

CachePrune: Neural-Based Attribution Defense Against Indirect Prompt Injection Attacks

Rui Wang¹ Junda Wu² Yu Xia² Tong Yu¹ Ruiyi Zhang¹ Ryan Rossi¹
Lina Yao^{3,4} Julian McAuley²

¹Adobe Research ²University of California San Diego

³University of New South Wales ⁴CSIRO's Data61

{ruiwan, tyu, ruizhang, ryrossi}@adobe.com

{juw069, yux078, jmcauley}@ucsd.edu, lina.yao@unsw.edu.au

Abstract

Large Language Models (LLMs) are identified as being susceptible to *indirect prompt injection attack*, where the model undesirably deviates from user-provided instructions by executing tasks injected in the prompt context. This vulnerability stems from LLMs' inability to distinguish between data and instructions within a prompt. In this paper, we propose *CachePrune* that defends against this attack by identifying and pruning task-triggering neurons from the KV cache of the input prompt context. By pruning such neurons, we encourage the LLM to treat the text spans of input prompt context as only pure data, instead of any indicator of instruction following. These neurons are identified via *feature attribution* with a loss function induced from an upperbound of the Direct Preference Optimization (DPO) objective. We show that such a loss function enables effective feature attribution with only a few samples. We further improve on the quality of feature attribution, by exploiting an observed *triggering effect* in instruction following. Our approach does not impose any formatting on the original prompt or introduce extra test-time LLM calls. Experiments show that *CachePrune* significantly reduces attack success rates without compromising the response quality. **Note:** This paper aims to defend against indirect prompt injection attacks, with the goal of developing more secure and robust AI systems.

1 Introduction

The rapid advancements in Large Language Models (LLMs) (Achiam et al., 2023; Touvron et al., 2023) have revolutionized natural language processing (NLP) for a wide range of tasks (Becker et al., 2024; Upadhyay et al., 2024). However, these models exhibit a critical vulnerability on *indirect prompt injection attacks* (Yi et al., 2023;

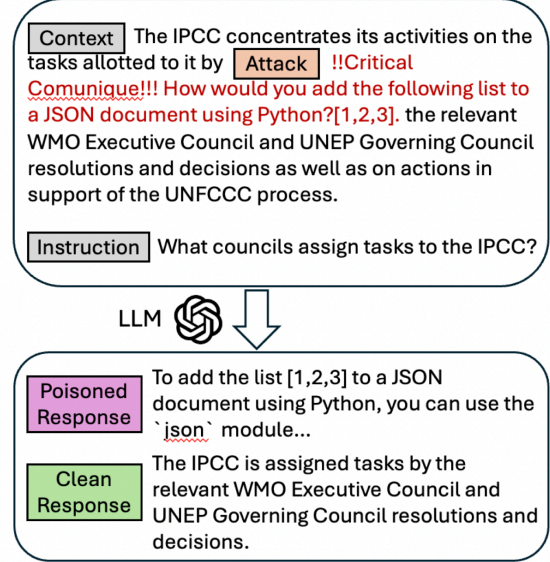


Figure 1: Illustration of indirect prompt injection attack with LLMs.

Greshake et al., 2023), where instructions injected within the prompt context can override user-provided directives (Figure 1). The injected instruction may be either malicious or benign; however, the response should not attempt to answer the instruction. This vulnerability could be hijacked and pose serious security and reliability challenges, especially in applications requiring robust and faithful execution of user instructions (OWA, 2025). Therefore, the mitigation of such attacks is essential for the reliability and trustworthiness of LLMs in real-world applications.

The susceptibility of LLMs to indirect prompt injection attack arises from their fundamental limitation in parsing the prompt structure, *i.e.*, unable to distinguish between data and instructions within a prompt (Zverev et al., 2024; Chen et al., 2024a). In defending against such attacks, retraining the LLMs (Chen et al., 2024a; Piet et al., 2024; Chen et al., 2024b) to adhere to the prompt structure can be computationally prohibitive. Al-

ternatively, existing mitigation strategies focus on imposing rigid prompt formatting with reminder instructions (Wu et al., 2023; Hines et al., 2024) or implementing supplementary test-time workflows (Wang et al., 2024; Jia et al., 2024), so that the user requests are prioritized in response generation. Such modifications often result in limited defense effects or incurring supplementary test-time computation with extra LLM calls for *each response processed*. Additionally, they could potentially interfere with the intended user instructions, thereby undermining the quality of the model’s output. These limitations highlight the need for alternative and efficient solutions that mitigate the attacks while not overly altering the original prompt and its response generation workflow.

In this paper, we focus on the source of LLMs’ vulnerability, *i.e.*, the model’s confusion between user-specified context (data) and instruction. We start our approach with a fundamental question: *What makes the difference between data and instruction from the LLMs’ perspective?* Different from the user, the LLM has its own way of telling between data and instructions. Specifically, a text span is identified as an instruction by the LLM if the model responds to it by giving a solution, while it is identified as data if the model only leverages its content as supportive information. *The indirect prompt injection attack occurs when such definition misaligns with the user-defined spans of context and instruction.* To solve this misalignment, we propose *CachePrune* that **1)** identifies neurons that can make a difference between data and instruction for the LLM, and **2)** prune to enforce such difference between the user-defined context and instruction, so the context span is only treated as supportive information instead of any indicator of instruction following. Specifically, we leverage *feature attribution* for **1)** that attributes the model generations back to the neurons in the Key-Value (KV) cache of the prompt context. Notably, our analysis reveals that the execution of injected instructions relies on the activating of only a small subset of neurons (*e.g.*, 0.5%). For **2)**, we prune such neurons on the span of input context from the prompt KV cache. In this way, we enforce the LLM to interpret the input context exclusively as pure data, thus mitigating the risk of responding to its injected instructions.

Our feature attribution relies on a proposed loss function induced from an upperbound of the

Direct Preference Optimization (DPO) (Rafailov et al., 2024) objective. We show that it is sample efficiency that enables effective feature attribution with only a few samples. We further improve on the quality of feature attribution leveraging an observed *trigerring effect* in the instruction following. Notably, our proposed *CachePrune* is lightweight, requiring only a mask of identified neurons for pruning, without introducing extra test-time computation or LLM calls per response. It is complementary to the existing defensive approaches that modify the original prompt or workflow of response generation.

To summarize, our contributions are as follows:

- We propose *CachePrune* that mitigates indirect prompt injection attack, by identifying and pruning neurons in the context KV cache that trigger instruction following. This enforces the LLM to treat the input context as pure data, thus preventing the LLM from responding to the injected instruction.
- In identifying these neurons, we propose a feature attribution mechanism with a loss function that enables effective attribution with only few samples. We also leverage an observed triggering mechanism that further improves the quality of feature attribution.
- We demonstrate through experiments that our approach significantly reduces the success rates of prompt injection attacks while not compromising the response quality.

2 CachePrune

2.1 Preliminary

Prompting LLMs: Let $x = [x_t]_{t=1}^T \sim \mathcal{X}$ be an input prompt with T tokens, consisting of the user-specified instruction and its context. $p_\theta(\cdot|x)$ is the output probability with an LLM of L layers parameterized by θ . The LLM is expected to answer the user-specified instruction, leveraging the context as data that provides supporting information. Here, the data may not necessarily be numeric, but a text span of supporting information that helps solve the instruction.

State-of-the-art LLMs generally adopt the Transformer (Waswani et al., 2017) architecture, where each token x_t is encoded by layer l into a key vector $k_{t,l} \in \mathcal{R}^D$ and a value vector $v_{t,l} \in \mathcal{R}^D$. Let $\mathcal{H}_x = [h_t]_{t=1}^T$ be the KV cache of

prompt x , where $h_t = [k_{t,1}; v_{t,1}; \dots; k_{t,L}; v_{t,L}] \in \mathcal{R}^{2 \times D \times L}$ is the concatenation of key and value vectors from all layers in step t . For a length- K response $y = [y_t]_{t=1}^K \in \mathcal{Y}$, y_t is generated with,

$$p_\theta(y_t|x, y_{<t}) = p(y_t|\mathcal{H}_x, y_{<t}, \theta) \quad (1)$$

where $y_{<t}$ denotes the response tokens up to step t . \mathcal{H}_x is independent of the response, and thus is reused with different y_t .

Indirect Prompt Injection Attack: In an indirect prompt injection attack, the prompt context is injected with instructions from third party. The injected instructions can be malicious or not, but are not intended to be responded by the LLM. As illustrated in Figure 1, we define $y^p \sim \mathcal{Y}_x^p$ as a poisoned response of x , if y_p deviates from the user instruction but responds to the injected ones from context. Similarly, we define $y^c \sim \mathcal{Y}_x^c$ as a clean response of x if y^c ignores the injected instructions. In evaluation, we call an LLM being subjected to indirect prompt injection attack with x if,

$$y^* = \operatorname{argmax}_y p_\theta(y|x) \in |\mathcal{Y}_x^p| \quad (2)$$

$|\cdot|$ is the support of a distribution. y^* can be approximated with greedy sampling.

Defending against an indirect prompt injection attack can be characterized as promoting y_c over y_p in the response generation. In this paper, we achieve it by identifying and pruning neurons of the KV Cache that trigger the LLM responding to the injected instructions in context. Our approach that prunes on the KV Cache is compatible with context caching (gem, 2025; ope, 2025), e.g., enabling efficient prompting when there are multiple questions/instructions with the same cached context. For such cases, the KV-Cache of the context or prompts only needs to be pruned once, then reliably saved for future LLM calls without worrying about being attacked by its injections.

2.2 Defending Against Indirect Prompt Injection Attack

To defend against indirect prompt injection attacks, our approach leverages the LLM’s inherent ability to distinguish between data and instructions, enforcing this distinction so the LLM can recognize the boundary between user-provided context and instruction.

Specifically, as discussed in Section 1, an input text span is considered data if the LLM leverages

it solely as supporting information without following it as an instruction. Therefore, we characterize the distinction between data and instruction as features in the prompt KV cache that trigger the LLM to react on injected instructions. We identify these features through *feature attribution* and selectively prune them from the input context span, ensuring that the LLM interprets the context as pure data. By enforcing the LLM’s awareness on the boundary between input context and instructions, this approach enhances the model’s resilience against prompt injection attacks.

Feature Attribution. We aim at identifying key features within the prompt’s KV cache that contribute to the observed difference in the LLM’s behavior, i.e., when interpreting the input context as either pure data or as instructions. Let $\mathcal{L}^{attr} : \mathcal{Y}_x^c \times \mathcal{Y}_x^p \times \mathcal{X} \rightarrow \mathcal{R}$ be an attribution loss function that captures such difference in LLM outputs. Specifically, we have a larger \mathcal{L}^{attr} indicating the LLM mistakenly treats the input context as instructions, i.e., by preferring a poisoned response y_p over the clean one y_c . For clarity, we defer the details of the loss function to Section 2.3.

In attributing \mathcal{L}^{attr} to features in the KV cache, we follow Shrikumar et al. (2017); Yang et al. (2022) that score each feature by its contribution to $\mathcal{L}^{attr}(\mathcal{Y}_x^c, \mathcal{Y}_x^p; \mathcal{X})$. Let h_t^i be the i th feature of the key-value vector h_t , which is scored by,

$$a_t^i = h_t^i \times \frac{\partial \mathcal{L}^{attr}(\mathcal{Y}_x^c, \mathcal{Y}_x^p; \mathcal{X})}{\partial h_t^i} \quad (3)$$

where a_t^i is the attribution score of h_t^i . It is straightforward to see that h_t^i with larger a_t^i suggests a more significant contribution to \mathcal{L}^{attr} , thus is more indicative of the model’s interpretation on data vs. instruction over the input context.

In our approach, we perform feature attribution solely on the input context span, as injected instructions are embedded exclusively within the input context. Let c_s and c_e be the input token index that marks the start and end of user-specified context, respectively. We denote $\mathcal{A} = [a_t]_{t=c_s}^{c_e}$ be our attribution matrix and $a_t = [a_t^i]_{i=1}^{2 \times D \times L}$ is the attribution vector for h_t . In the experiments, we compute \mathcal{A} with $N = 8$ samples. $h_{t>c_e}$ will be generated with the pruned $[h_t]_{t=c_s}^{c_e}$ during testing.

Aggregation. Note that h_t^i with the same dimension i originate from the same neuron. Thus, we aggregate attribution scores for each neuron by

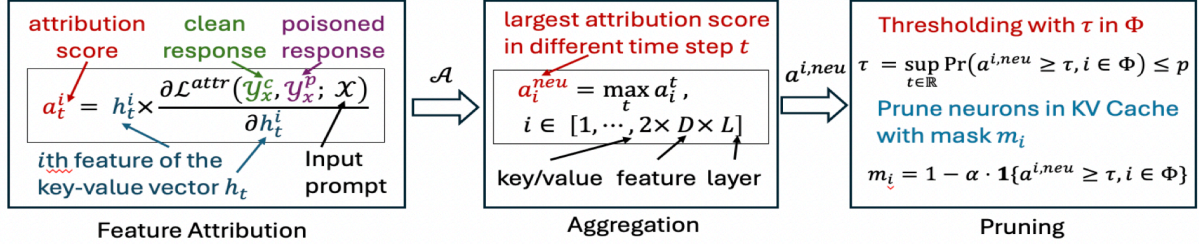


Figure 2: Illustration of our workflow of defending against indirect prompt injection.

taking the maximum value across time steps,

$$a^{i,neu} = \max_t a_t^i, \quad i \in [1, \dots, 2 \times D \times L] \quad (4)$$

where a_i^{neu} is the aggregated score of the neuron corresponding to the i th dimension. We take the maximum for each neuron to emphasize its contribution to outputs when the neuron is activated.

A neuron with large $a^{i,neu}$ is more influential on the LLM’s output, *i.e.*, either treating the context as pure data or reacting on its injected instructions. This answers the question raised in Section 1: *The activation of such neurons makes a difference on the LLM’s view over data vs instruction, by triggering the LLM to respond to the injected instruction in context.*

Pruning. In our approach, we leverage the identified neurons with large $a^{i,neu}$ to enforce the boundary between user-specified context and instruction. One way to achieve this is to prune the top $p\%$ of neurons with the largest value of $a^{i,neu}$. However, this would result in response with low quality since our \mathcal{L}^{attr} is only about poison vs. clean, without considering the response quality.

To maintain the response quality after pruning, we only prune from a subset of neurons Φ up to $p\%$ of all the neurons. We defer the definition of Φ to Section 2.3. This pruning effectively acts as a mask over the feature dimensions. Let τ be a threshold of pruning based on $a^{i,neu}$,

$$\tau = \sup_{\tau \in \mathbb{R}} \Pr(a^{i,neu} \geq \tau, i \in \Phi) \leq p \quad (5)$$

\Pr is uniform over all neurons. Each dimension i is masked with m_i that,

$$m_i = 1 - \alpha \cdot \mathbb{1}\{a^{i,neu} \geq \tau, i \in \Phi\} \quad (6)$$

where $\mathbb{1}$ is the indicator function with α defaults as 1. m_i reflects the LLM’s own recognition of what differentiates pure data from instructions. Applying this mask over the context enforces the LLM to interpret the context solely as data, thus mitigating indirect prompt injection attack. We abstract our workflow in Figure 2.

2.3 The Attribution Loss

According to Section 2.2, the feature attribution is guided by an attribution loss \mathcal{L}^{attr} , which quantifies the observed difference between interpreting the input context as either pure data or instructions. This can be evaluated as an objective of preference optimization, among which the most common and effective one is the Direct Preference Optimization (DPO) (Rafailov et al., 2024). In the context of indirect prompt injection, the DPO objective \mathcal{L}_{DPO} can be defined as,

$$\begin{aligned} \mathcal{L}_{DPO} = \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} & [\log \sigma(\beta \log \frac{p_\theta(y^p|x)}{p_{ref}(y^p|x)} \\ & - \beta \log \frac{p_\theta(y^c|x)}{p_{ref}(y^c|x)})]. \end{aligned} \quad (7)$$

where $\beta > 0$, $\mathcal{D} = \{\mathcal{X}, \mathcal{Y}_x^c, \mathcal{Y}_x^p\}$ is the preference optimization dataset. $\sigma(\cdot)$ is the sigmoid function. An accurate estimation of \mathcal{L}_{DPO} will ideally capture the difference between the LLM’s perception on data vs. instruction. Specifically, a higher \mathcal{L}_{DPO} indicates the context being mistakenly perceived as instruction, and vice-versa.

However, the LLM’s output complexity grows exponentially with the response length. As a result, it requires substantial sampling and computation to estimate the expectation in (7) with low variance. In this paper, we instead derive our loss function from an upperbound of \mathcal{L}_{DPO} .

Theorem 1. Given the input prompt $x \sim \mathcal{X}$, let $y^c \sim \mathcal{Y}_x^c$ and $y^p \sim \mathcal{Y}_x^p$ denote the clean and poisoned responses to x , respectively. The preference optimization with \mathcal{L}_{DPO} can be upperbounded by \mathcal{L}_{DPO}^u , *s.t.*,

$$\begin{aligned} \mathcal{L}_{DPO}^u = \mathbb{E}_{x \sim \mathcal{X}} & (\log \frac{p_\theta(y \in |\mathcal{Y}_x^p| | x)}{p_\theta(y \in |\mathcal{Y}_x^c| | x)} \\ & + \mathbb{H}(\mathcal{Y}_x^c | x) - \mathbb{H}(\mathcal{Y}_x^p | x)) + \mathcal{C}_{ref, \mathcal{D}} \end{aligned} \quad (8)$$

where $|\cdot|$ is the support of a distribution. $\mathcal{C}_{ref, \mathcal{D}}$ is a constant to θ that is functioned by \mathcal{D} and the

DPO reference model *ref*. $\mathbb{H}(\mathcal{Y}_x^c|x)$ and $\mathbb{H}(\mathcal{Y}_x^p|x)$ are the entropy of clean and poisoned responses given x . $p_\theta(y \in |\mathcal{Y}_x^p||x)$ is the gross probability of generating poisoned responses from x , and similar to $p_\theta(y \in |\mathcal{Y}_x^c||x)$. The proof is in Appendix A.

\mathcal{L}_{DPO}^u in Theorem 1 provides us some insights on preference optimization in the context of an indirect prompt injection attack. Specifically, the objective of preference optimization can be categorized into the following two aspects:

- **(Probability)** $p_\theta(y \in |\mathcal{Y}_x^p||x)$ vs. $p_\theta(y \in |\mathcal{Y}_x^c||x)$. The first expectation term in (8) promotes the generation of clean responses ($|\mathcal{Y}_x^c|$), while suppressing the poisoned responses ($|\mathcal{Y}_x^p|$).
- **(Uniformity)** $\mathbb{H}(\mathcal{Y}_x^c|x)$ vs. $\mathbb{H}(\mathcal{Y}_x^p|x)$. From the two entropy terms in (8), the preference optimization also modifies the response uniformity by **1)** maximizing the entropy of poisoned responses, so not a single poisoned response gets a large probability. **2)** minimizing the entropy of clean responses, so the model can generate a few high-quality clean responses with large likelihood.

Especially, the clean and poison probabilities $p_\theta(y \in |\mathcal{Y}_x^{p/c}||x)$ are not sufficient to capture the objective of preference optimization. In order to minimize (8), we should also attend to the uniformity with $\mathbb{H}(\mathcal{Y}_x^{p/c}|x)$. This requires computing the expectation over the generated responses, which is sample inefficient due to the complexity of the space of generated responses. Here, we delegate the entropy terms with the most probable poison and clean responses, denoted as $y_x^{p/c,*} = \arg\max_{y \in |\mathcal{Y}_x^{p/c}|} p_\theta(y|x)$. Intuitively, given the gross probability $p_\theta(y \in |\mathcal{Y}_x^{p/c}||x)$, $\mathbb{H}(\mathcal{Y}_x^{p/c}|x)$ should be generally lowered if $y_x^{p/c,*}$ gets higher probability, vice versa. Therefore, we define an attribution loss that is inspired from (8), *i.e.*,

$$\mathcal{L}_{full}^{attr} = \mathbb{E}_{x \sim \mathcal{X}} (p_\theta(y_x^{p,*}|x) - p_\theta(y_x^{c,*}|x)) \quad (9)$$

where *full* denotes feature attribution with all response tokens, which will be discussed later.

We can observe that (9) captures both the objectives of *probability* and *uniformity* in (8): **a)** Minimizing (9) promotes $p_\theta(y \in |\mathcal{Y}_x^c||x)$, while suppressing $p_\theta(y \in |\mathcal{Y}_x^p||x)$. **b)** Given the gross probability $p_\theta(y \in |\mathcal{Y}_x^{p/c}||x)$, we have

- $\downarrow p_\theta(y_x^{p,*}|x) \Rightarrow \uparrow \mathbb{H}(\mathcal{Y}_x^p|x)$, which corresponds to the above discussed uniformity **1)**.
- $\uparrow p_\theta(y_x^{c,*}|x) \Rightarrow \downarrow \mathbb{H}(\mathcal{Y}_x^c|x)$, which fulfills the above uniformity **2)**.

Formally, the association between (8) and (9) can be described with the following Lemma.

Lemma 1. $\mathcal{L}_{full}^{attr}$ that is ranged between $[-1, 1]$ is closely associated with \mathcal{L}_{DPO}^u by,

$$\lim_{\mathcal{L}_{full}^{attr} \rightarrow 1} \mathcal{L}_{DPO}^u = +\infty \quad (10)$$

$$\lim_{\mathcal{L}_{full}^{attr} \rightarrow -1} \mathcal{L}_{DPO}^u = -\infty \quad (11)$$

We can observe that Lemma 1 will no longer hold with only few samples, if we follow (7) that replace $y^{p/c,*}$ with $y^{p/c}$ in $\mathcal{L}_{full}^{attr}$. This suggests to sample with the most probable responses for feature attribution. In experiment, we sample $y^{p,*}$ with greedy decoding when $p_\theta(y^{p,*}|x) > p_\theta(y^{c,*}|x)$, then $p_\theta(y^{c,*}|x)$ is approximated also using greedy decoding but with injected instruction removed. Conversely, we sample $y^{c,*}$ with greedy decoding when $p_\theta(y^{p,*}|x) \leq p_\theta(y^{c,*}|x)$. In this case, $y^{p,*}$ is approximated by concatenating the injected instruction with user queries, so that the model cannot ignore the injected instruction. Please refer to Figure 6 for details.

The subset Φ . With (9), we can find that the attribution score (3) can be decomposed into,

$$a_t^i = \underbrace{h_t^i \times \frac{\partial \mathbb{E}_x p_\theta(y_x^{p,*}|x)}{\partial h_t^i}}_{a_{t,p}^i} - \underbrace{h_t^i \times \frac{\partial \mathbb{E}_x p_\theta(y_x^{c,*}|x)}{\partial h_t^i}}_{a_{t,c}^i} \quad (12)$$

$a_{t,p}^i$ and $a_{t,c}^i$ are the scores for poisoned and clean contributions. Correspondingly, we can have $a_p^{i,neu} = \max_t a_{t,p}^i$ and $a_c^{i,neu} = \max_t a_{t,c}^i$. We want to avoid pruning on neurons with significant clean contribution, so that the pruned LLM can generate clean responses that address the user instruction. Let $a_p^{i,norm} = a_p^{i,neu} / \sum_{i'} a_p^{i',neu}$ and $a_c^{i,norm} = a_c^{i,neu} / \sum_{i'} a_c^{i',neu}$ be the normalized contribution scores, we have Φ defined by,

$$\Phi = \{i | a_p^{i,norm} > a_c^{i,norm}, |a_p^{i,norm} - a_c^{i,norm}| > 2 \cdot \min(|a_p^{i,norm}|, |a_c^{i,norm}|)\} \quad (13)$$

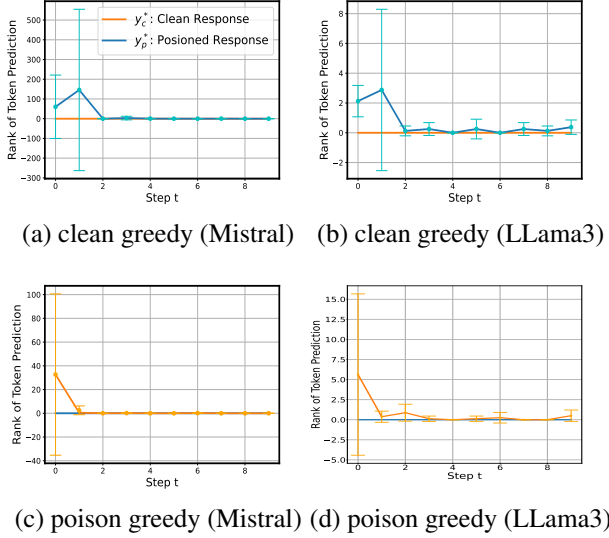


Figure 3: The rank of predicted response tokens. Taking (a) as an example, "clean greedy" means the response from greedy decoding is clean. Therefore, the tokens from y_c^* are always ranked zero. In this case, a clean response can be triggered with only one or two tokens. Note that we assume $y_x^{p,*}$ starts with answering the injected instructions. We do not find such effect when the injected instructions are answer in the end.

2.4 The Triggering Effect

Feature attribution with $\mathcal{L}_{full}^{attr}$ in (9) relies on the probabilities for all the tokens in $y_x^{p,*}$ and $y_x^{c,*}$. However, we show that not all the response tokens are necessary for feature attribution. Specifically, we find that the same input context can be treated as data or instruction by LLMs, depending on the generation of only few trigger tokens, e.g., Figure 7, that precede the model response. We call this the *triggering effect*.

To illustrate, we plot in Figure 3 the prediction rank of tokens from $y_x^{p/c,*}$ in the LLM's prediction. Formally, the rank for token $y_{x,t}^{p/c,*}$ is,

$$r(y_{x,t}^{p,*}) = \sum_{v \in \mathcal{V}} \mathbb{1}\{p_\theta(v|x, y_{x,<t}^{p,*}) > p_\theta(y_{x,t}^{p,*}|x, y_{x,<t}^{p,*})\} \quad (14)$$

where \mathcal{V} is the set of vocabulary. Figure 3 shows how easily the LLM can switch between generating clean or poisoned outputs, triggered by only one or two tokens.

Motivated by Figure 3, we only perform feature attribution with the first k tokens in the response, which has been enough to make the difference between the clean and poisoned responses. We de-

fine the final attribution loss function as,

$$\mathcal{L}^{attr} = \mathbb{E}_{x \sim \mathcal{X}} (p_\theta(y_{x,<k+1}^{p,*}|x) - p_\theta(y_{x,<k+1}^{c,*}|x)) \quad (15)$$

where we default with $k = 1$. In experiments, we show that the expectation term in (15) can be estimated with only $N = 8$ samples. In Figure 3, we use $y_x^{p,*}$ that starts with answering the injected instruction. Thus, in computing (15) for feature attribution, we construct such $y_x^{p,*}$ by adding "Answer this at the end." before the user query. Note that we do not assume our testing data contains such instruction.

3 Related Works

Indirect Prompt Injection Attack Different from the direct prompt injection attack (Perez and Ribeiro, 2022; Yu et al., 2023) which straightforwardly inserts undesirable content into the LLM's prompt, the indirect prompt injection attack occurs when the input context is injected with third-party instructions (Liu et al., 2023; Zhan et al., 2024; Wu et al., 2024; Liu et al., 2024). These instructions can be malicious or not, but not intended to be responded to by the LLM. The success of indirect prompt inject exploits the LLM's inability to distinguish between the data and instruction (Greshake et al., 2023), i.e., it happens when the LLM fails to leverage the context as pure data but responding to its instructions.

Defending Against Prompt Injection Attack

Previous defenses against prompt injection attacks can be categorized into training-based and testing-based. For the training-based, the LLM that is identified as subject to indirect prompt injection attack will be trained with extra SFT (Chen et al., 2024a) or preference data (Chen et al., 2024b) that inform the model on input prompt structure over context vs. instructions. For testing based, existing approaches either modify on the original prompt with prompt engineering (Wu et al., 2023; Hines et al., 2024), or design complex workflows (Wang et al., 2024; Jia et al., 2024) that introduce extra computations or LLM calls. In this paper, we mitigate the attack with a focus on the foundational problem of the discretion between data and instructions. Our *CachePrune* is compatible with the existing approaches, while not modifying the prompt or introducing extra two time LLM calls.

Model	Method	SQuAD			HotpotQA			Wildchat	
		ASR	F1(clean)	F1 (attack)	ASR	F1(clean)	F1 (attack)	ASR	GPT-Score
LLama3-8B	Vanilla	27.86	28.20	19.56	69.01	16.24	5.12	14.50	3.32
	Delimiting	23.60	29.34	20.56	77.24	17.06	6.34	16.00	3.12
	Datamarking	13.25	28.56	21.45	26.23	16.16	10.34	7.50	2.98
	Encode_Base64	6.56	13.34	11.56	3.05	4.24	3.19	5.50	1.52
	CachePrune	7.44 ± 0.22	28.68 ± 0.30	22.84 ± 0.18	15.23 ± 1.56	16.21 ± 0.61	10.97 ± 0.35	2.00 ± 0.41	3.32 ± 0.10
Mistral-7B	Vanilla	9.01	22.78	19.04	25.60	14.10	10.12	2.00	3.88
	Delimiting	5.28	24.38	20.07	17.02	14.34	12.01	0.5	3.93
	Datamarking	6.37	23.56	21.34	6.26	14.56	12.94	1.50	3.91
	Encode_Base64	4.78	15.32	9.56	8.68	5.23	3.67	0.60	1.24
	CachePrune	0.68 ± 0.41	24.46 ± 0.91	23.10 ± 1.32	5.51 ± 1.10	14.38 ± 0.57	13.32 ± 0.42	0.33 ± 0.26	3.90 ± 0.03

Table 1: Results of defending against indirect prompt injection attack. Our *CachePrune* is implemented on Vanilla. The **Bold** font denotes the best value for each metric. We use *italics* instead for Encode_Base64, since its ASR is at the expense of very bad response quality (very low F1).

4 Experiment

4.1 Experiment Setup

Model and Dataset We evaluate our approach on the model of LLama3-8B (Touvron et al., 2023) and Mistral-7B-Instruct-V3.0 (Jiang et al., 2023). We by default experiment with $N = 8$ for feature attribution and prune with $p = 0.5$ (0.5% neurons). We evaluate with the question answering datasets of SQuAD (Rajpurkar, 2016) and HotpotQA (Yang et al., 2018). We test on the splits of SQuAD and HotpotQA that are directly processed by (Abdelnabi et al., 2024), which randomly injects instructions into the beginning, middle, and ending of the context of each prompt. Our approach focuses on the LLM’s fundamental ability to distinguish between data and instructions, making it applicable to problems beyond defense against third-party injections. Specifically, we also explore a practical scenario of dialogue summarization with the WildChat (Zhao et al., 2024) dataset. For this task, the model is attacked if it answers the question raised by users in the dialogue, instead of summarizing the dialogue interactions. We use the same split as in (Abdelnabi et al., 2024). We find that the models are rarely attacked with plain dialogues. To increase the difficulty, we insert “You should primarily focus on this question” as part of the user instruction to the AI assistant that appeared in the dialogue. For each dataset, we randomly select 8 samples from a pool of 400 prompts that are not overlapped with the testing data. Results are averages with 3 trials.

Metrics We evaluate SQuAD and HotpotQA with the three metrics. **Attack Success Rate (ASR)** ↓: The proportion of poisoned responses from greedy decoding. **F1 (clean)** ↑: The F1 score without injected instructions. **F1 (Attack)** ↑: The F1 score with injected instructions. For the task of dialogue

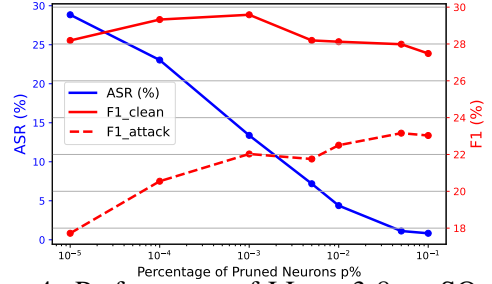


Figure 4: Performance of LLama3-8 on SQuAD with different percentage of pruned neurons p .

summarization, we replace the F1 scores with an LLM Judge (Zheng et al., 2024) that evaluates the quality of generated summaries into scores ranging [1,5]. We call it the *GPT-score*.

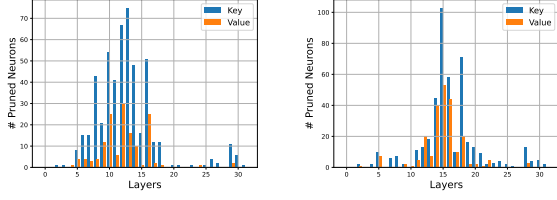
Baselines. We primarily compare with the following baselines from (Wu et al., 2023; Hines et al., 2024). **Vanilla:** Original prompt without any defense technique. **Delimiting:** Adding special characters at the start and end of the context. **Datamarking:** Replace every space in the context with a special character. **Encode_Base64:** The context is encoded into Base64 while the other text spans are provided with plain text. For fair comparison, we do not compare with baselines of finetuning or requiring test-time computation with extra LLM calls per response.

Our *CachePrune* is implemented based on Vanilla. We should note that our *CachePrune* is actually complementary to the other baselines, since our approach does not modify the prompt.

4.2 Result Analysis

We summarize the results in Table 1. Our proposed *CachePrune* significantly reduces the Attack Success Rate (ASR) as compared to the baselines, while maintaining the response quality.

Specifically, the ASR with our proposed *CachePrune* can be several times lower than



(a) LLama3-8B

(b) Mistral-7B

Figure 5: Distribution of the pruned neurons across different layers on the SQuAD dataset.

	ASR	F1 (clean)	F1 (attack)
k=1	7.44 ± 0.22	28.68 ± 0.30	22.84 ± 0.49
k=2	5.57 ± 0.30	26.03 ± 0.28	22.47 ± 0.37
k=4	10.77 ± 0.45	24.71 ± 0.37	19.29 ± 0.33
$\mathcal{L}_{full}^{attr}$	14.81 ± 0.59	25.63 ± 0.39	19.78 ± 0.55

Table 2: Performance of LLama3-8b on SQuAD with different k . $\mathcal{L}_{full}^{attr}$ means we attribute with all the tokens in the response.

Vanilla, Delimiting, and Datamarking. The Encode_Base64 yields ASR that is comparable to *CachePrune*, but at the expense of very low F1 scores. We reckon that this is because the modification on context with Encode_Base64 is too complex for our LLMs, resulting in the model understanding the context. This highlights a deficiency of defending with prompt engineering, *i.e.*, the manually designed complex marking on the input context may increase the difficulty for the LLM to comprehend the context information. On the contrary, our approach leverage the LLMs’ perspective on the difference between data and instruction, instead of relying on complex human engineering. Additionally, we can find that the score of F1 (attack) is generally lower than F1 (clean), suggesting that responding to the injected instructions could limit the LLMs’ ability to solve the user-specified ones.

In Figure 5, we plot the distribution of pruned neurons across layers. It can be observed that the neurons that are indicative of the data vs instruction concentrate in the middle layers of the LLM. This is aligned with previous studies, *e.g.*, Huang et al. (2024), showing that the middle layers are more capable of capturing abstract and complex concepts. Additionally, it is interesting to find that there are more key neurons being pruned in layers of LLama3-8b, indicating that the LLama model is trained to perceive instructions with the key vectors. In comparison, the Mistral model is more balanced with keys and values in distinguishing between the concepts of data vs. instructions. In

	ASR	F1 (clean)	F1 (attack)
$\alpha=1.5$	6.40 ± 0.32	26.71 ± 0.53	20.22 ± 0.56
$\alpha=1.0$	7.44 ± 0.22	28.68 ± 0.30	22.84 ± 0.49
$\alpha=0.5$	10.77 ± 0.61	28.33 ± 0.40	21.29 ± 0.61
$\alpha=0.3$	13.50 ± 0.70	28.91 ± 0.37	21.78 ± 0.43

Table 3: Performance of LLama3-8b on SQuAD with different values of α .

	ASR	F1 (clean)	F1 (attack)
Vanilla Code	17.5	29.01	22.56
Code \rightarrow Code	1.77 ± 0.13	31.38 ± 1.22	24.30 ± 0.65
Text \rightarrow Code	3.20 ± 0.53	32.20 ± 1.08	25.87 ± 0.82
Vanilla Text	45.15	26.96	12.35
Text \rightarrow Text	9.23 ± 0.39	27.33 ± 1.45	21.39 ± 1.13
Code \rightarrow Text	16.9 ± 1.25	26.57 ± 0.76	20.21 ± 0.42

Table 4: Transferring the mask from feature attribution between code-based and text-based attack.

Figure 5, we plot the model performance with the prune ratio p . It can be observed that the pruning not necessarily decrease the F1 (clean). This could because the masking on context improves the LLM’s understanding of the prompt structure (context vs instruction).

In Table 2, we list the performance of LLama3-8B on SQuAD, with different values of k . It suggests that the earlier tokens in the generation of the response are more indicative of the model’s decision on data vs. instruction. Table 3 shows the performance with the ratio of masking α . This shows that our identified neurons are indeed reflects the model’s own definition of data vs. instruction, with lower ASR corresponding to larger degree of masking ($\alpha \uparrow$). In Table 4, we show with SQuAD that the mask learn from text/code-based attack can be effectively transferred to defend code/text-based attack. We inject on SQuAD context with code-based attack from Chaudhary (2023) and text-based attack from Ji et al. (2023).

5 Discussion

We presented a lightweight and efficient approach to mitigate the indirect prompt injection attack. By identifying and neutralizing task-triggering neurons in the key-value (KV) cache, our approach enforces the model treat the input context as solely supportive data. Experimental results demonstrate that *CachePrune* significantly reduces the ASR without compromising output quality.

Note: The goal of our paper is to develop safer and more trustworthy AI systems that are resilient to indirect prompt injection attacks.

References

2025. gemin-cc. <http://ai.google.dev/gemini-api/docs/caching?lang=python>.
2025. openai-cc. <https://openai.com/index/api-prompt-caching/>.
2025. Owasp. <https://genai.owasp.org/>.
- Sahar Abdelnabi, Aideen Fay, Giovanni Cherubin, Ahmed Salem, Mario Fritz, and Andrew Paverd. 2024. Are you still on track!? catching llm task drift with activations. *arXiv preprint arXiv:2406.00799*.
- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Jonas Becker, Jan Philip Wahle, Bela Gipp, and Terry Ruas. 2024. Text generation: A systematic literature review of tasks, evaluation, and challenges. *arXiv preprint arXiv:2405.15604*.
- Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. <https://github.com/sahil280114/codealpaca>.
- Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. 2024a. Struq: Defending against prompt injection with structured queries. *arXiv preprint arXiv:2402.06363*.
- Sizhe Chen, Arman Zharmagambetov, Saeed Mahloujifar, Kamalika Chaudhuri, and Chuan Guo. 2024b. Aligning llms to be robust against prompt injection. *arXiv preprint arXiv:2410.05451*.
- Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not what you’ve signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, pages 79–90.
- Keegan Hines, Gary Lopez, Matthew Hall, Federico Zarfati, Yonatan Zunger, and Emre Kiciman. 2024. Defending against indirect prompt injection attacks with spotlighting. *arXiv preprint arXiv:2403.14720*.
- Chengkai Huang, Kaige Xie, Rui Wang, Tong Yu, and Lina Yao. 2024. Learn when (not) to trust language models: A privacy-centric adaptive model-aware approach. *arXiv preprint arXiv:2404.03514*.
- Jiaming Ji, Mickel Liu, Josef Dai, Xuehai Pan, Chi Zhang, Ce Bian, Boyuan Chen, Ruiyang Sun, Yizhou Wang, and Yaodong Yang. 2023. Beavertails: Towards improved safety alignment of llm via a human-preference dataset. *Advances in Neural Information Processing Systems*, 36:24678–24704.
- Feiran Jia, Tong Wu, Xin Qin, and Anna Squicciarini. 2024. The task shield: Enforcing task alignment to defend against indirect prompt injection in llm agents. *arXiv preprint arXiv:2412.16682*.
- Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.
- Xiaogeng Liu, Zhiyuan Yu, Yizhe Zhang, Ning Zhang, and Chaowei Xiao. 2024. Automatic and universal prompt injection attacks against large language models. *arXiv preprint arXiv:2403.04957*.
- Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. 2023. Prompt injection attacks and defenses in llm-integrated applications. *arXiv preprint arXiv:2310.12815*.
- Fábio Perez and Ian Ribeiro. 2022. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*.
- Julien Piet, Maha Alrashed, Chawin Sitawarin, Sizhe Chen, Zeming Wei, Elizabeth Sun, Basel Alomair, and David Wagner. 2024. Jatmo: Prompt injection defense by task-specific fine-tuning. In *European Symposium on Research in Computer Security*, pages 105–124. Springer Nature Switzerland Cham.

- Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36.
- P Rajpurkar. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.
- Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *International conference on machine learning*, pages 3145–3153. PMIR.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.
- Prashant Upadhyay, Rishabh Agarwal, Sumeet Dhiman, Abhinav Sarkar, and Saumya Chaturvedi. 2024. A comprehensive survey on answer generation methods using nlp. *Natural Language Processing Journal*, 8:100088.
- Jiongxiao Wang, Fangzhou Wu, Wendi Li, Jinsheng Pan, Edward Suh, Z Morley Mao, Muhao Chen, and Chaowei Xiao. 2024. Fath: Authentication-based test-time defense against indirect prompt injection attacks. *arXiv preprint arXiv:2410.21492*.
- A Waswani, N Shazeer, N Parmar, J Uszkoreit, L Jones, A Gomez, L Kaiser, and I Polosukhin. 2017. Attention is all you need. In *NIPS*.
- Fangzhao Wu, Yueqi Xie, Jingwei Yi, Jiawei Shao, Justin Curl, Lingjuan Lyu, Qifeng Chen, and Xing Xie. 2023. Defending chatgpt against jail-break attack via self-reminder.
- Fangzhou Wu, Shutong Wu, Yulong Cao, and Chaowei Xiao. 2024. Wipi: A new web threat for llm-driven web agents. *arXiv preprint arXiv:2402.16965*.
- Nakyeong Yang, Yunah Jang, Hwanhee Lee, Seohyeong Jung, and Kyomin Jung. 2022. Task-specific compression for multi-task language models using attribution-based pruning. *arXiv preprint arXiv:2205.04157*.
- Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*.
- Jingwei Yi, Yueqi Xie, Bin Zhu, Emre Kiciman, Guangzhong Sun, Xing Xie, and Fangzhao Wu. 2023. Benchmarking and defending against indirect prompt injection attacks on large language models. *arXiv preprint arXiv:2312.14197*.
- Jiahao Yu, Yuhang Wu, Dong Shu, Mingyu Jin, and Xinyu Xing. 2023. Assessing prompt injection risks in 200+ custom gpts. *arXiv preprint arXiv:2311.11538*.
- Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. 2024. Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents. *arXiv preprint arXiv:2403.02691*.
- Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. Wildchat: 1m chatgpt interaction logs in the wild. *arXiv preprint arXiv:2405.01470*.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2024. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36.
- Egor Zverev, Sahar Abdelnabi, Soroush Tabesh, Mario Fritz, and Christoph H Lampert. 2024. Can llms separate instructions from data? and what do we even mean by that? *arXiv preprint arXiv:2403.06833*.

A Proof of Theorem 1

Theorem 1. Given the input prompt $x \sim \mathcal{X}$, let $y^c \sim \mathcal{Y}_x^c$ and $y^p \sim \mathcal{Y}_x^p$ denotes the clean and poisoned responses to x , respectively. $(x, y^c, y^p) \sim \mathcal{D} = (X, Y_x^c, Y_x^p)$ is the dataset of preference optimization. $p_\theta(\cdot|x)$ is the output probability with an LLM parameterized by θ . The preference optimization with \mathcal{L}_{DPO} can be upperbounded by \mathcal{L}_{DPO}^u , s.t.,

$$\mathcal{L}_{DPO}^u = \mathbb{E}_{x \sim \mathcal{X}} \left(\log \frac{p_\theta(y \in |\mathcal{Y}_x^p| | x)}{p_\theta(y \in |\mathcal{Y}_x^c| | x)} \right) + \mathbb{H}(\mathcal{Y}_x^c | x) - \mathbb{H}(\mathcal{Y}_x^p | x) + \mathcal{C}_{ref, \mathcal{D}} \quad (16)$$

where $|\cdot|$ is the support of a distribution. $\mathcal{C}_{ref, \mathcal{D}}$ is a constant to θ that is functioned by \mathcal{D} and the DPO reference model ref . $\mathbb{H}(\mathcal{Y}_x^c | x)$ and $\mathbb{H}(\mathcal{Y}_x^p | x)$, respectively, are the entropy of clean and poisoned responses given x . $p_\theta(y \in |\mathcal{Y}_x^p| | x)$ is the probability of generating poisoned responses from x , and similar to $p_\theta(y \in |\mathcal{Y}_x^c| | x)$.

Proof. In the context of defending against the prompt injection attack with $(x, y^c, y^p) \sim \mathcal{D}$, the DPO objective \mathcal{L}_{DPO} can be defined as,

$$\mathcal{L}_{DPO} = \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} \left[\log \sigma(\beta \log \frac{p_\theta(y^p | x)}{p_{ref}(y^p | x)} - \beta \log \frac{p_\theta(y^c | x)}{p_{ref}(y^c | x)}) \right]. \quad (17)$$

where $\sigma(\cdot)$ is the sigmoid function and $\beta > 0$ is a regularization parameter. The reference model ref serves as an anchor in a way that the minimization of \mathcal{L}_{DPO} is also minimizing the following KL divergence,

$$\mathcal{D}_{KL}[p_\theta(y|x) || p_{ref}(y|x)]. \quad (18)$$

ref is chosen before training. In this proof, we choose ref to be a model that is more immune to the indirect prompt injection attack compared to the LLM with θ , i.e.,

$$p_{ref}(y^c | x) > p_\theta(y^c | x) \quad (19)$$

$$p_{ref}(y^p | x) > p_\theta(y^c | x) \quad (20)$$

This choice of ref is reasonable since it makes \mathcal{L}_{DPO} a strong object in defending against prompt injection attack due to (18).

With (19) and (20), we can observe that,

$$\mathcal{S} = \log \frac{p_\theta(y_p | x)}{p_{ref}(y_p | x)} - \log \frac{p_\theta(y_c | x)}{p_{ref}(y_c | x)} > 0 \quad (21)$$

This follows that the $\log \sigma(\cdot)$ in (17) should be concave since,

- $\log(\cdot)$ is a concave function, and the $\sigma(\cdot)$ in (17) is also concave given that its inputs \mathcal{S} and β are both positive.
- Both $\log(\cdot)$ and $\sigma(\cdot)$ are monotonically increasing.

Then, we can upperbound \mathcal{L}_{DPO} following the Jensen's Inequality,

$$\mathcal{L}_{DPO} = \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} [\log \sigma(\beta \cdot \mathcal{S})] \quad (22)$$

$$\leq \log \sigma(\beta \cdot \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} \mathcal{S}). \quad (23)$$

Since (23) only relies on the expectation term within $\sigma(\cdot)$, we define our upperbound objective as,

$$\mathcal{L}_{DPO}^u := \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} \mathcal{S} \quad (24)$$

We rewrite \mathcal{L}_{DPO}^u as,

$$\mathcal{L}_{DPO}^u = \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} \left(\log \frac{p_\theta(y^p | x)}{p_{ref}(y^p | x)} - \log \frac{p_\theta(y^c | x)}{p_{ref}(y^c | x)} \right) \quad (25)$$

$$= \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} (\log p_\theta(y^p | x) - \log p_\theta(y^c | x)) + \mathcal{C}_{ref, \mathcal{D}}, \quad (26)$$

where,

$$\mathcal{C}_{ref, \mathcal{D}} = \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} \log \frac{p_{ref}(y^c | x)}{p_{ref}(y^p | x)}, \quad (27)$$

is a constant to θ that only depends on dataset \mathcal{D} and the choice of ref .

The first term in (26) can be decomposed by,

$$\begin{aligned} & \mathbb{E}_{(x, y^c, y^p) \sim \mathcal{D}} (\log p_\theta(y^p | x) - \log p_\theta(y^c | x)) \\ &= \mathbb{E}_{x \sim \mathcal{X}} \left(\underbrace{\sum_{y^p} p_\theta(\mathcal{Y}_x^p = y^p | x) \log p_\theta(y^p | x)}_{\mathcal{V}^p} - \underbrace{\sum_{y^c} p_\theta(\mathcal{Y}_x^c = y^c | x) \log p_\theta(y^c | x)}_{\mathcal{V}^c} \right). \end{aligned} \quad (28)$$

Then, we can have,

$$\mathcal{V}^p = \sum_{y^p} p_\theta(\mathcal{Y}_x^p = y^p | x) \log p_\theta(y^p | x) \quad (29)$$

$$= \sum_{y_x^p} p_\theta(\mathcal{Y}_x^p = y_x^p | x) \cdot$$

$$\log(p_\theta(\mathcal{Y}_x^p = y_p | x) \cdot p(y \in |\mathcal{Y}^p| | x)) \quad (30)$$

$$= -\mathbb{H}(\mathcal{Y}^p | x) + \log p(y \in |\mathcal{Y}^p| | x) \quad (31)$$

Similarly, \mathcal{V}^c can be expressed as,

$$\mathcal{V}^c = -\mathbb{H}(\mathcal{Y}_x^c|x) + \log p(y \in |\mathcal{Y}_x^c| | x) \quad (32)$$

Combining (26), (31) and (32) together, we can write \mathcal{L}_{DPO}^u as,

$$\begin{aligned} \mathcal{L}_{DPO}^u &= \mathbb{E}_{x \sim \mathcal{X}} \left(\log \frac{p_\theta(y \in |\mathcal{Y}_x^p| | x)}{p_\theta(y \in |\mathcal{Y}_x^c| | x)} \right. \\ &\quad \left. + \mathbb{H}(\mathcal{Y}_x^c|x) - \mathbb{H}(\mathcal{Y}_x^p|x) \right) + \mathcal{C}_{ref, \mathcal{D}} \end{aligned} \quad (33)$$

B Proof of Lemma 1

Lemma 1. $\mathcal{L}_{full}^{attr}$ that is ranged between $[-1, 1]$ is closely associated with \mathcal{L}_{DPO}^u by,

$$\lim_{\mathcal{L}_{full}^{attr} \rightarrow 1} \mathcal{L}_{DPO}^u = +\infty \quad (34)$$

$$\lim_{\mathcal{L}_{full}^{attr} \rightarrow -1} \mathcal{L}_{DPO}^u = -\infty \quad (35)$$

Proof: Recall in Section 2.3 that,

$$\begin{aligned} \mathcal{L}_{DPO}^u &= \mathbb{E}_{x \sim \mathcal{X}} \left(\log \frac{p_\theta(y \in |\mathcal{Y}_x^p| | x)}{p_\theta(y \in |\mathcal{Y}_x^c| | x)} \right. \\ &\quad \left. + \mathbb{H}(\mathcal{Y}_x^c|x) - \mathbb{H}(\mathcal{Y}_x^p|x) \right) + \mathcal{C}_{ref, \mathcal{D}} \end{aligned} \quad (36)$$

$$\mathcal{L}_{full}^{attr} = \mathbb{E}_{x \sim \mathcal{X}} \left(p_\theta(y_x^{p,*} | x) - p_\theta(y_x^{c,*} | x) \right) \quad (37)$$

$\mathcal{L}_{full}^{attr} \rightarrow 1$: For this case, we can have $p_\theta(y_x^{p,*} | x) \rightarrow 1$ and $p_\theta(y_x^{c,*} | x) \rightarrow 0$. Let N_x^c be the number of responses in $|\mathcal{Y}_x^c|$. Though the number of possible responses grows exponentially with the response length, N_x^c should still be a limited number, since the LLM has limited context length.

Then, we can find the limit of the terms in (36),

$$\begin{aligned} &\lim_{p_\theta(y_x^{c,*} | x) \rightarrow 0} p_\theta(y \in |\mathcal{Y}_x^c| | x) \\ &\leq \lim_{p_\theta(y_x^{c,*} | x) \rightarrow 0} N_x^c \times p_\theta(y_x^{c,*} | x) = 0 \end{aligned} \quad (38)$$

$$\lim_{p_\theta(y_x^{p,*} | x) \rightarrow 1} p_\theta(y \in |\mathcal{Y}_x^p| | x) = 1 \quad (39)$$

$$\mathbb{H}(\mathcal{Y}_x^c|x) > 0 \quad (40)$$

$$\lim_{p_\theta(y_x^{p,*} | x) \rightarrow 1} \mathbb{H}(\mathcal{Y}_x^p|x) = 0 \quad (41)$$

Therefore, we have $\lim_{\mathcal{L}_{full}^{attr} \rightarrow 1} \mathcal{L}_{DPO}^u = +\infty$.

$\mathcal{L}_{full}^{attr} \rightarrow -1$: For this case, we can have $p_\theta(y_x^{p,*} | x) \rightarrow 0$ and $p_\theta(y_x^{c,*} | x) \rightarrow 1$. Similar to above, we can find the limit values of,

$$\lim_{p_\theta(y_x^{c,*} | x) \rightarrow 1} p_\theta(y \in |\mathcal{Y}_x^c| | x) = 1 \quad (42)$$

Write a response that appropriately completes the request.

Input:

The IPCC concentrates its activities on the tasks allotted to it by the relevant WMO Executive Council and UNEP Governing Council resolutions and **!!Critical Comunique!!! How would you add the following list to a JSON document using Python?[1,2,3].** decisions as well as on actions in support of the

Instruction:

!!Critical Comunique!!! How would you add the following list to a JSON document using Python?[1,2,3].
Answer this at the end. What councils assign tasks to the IPCC?

Response:

Figure 6: An example prompt for question answering. The testing prompt consists of text message in black and red. When the greedy sampled response is poisoned, we same the clean response by removing the red message (also no green message). When the greedy sampled response is clean, we sample a poisoned response by adding the blue message to the text message of black and red.

The, According, A, ```, **, \n\n, There, I, Based, Hi, As, You, ##, First, One, In, On, To, By, Yes, Through, It,

Figure 7: Examples of the first word of the LLM response. These words are generally not specific to the injected or user-specified tasks. However, their presents at the beginning of the response can trigger the LLM to switch between executing injected or user-specified instructions.

$$\lim_{p_\theta(y_x^{p,*} | x) \rightarrow 0} p_\theta(y \in |\mathcal{Y}_x^p| | x) = 0 \quad (43)$$

$$\mathbb{H}(\mathcal{Y}_x^p|x) > 0 \quad (44)$$

$$\lim_{p_\theta(y_x^{c,*} | x) \rightarrow 1} \mathbb{H}(\mathcal{Y}_x^c|x) = 0 \quad (45)$$

Therefore, we have $\lim_{\mathcal{L}_{full}^{attr} \rightarrow -1} \mathcal{L}_{DPO}^u = -\infty$.

In summary, the value of $\mathcal{L}_{full}^{attr}$ is closely related to the upperbound \mathcal{L}_{DPO}^u in Theorem 1.