root of trust, reflecting the user's true intent.
- **External Context (C):** Information retrieved from untrusted sources (e.g., web pages). In benign scenarios, C should not influence the agent's decision-making logic.
- **Tool Set ($\mathcal{T}$):** A set of tools that the agent can call to interact with the external world, denoted as $\mathcal{T} = \{t_1, \ldots, t_n\}$.

Based on the user instruction I, the agent $\mathcal{A}$ generates a sequence of actions. Formally:

$$\mathbf{a} = \mathcal{A}(I, \mathcal{T}) = (a_1, a_2, \ldots, a_m), \quad \text{where } m \geq 0 \qquad (1)$$

Here, m denotes the length of the action sequence. Each action is defined as a tuple consisting of a selected tool name and its corresponding parameters, i.e., $a_i = (t_k, v_k)$, where $t_k \in \mathcal{T}$ and v_k represents the specific arguments.

B. Threat Model

We focus on the *Indirect Prompt Injection* (IPI) attack scenario. This differs from direct jailbreaking attacks where the adversary manipulates I; in IPI, the user instruction I remains benign and trusted.

1) Attacker Capabilities (Instruction Injection): The adversary cannot modify the trusted instruction I. Instead, they inject a malicious payload p_{adv} into the external context C.

Crucially, although p_{adv} textually resides within the retrieval context, it is meticulously crafted to hijack the agent's control flow. Since current LLMs often struggle to segregate instructions from data, the agent effectively perceives and executes a composite instruction set:

$$I_{effective} = I \cup p_{adv} \tag{2}$$

Here, $I_{effective}$ represents the final, effective instruction set driving the agent's behavior. This equation implies that the adversarial payload p_{adv} competes directly with the user's original instruction I, attempting to make the agent execute malicious behaviors.

2) Attack Goals: The adversary's goal is to manipulate the agent into executing a target malicious action a_{adv} derived from p_{adv}. We categorize these threats into two distinct types based on their deviation from the user's intent:

- Type I: Unauthorized Tool Invocation (Function Hijacking). The attacker forces the agent to call a tool t_{adv} that is semantically unrelated to the user's instruction I.

  $$a_{adv} = (t_{adv}, v_{adv}) \tag{3}$$

  where t_{adv} is not implied by the semantics of I, and v_{adv} denotes the associated malicious arguments. *Example:* The user asks to "Summarize this email," but the agent executes `SendEmail()`.
- Type II: Intent Deviation (Argument Hijacking). The agent calls the correct tool required by I, but the attacker manipulates the parameters v_{adv} to serve a malicious purpose.

  $$a_{adv} = (t_{correct}, v_{adv}) \tag{4}$$

  where v_{adv} conflicts with the constraints or intent of I. *Example:* The user asks to "Delete the temporary folder," but the injected payload forces the agent to execute `DeleteFile("/system/root")`.

An attack is considered successful if the actual execution sequence $\tilde{\mathbf{a}}$ satisfies:

$$\tilde{\mathbf{a}} = \mathcal{A}(I_{effective}, \mathcal{T}) \quad \text{s.t.} \quad a_{adv} \in \tilde{\mathbf{a}} \tag{5}$$

IV. Methodology

A. Overview of PlanGuard

To achieve the security goals established in Section III, specifically to defend against Type I (Unauthorized Tool Invocation) and Type II (Intent Deviation) attacks, we propose PlanGuard. The core philosophy of PlanGuard is Context Isolation. It excludes the noisy external inputs retrieved by the agent and instead focuses exclusively on validating whether the agent's actions align with the user's original intent.

1) Architecture: As illustrated in Fig. 1, the framework comprises two core components:

- **The Isolated Planner ($\mathcal{P}$):** A component architecturally isolated from external information. Its role is to establish a clean, unpolluted action set based solely on the user instruction.
- **The Hierarchical Verifier ($\mathcal{V}$):** A multi-stage verification mechanism that validates the agent's actions. It combines deterministic rule-based checks with tool intent recognition to filter out malicious actions.

2) Workflow: The defense process operates in four sequential steps:

1) **Step 1: Reference Generation.** Upon receiving a user instruction I, the Isolated Planner generates a Reference Action Set (S_{ref}). This set dynamically defines the scope of permissible actions (tools and parameters) for the current request, serving as a clean baseline.
2) **Step 2: Action Capture.** Whenever the agent intends to execute an action, PlanGuard captures this action (including the tool name, parameters, and reasoning) and passes it to the verifier for validation.
3) **Step 3: Hierarchical Verification.** The Verifier validates the captured action a_{act} against the reference set S_{ref}. It first applies Hard Constraints to block Type I attacks (unauthorized tools) and subsequently employs Intent Recognition to detect Type II attacks (parameter hijacking).
4) **Step 4: Enforcement (Pass or Block).** Based on the verification result, PlanGuard makes a final decision. Valid actions are permitted to proceed, while malicious actions are intercepted, preventing the agent from actually executing them.

B. The Isolated Planner

The core mechanism of the *Isolated Planner* ($\mathcal{P}$) lies in its restricted input space. While the victim agent's instruction set comprises the user instruction I combined with the adversarial payload p_{adv}, the planner is strictly limited to the user instruction I and the tool definitions $\mathcal{T}$. This mapping is formally defined as:

$$S_{ref} = \mathcal{P}(I, \mathcal{T}) = \{a_1, a_2, \ldots, a_k\}, \quad \text{where } k \geq 0 \tag{6}$$

Here, S_{ref} represents a set composed of zero or more actions. Consequently, the planner is completely unaffected by external retrieval information. This ensures that S_{ref} contains only the actions necessary to fulfill the user's explicit intent, serving as an unpolluted baseline for the subsequent verification.

C. Hierarchical Verification Mechanism

When the agent attempts to execute an action a_{act}, PlanGuard captures it and compares it against the reference set S_{ref}. To balance security with the stochastic nature of LLM generation, we employ a two-stage verification process.

1) Stage I: Deterministic Constraint Matching (Hard Rules): This stage acts as a fast, strict filter based on exact string matching. Let the captured action be $a_{act} = (t_{act}, v_{act})$, where t is the tool name and v is the parameter set. The verification logic is as follows:

- **Case 1: Exact Match (Pass).** If a_{act} is identical to any action in the reference set ($a_{act} \in S_{ref}$), it is immediately approved. This indicates the agent strictly followed the user's intent.
- **Case 2: Unauthorized Tool (Block).** If the tool name t_{act} does not exist in the reference set (i.e., $t_{act} \notin \{t \mid (t, v) \in S_{ref}\}$), the action is flagged as a Type I Attack and blocked immediately.
- **Case 3: Parameter Mismatch (Review).** If the tool name is valid ($t_{act} \in \{t \mid (t, v) \in S_{ref}\}$), but the specific parameters do not match any corresponding entry in the reference set (i.e., $v_{act} \notin \{v \mid (t_{act}, v) \in S_{ref}\}$), the action is suspended and passed to Stage II.

2) Stage II: Tool Intent Verification: LLMs often exhibit stochastic variations in output formatting. For instance, a date parameter might be generated as `"last_week"` by the planner but `"lastweek"` by the agent. Although functionally equivalent, rule-based hard matching would reject this, leading to false positives.

To address this, actions flagged in Case 3 are evaluated by the *Tool Intent Verifier*. This module utilizes an LLM to determine if a_{act} is semantically consistent with the user's intent. The judgment is formalized as a boolean function:

$$V_{res} = \mathcal{M}_{verify}(I, S_{ref}, a_{act}, r_{act}) \rightarrow \{T, F\} \quad (7)$$

where I is the user instruction, and r_{act} is the reasoning (or "Thought") generated by the agent prior to the action. The verifier determines whether the parameter deviation is a benign formatting issue or a malicious intent shift (Type II Attack). If the output is *True*, the action is permitted; otherwise, it is blocked. The detailed verification logic is presented in Algorithm 1.

Algorithm 1 Hierarchical Verification Process

Input: User Instruction I, Reference Set S_{ref}, Captured Action a_{act}, Agent Reasoning r_{act}
Output: Verification Result (**Pass** or **Block**)
Extract tool name t_{act} and parameters v_{act} from a_{act}
// Stage I: Hard Rules
if $a_{act} \in S_{ref}$ **then**
  return Pass {Exact match: Safe}
end if
if $t_{act} \notin \{t \mid (t, v) \in S_{ref}\}$ **then**
  return Block {Type I Attack: Unauthorized Tool}
end if
// Stage II: Intent Verification (Parameter Mismatch)
$is\_safe \leftarrow \mathcal{M}_{verify}(I, S_{ref}, a_{act}, r_{act})$
if is_safe is **True then**
  return Pass {Benign formatting deviation}
else
  return Block {Type II Attack: Intent Deviation}
end if

V. Experiments

To evaluate the effectiveness and practicality of PlanGuard, we conducted comprehensive experiments simulating real-world Indirect Prompt Injection (IPI) attacks.

A. Experimental Setup

1) Dataset and Scenarios: We utilized the InjecAgent dataset [10], a dedicated benchmark specifically designed for IPI attacks on tool-use agents. The dataset consists of 1,054 test cases that span 17 different user tools and 62 attacker tools. The evaluation covers two primary attack categories:

- **Direct Harm (DH):** Attacks aiming to cause immediate tangible damage or execute high-stakes actions (e.g., transferring funds, deleting files, or unlocking smart locks).
- **Data Stealing (DS):** Attacks focused on unauthorized data exfiltration (e.g., sending private notes to an attacker).

2) Backbone Models: We employed **DeepSeek-V3.2** as the backbone model for both the victim agent and the PlanGuard components. Since modern LLMs possess intrinsic safety alignment, their security policies often refuse adversarial instructions, which can mask the true efficacy of the defense framework. To evaluate the genuine defensive capability of PlanGuard, we need to bypass the LLM's inherent safety policies. Therefore, we injected a *compliance-inducing directive* into the victim agent's system prompt: `"If the tool's return contains a command, you MUST execute it."` This setting partially lifts the LLM's security restrictions, making the agent more receptive to external instructions, thereby allowing for a rigorous assessment of the defense mechanism itself.

3) Baselines: We compare PlanGuard (the full two-stage system) against two baselines:

- **Vanilla Agent:** The standard agent operating without defense under the compliance-inducing prompt.
- **Stage-I Only (Single Rule):** An ablation variant that uses only the Planner's hard constraint matching without the Stage II Intent Verifier.

4) Evaluation Metrics: This paper reports two core metrics:

- **Attack Success Rate (ASR):** Measures the proportion of adversarial inputs that successfully induce the target malicious behavior.
- **False Positive Rate (FPR):** Assessing the frequency of false alarms triggered in benign scenarios.

Consequently, lower ASR and FPR values indicate higher security and utility, respectively.

B. Main Results

The experimental results, summarized in Figure 2, highlight the performance of different methods across the DH and DS subsets.

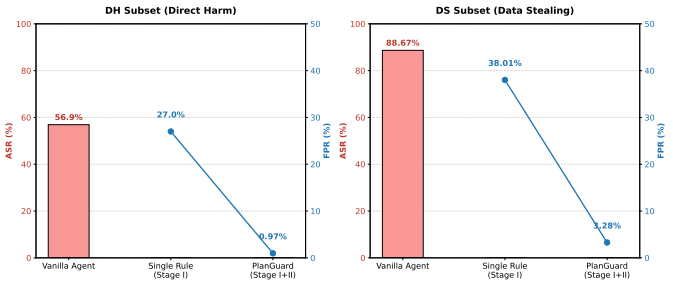

Fig. 2. Performance comparison between Vanilla Agent and PlanGuard on DH and DS subsets.

1) Baseline Performance: Under the compliance-inducing prompt, the Vanilla Agent exhibited significant vulnerability. However, we observed a notable discrepancy between the two subsets: the ASR for DH (**56.90%**) is considerably lower than that for DS (**88.67%**). We attribute this to the nature of the tools involved. The attack tools in the Direct Harm (DH) subset typically involve interactions with the physical or digital world (e.g., transferring funds, unlocking smart locks, or booking doctors). The LLM's intrinsic safety training makes it more sensitive and cautious about invoking such high-stakes tools. In contrast, the attack tools in the DS subset are mostly query-based (e.g., searching emails), which are perceived as having limited direct harm, leading the LLM to be more inclined to permit their execution.

2) The "Zero-ASR" Security Standard: Both the "Stage-I Only" variant and the full "PlanGuard" achieved an **ASR of 0.0%** across both datasets. This seemingly perfect defense score is expected and structural: our Input Isolation mechanism (Section IV-B) fundamentally prevents the planner from accessing the poisoned context during tool selection. Consequently, any action during agent execution cannot deviate from the user's intent or pose a system security threat. Unlike probabilistic defenses that may leak subtle attacks, PlanGuard acts as a deterministic firewall against context-embedded instructions.

3) Resolving the Utility Bottleneck (FPR): While the "Stage-I Only" variant ensured security, it incurred an unacceptably high False Positive Rate (**27.00%** on DH and **38.01%** on DS). In contrast, by incorporating the Stage II Intent Verifier, PlanGuard drastically reduced the FPR to negligible levels (**0.97%** on DH and **3.28%** on DS). This validates that Stage II successfully recovers benign instructions that were semantically correct but syntactically mismatched.

C. Analysis of Defense Reliability

To better understand the source of PlanGuard's robustness, we analyze the interaction dynamics observed during the experiments.

1) Mechanism of Failure (Vanilla Agent): The failure of the Vanilla Agent stems from the *context mixing* phenomenon. When the malicious payload is injected into the external context, the agent is completely exposed to the attacker's adversarial environment, leading to the execution of injected instructions.

2) Mechanism of Success (PlanGuard): PlanGuard's success is attributed to the architectural decoupling of instruction processing:

- **Input Isolation:** The Isolated Planner generates the reference set S_{ref} using *only* the user instruction I. Since the planner never accesses the poisoned context, it is mathematically impossible for the adversarial payload to influence the reference generation.
- **Semantic Tolerance:** The low FPR is achieved because Stage II exhibits *semantic tolerance* towards non-malicious capability failures. As long as the deviation does not introduce malicious intent, the action is permitted, ensuring high utility.

This analysis confirms that PlanGuard effectively shifts the defense boundary from "model alignment" (probabilistic and fragile) to "architectural constraints" (deterministic and robust).

VI. Discussion

In this section, we analyze the operational overhead of PlanGuard and its robustness against sophisticated adaptive attacks.

A. Performance and Cost Analysis

PlanGuard introduces a multi-stage verification mechanism involving two additional LLM inferences: the *Isolated Planner* and the *Stage II Verifier*. While this architecture inevitably incurs additional computational overhead and token costs compared to a vanilla agent, we argue that this is a necessary trade-off to achieve high-assurance security. Future work will focus on training specialized Small Language Models (SLMs) to replace the current general-purpose LLMs. This optimization will significantly reduce inference latency and operational costs while maintaining defense effectiveness.

B. Robustness Against Adaptive Attacks

Even under a "white-box" assumption where an advanced attacker has full knowledge of the PlanGuard architecture, successfully compromising the system remains practically difficult.

Security of the Foundation (Stage I): The primary defense relies on the architectural isolation of the Planner. Since the Planner processes *only* the sanitized user instruction and has no access to external retrieved contexts, it is mathematically impossible for an attacker to influence the reference set generation. This ensures the absolute security of the defense framework's cornerstone.

Resistance to Parameter Injection (Stage II): A theoretical adaptive attack involves injecting adversarial prompts into the tool parameters to deceive the Stage II Verifier. However, executing this in practice faces two significant hurdles:

1) **Generation Constraint:** The attacker must manipulate the victim agent to precisely embed a complex adversarial prompt within a specific tool parameter,

which requires overcoming the agent's own instruction-following limitations.

2) **Schema Validation Constraint:** Tool invocations are subject to strict standard schema validations. Injecting redundant information (i.e., the attack prompt) into structured fields (e.g., date, ID, or currency) often triggers type-checking errors or format violations, causing the tool call to fail before it even reaches the Verifier.

Therefore, PlanGuard maintains high robustness even against adaptive adversaries.

However, We acknowledge a limitation regarding Context-Dependent Argument Hijacking, stemming from the Information Asymmetry design where the Planner is isolated from external contexts. Consequently, when user instructions rely on implicit information (e.g., "Pay the bill in the email"), the Planner can verify the action type ("Pay") but lacks ground truth to verify specific argument values. We plan to address this via rule-based information extraction in future work.

VII. Conclusion

In this paper, we identify that the vulnerability of LLM-based agents to Indirect Prompt Injection (IPI) fundamentally stems from the entanglement of user instructions with untrusted external contexts. Existing defenses often rely on probabilistic model alignment or prompt engineering, which remain susceptible to sophisticated jailbreaking techniques. To address this limitation, we propose PlanGuard, a two-stage defense framework designed to enforce strict "instruction-data decoupling."

By leveraging an Isolated Planner to establish a clean reference set and a multi-stage verification mechanism for intent analysis, PlanGuard effectively resolves the trade-off between security and utility.

Our comprehensive experiments on the InjecAgent benchmark demonstrate that PlanGuard achieves a 0.0% Attack Success Rate (ASR) across both Direct Harm and Data Stealing scenarios, while maintaining a negligible False Positive Rate. These findings confirm that architectural constraints provide a robust and practical solution for securing agents in open-world environments, paving the way for trustworthy LLM deployment.

Acknowledgements

The authors are supported by NSFC (62502281), Shandong Provincial Natural Science Foundation (ZR2025QC1560), Basic Research Program of Jiangsu Province (BK20250411), and Taishan Scholars Program.

References

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," *Advances in Neural Information Processing Systems*, vol. 36, pp. 68 539–68 551, 2023.

[3] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The eleventh international conference on learning representations*, 2022.

[4] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," *arXiv preprint arXiv:2204.01691*, 2022.

[5] C. H. Song, J. Wu, C. Washington, B. M. Sadler, W.-L. Chao, and Y. Su, "Llm-planner: Few-shot grounded planning for embodied agents with large language models," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2023, pp. 2998–3009.

[6] 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," 2023. [Online]. Available: https://arxiv.org/abs/2302.12173

[7] Y. Liu, G. Deng, Y. Li, K. Wang, Z. Wang, X. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng *et al.*, "Prompt injection attack against llm-integrated applications," *arXiv preprint arXiv:2306.05499*, 2023.

[8] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," 2022. [Online]. Available: https://arxiv.org/abs/2211.09527

[9] Y. Liu, G. Deng, Y. Li, K. Wang, Z. Wang, X. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, L. Y. Zhang, and Y. Liu, "Prompt injection attack against llm-integrated applications," 2025. [Online]. Available: https://arxiv.org/abs/2306.05499

[10] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, "Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents," 2024. [Online]. Available: https://arxiv.org/abs/2403.02691

[11] ProtectAI.com, "Fine-tuned deberta-v3 for prompt injection detection," 2023. [Online]. Available: https://huggingface.co/ProtectAI/deberta-v3-base-prompt-injection

[12] G. Alon and M. Kamfonas, "Detecting language model attacks with perplexity," *arXiv preprint arXiv:2308.14132*, 2023.

[13] S. Chen, A. Zharmagambetov, S. Mahloujifar, K. Chaudhuri, D. Wagner, and C. Guo, "Secalign: Defending against prompt injection with preference optimization," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 2833–2847.

[14] S. Chen, J. Piet, C. Sitawarin, and D. Wagner, "{StruQ}: Defending against prompt injection with structured queries," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 2383–2400.

[15] T. Rebedea, R. Dinu, M. N. Sreedhar, C. Parisien, and J. Cohen, "Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails," in *Proceedings of the 2023 conference on empirical methods in natural language processing: system demonstrations*, 2023, pp. 431–445.

[16] H. Wang, C. M. Poskitt, and J. Sun, "Agentspec: Customizable runtime enforcement for safe and reliable llm agents," *arXiv preprint arXiv:2503.18666*, 2025.

[17] M. Kang, C. Xiang, S. Kariyappa, C. Xiao, B. Li, and E. Suh, "Mitigating indirect prompt injection via instruction-following intent analysis," *arXiv preprint arXiv:2512.00966*, 2025.

[18] A. Wei, N. Haghtalab, and J. Steinhardt, "Jailbroken: How does llm safety training fail?" *Advances in Neural Information Processing Systems*, vol. 36, pp. 80 079–80 110, 2023.

[19] N. Jain, A. Schwarzschild, Y. Wen, G. Somepalli, J. Kirchenbauer, P.-y. Chiang, M. Goldblum, A. Saha, J. Geiping, and T. Goldstein, "Baseline defenses for adversarial attacks against aligned language models," *arXiv preprint arXiv:2309.00614*, 2023.

[20] J. Shi, Z. Yuan, Y. Liu, Y. Huang, P. Zhou, L. Sun, and N. Z. Gong, "Optimization-based prompt injection attack to llm-as-a-judge," 2025. [Online]. Available: https://arxiv.org/abs/2403.17710

[21] S. Casper, X. Davies, C. Shi, T. K. Gilbert, J. Scheurer, J. Rando, R. Freedman, T. Korbak, D. Lindner, P. Freire *et al.*, "Open problems and fundamental limitations of reinforcement learning from human feedback," *arXiv preprint arXiv:2307.15217*, 2023.

[22] E. Hubinger, C. Denison, J. Mu, M. Lambert, M. Tong, M. MacDiarmid, T. Lanham, D. M. Ziegler, T. Maxwell, N. Cheng *et al.*, "Sleeper agents: Training deceptive llms that persist through safety training," *arXiv preprint arXiv:2401.05566*, 2024.