USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

HijackKV: New Threat in Position-Independent KV Cache Reuse

Yichi Zhang and Zhiqi Wang, *The Pennsylvania State University;*
Huan Zhang, *University of Illinois Urbana-Champaign;*
Yuchen Yang, *The Pennsylvania State University*

https://www.usenix.org/conference/usenixsecurity26/presentation/zhang-yichi

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

HIJACKKV: New Threat in Position-Independent KV Cache Reuse

Yichi Zhang[1], Zhiqi Wang[1], Huan Zhang[2], Yuchen Yang[1]
[1]*The Pennsylvania State University,* [2]*University of Illinois Urbana-Champaign*
{yichi.zhang, zhiqi.wang, yuchen.yang}@psu.edu, huan@huan-zhang.com

Abstract

Key-Value (KV) cache reduces inference latency in large language models (LLMs). Traditional prefix-based reuse has low cache hit rates across inference requests because it requires exact token and position matches. To improve efficiency, recent system optimizations introduce position-independent KV reuse, allowing KV cache to be reused whenever identical text chunks appear, regardless of their position in the sequence.

We show this design introduces a new threat, *KV Cache Hijacking*. Since KV caches are retrieved by token match but encode the context in which they were originally computed, the KV tied to a benign-looking token chunk may encode an attacker-controlled prefix. When later reused in a victim query, this contaminated KV silently hijacks the model's behavior, even if *no attacker-controlled text appears in the input.*

We introduce HIJACKKV, the *first* attack framework that systematically exploits this vulnerability, demonstrating its severity and practicality. HIJACKKV optimizes an attacker-controlled prefix, so that the KV computed for a subsequent common benign text encodes the attacker's goal, while the text remains unchanged for future cache hits. HIJACKKV achieves an average 94% success rate in a single attempt, remains effective under realistic constraints including low hit rates (10%) and frequent recomputation (50%), persists over multi-turn interactions, and transfers across models in black-box settings. We further provide design insights for building secure KV reuse systems.[1]

1 Introduction

LLMs are now widely deployed in personal assistance [35,41], healthcare [33,44], and retrieval-augmented applications [6, 22], where user queries are often augmented with long contextual input for accurate responses. However, long inputs slow down inference because the model must perform a full prefill pass to build the Key-Value (KV), which dominates

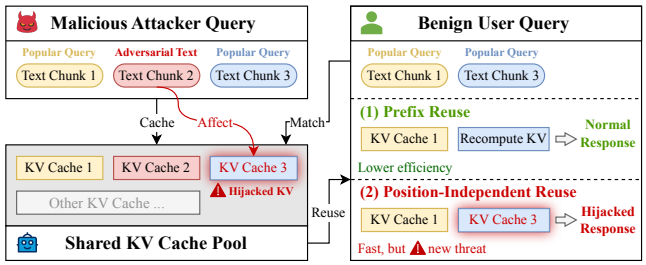

Figure 1: **New threat** from position-independent KV reuse in multi-tenant systems. **(1) Prefix reuse is slow** because reuse requires the same preceding prefix. **(2) Position-independent reuse is fast but unsafe** because it will reuse a hijacked KV despite position mismatch, enabling cross-user attacker-controlled outputs without any adversarial text.

the time-to-first-token (TTFT) and scales super-linearly with input length [5, 20]. To reduce this overhead, modern inference engines increasingly rely on caching and reusing KV across requests, as adopted by major providers, *e.g.*, Anthropic Claude [1], Google Gemini [8] and OpenAI ChatGPT [34]. Traditional reuse is strictly **prefix-based** [38], requiring exact matches of both tokens and positions, resulting in a low hit rate [62], and provides marginal benefit compared with no KV cache reuse. Figure 1(b)(1) shows that reuse requires the same preceding prefix: KV3 was cached after chunk 2, so if chunk 2 is missing in the future query, KV3 cannot be reused even if chunk 3's text is identical.

Recent system research proposes efficiency-driven optimizations that relax prefix and position constraints, including **(i) position-independent KV reuse** [14,49,57,58,62], which enables chunk-level reuse if the tokens match, regardless of original prefix. Position-independent reuse is already deployed in commercial platforms such as LMCache [28] (commercializing the award-winning research prototype CacheBlend [62]), as well as a growing body of follow-up work [14,49,57]; and **(ii) multi-tenant KV reuse** [48,53], where KV states computed for one user's request are shared to accelerate independent requests from different users that con-

[1]The source code of HIJACKKV is publicly available: https://github.com/YichiCS/KV-Cache-Hijack

tain overlapping context. Widely adopted open-source serving engines such as vLLM [48], used by organizations including Cloudflare and IBM, enable shared KV reuse by default. In practice, users frequently submit overlapping text (e.g., retrieved documents, organizational knowledge bases, or public materials), making cache reuse common and predictable.

While significantly improving efficiency, these optimizations assume that a text chunk's KV state remains invariant across different preceding contexts. However, KV states in Transformers are inherently *context-dependent*, and the same text chunk can produce different internal states under different contexts. Prior work views this misalignment as a utility issue caused by attention shift, and mitigates it via selective recomputation [14, 62], refreshing 10-20% KV under the new context to recover accuracy. In contrast, we reveal this misalignment as a **critical security vulnerability**: *by decoupling KV states from their causal context, modern reuse mechanisms inadvertently create a channel for adversarial KV cache hijacking.* We formalize this security vulnerability in Section 4.2. Next, we demonstrate that this channel is not only theoretical but can be exploited in realistic deployments.

We propose HIJACKKV, the *first* KV cache hijacking attack against position-independent KV cache systems under multi-tenant deployment, to demonstrate that an unprivileged attacker can stealthily alter the LLM inference behavior of other users. As shown in Figure 1(a), an attacker can prepend a commonly reused benign Text Chunk 3 (e.g., company FAQs) with an adversarially optimized prefix Text Chunk 2. This prefix encodes a hidden *adversarial objective* by conditioning the KV Cache 3 of the benign chunk 3 on the Text Chunk 2. Under position-independent reuse (Figure 1(b)), a subsequent benign user query may trigger reuse of the KV Cache 3 despite the absence of its preceding prefix (text chunk 2), enabling *silent cross-user output hijacking* without any adversarial tokens appearing in the benign user's input. This is the key difference from prompt injection [25], where adversarial tokens must appear in the victim's input, either directly or via retrieved content.

HIJACKKV is both powerful and practical in terms of the following aspects. *(1) Attack success:* it attains an average 94% attack success rate with a single attempt. *(2) Robustness and Practice:* it remains effective under realistic deployment conditions, even when the match chunk hit rate is as low as 10% and when recomputation is enabled at levels up to 50%. *(3) Persistence:* it retains its impact even when over 1,000 tokens of new, unrelated context are inserted, indicating that a single adversarial injection can persist through multi-turn interactions. *(4) Transferability:* it demonstrates strong cross-model transferability under a black box threat model, as the same malicious prefix succeeds across multiple models.

Contribution. We make the following contributions:

- We identify and formalize a new threat introduced by efficiency-oriented KV reuse optimization.
- We design HIJACKKV, the *first* attack framework that hijacks KV reuse, revealing a previously unknown vulnerability in modern LLM serving systems.
- We evaluate state-of-the-art KV reuse mechanisms across multiple models and benchmarks, showing that they remain vulnerable even with selective recomputation and low cache hit rates, and offer suggestions for building secure-by-design KV cache systems.

2 Related Work

2.1 Efficiency Optimization for KV Cache

KV Cache Reuse and Sharing. To address the low hit rate of prefix KV reuse [38] and reduce the computational cost across multiple requests, recent systems have explored cache reuse and sharing strategies for multi-tenant LLM serving.

For reuse, CacheBlend [62] addresses KV reuse in RAG, which enables chunk-level reuse of precomputed KV caches regardless of their positions and selectively recomputes KV values for a small subset of tokens to mitigate utility loss. EPIC [14] and MEPIC [49] formalize Position-Independent KV Cache by introducing the LegoLink algorithm to mitigate attention sink effects, and add memory-efficient optimizations such as chunk-level paging and RoPE fusion. KVShare [57] designs a dual-stage high deviation algorithm that selectively recomputes KV cache during prefill and decode phases via a cache-aware scheduler. *For sharing,* PagedAttention [20] enables multi-tenant KV cache sharing via zero-copy mapping of logical prompts to the shared physical KV blocks. SGLang [64] stores the KV cache in GPU memory with a Radix Tree to maximize sharing across requests.

While these systems improve service efficiency, their security implications remain largely unexplored. *Our work reveals that these efficiency-driven optimizations open a hidden channel for KV cache hijacking.*

KV Cache Compression. Compression aims to reduce the storage overhead of the KV cache. The first line of work focuses on quantization [12, 27], which demonstrates that quantization to low-bit representations (e.g., 4-bit or even 2-bit) maintains generation quality while significantly reducing memory consumption. The second approach targets the selective pruning of less important KV states. Early strategies keep only the most recent tokens or preserve initial tokens alongside recent context [54]. More advanced methods leverage attention score statistics to identify and evict tokens with minimal impact on subsequent generations [26, 63]. Several KV-level defenses [17, 50] also adopt pruning to limit misuse. *However, simply applying KV cache compression does not prevent KV cache hijacking as shown in Section 8.*

2.2 Adversarial Attacks

KV Cache Privacy Leakage Attack. Sharing the KV cache across multi-tenant LLM serving introduces privacy vulnerabilities, allowing adversaries to reconstruct sensitive user inputs via multiple attack vectors. Wu *et al.* [53] present the first systematic investigation of security risks in multi-tenant LLM serving with KV cache sharing in frameworks like SGLang [64] and vLLM [20]. Luo *et al.* [29] study KV cache privacy leakage under three attack paradigms and identify the KV cache as a privacy-critical component. Song *et al.* [45] discover timing side channels in LLM serving systems arising from shared caches and GPU memory allocations, which can be exploited to infer both confidential system prompts and sensitive requests issued by other users.

While prior works have primarily focused on the privacy issues associated with KV caches, *our work investigates how to exploit the KV cache to hijack model outputs.*

KV Cache Modification Attack. Beyond passive privacy leakage, recent work explores active attacks that modify the KV cache to change model behavior. CacheTrap [31] introduce a trojan attack that corrupts value vectors in the KV cache through bit-flip operations, achieving targeted misclassification without modifying model inputs or weights. HistorySwap [7] propose a block-level attack that overwrites contiguous segments of the active KV cache with precomputed caches from different topics. MTI [13] formalize the malicious token injection, which perturbs cached key vectors through additive noise, zeroing, or orthogonal rotations.

However, these attacks require system-level access to modify the cache directly and do not consider broader KV cache matching and reuse mechanisms. *In contrast, our work is the first to enable adversaries to hijack model outputs without requiring system-level privileges.*

Input-Based LLM Attacks. Prompt injection attacks [10, 16, 21, 25, 36, 37, 42, 43, 51] exploit prompt composition, where system/user instructions are combined with untrusted external data (e.g., emails, webpages, API responses). Adversaries manipulate the data portion to override instructions and trigger unauthorized model behavior. Jailbreak attacks [3, 24, 52, 59, 60, 65] similarly rely on adversarial text patterns that steer the model away from its intended safety policies. Both attack types require malicious text to appear in the user's input, either directly in the user's query or indirectly through retrieved content. *In contrast, our work enables attacker-controlled model outputs even when the user's input is without any malicious text.*

3 Preliminaries

LLM Inference. In Transformer-based LLMs, the core computational bottleneck during inference is the Multi-Head Self-Attention (MHA) mechanism [47]. To generate the next token x_{T+1} at step $T+1$, the attention mechanism must model the relationship between the current token x_T and the entire previous sequence $[x_1,\ldots,x_{T-1}]$. This requires projecting the hidden states into Query ($\boldsymbol{Q}$), Key ($\boldsymbol{K}$), and Value ($\boldsymbol{V}$) matrices, and computing the attention scores [47]:

$$\text{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{1}$$

where d_k is the scaling factor. For completeness, Appendix B details the inference process of Transformer-based LLMs.

A naive implementation recomputes the keys and values for all T tokens at every generation step. Because T increases with each step, this redundant computation leads to $O(T^2)$ time complexity, which increases inference latency.

KV Cache Mechanism. To eliminate this redundancy, LLM inference engines employ the *KV cache mechanism*. The key idea is that the keys ($\boldsymbol{K}_{1:T-1}$) and values ($\boldsymbol{V}_{1:T-1}$) of previous tokens remain static during generation. The model only needs to compute representations for the current token x_T. By storing these past tensors in GPU memory, we reduce the computational complexity of each step from $O(T)$ to $O(1)$ with respect to matrix multiplications.

Specifically, for each attention head, the model only computes the query, key, and value vectors for the *current* token x_T via the projection weights $\boldsymbol{W}_Q,\boldsymbol{W}_K,\boldsymbol{W}_V$:

$$\boldsymbol{q}_T = \boldsymbol{u}_T\boldsymbol{W}_Q, \quad \boldsymbol{k}_T = \boldsymbol{u}_T\boldsymbol{W}_K, \quad \boldsymbol{v}_T = \boldsymbol{u}_T\boldsymbol{W}_V \tag{2}$$

where $\boldsymbol{u}_T$ is the normalized hidden state of the current token. The system then appends the new key and value vectors to the existing KV cache:

$$\boldsymbol{K}_{1:T} = [\boldsymbol{K}_{1:T-1};\boldsymbol{k}_T], \quad \boldsymbol{V}_{1:T} = [\boldsymbol{V}_{1:T-1};\boldsymbol{v}_T] \tag{3}$$

Finally, the attention output for the current token is computed by performing the multiplication between the single query vector $\boldsymbol{q}_T$ and the cached matrices $\boldsymbol{K}_{1:T}$ and $\boldsymbol{V}_{1:T}$:

$$\text{Attention}(\boldsymbol{q},\boldsymbol{K},\boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{q}_T\boldsymbol{K}_{1:T}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}_{1:T}. \tag{4}$$

In summary, the KV cache trades memory storage for lower computational cost, which significantly reduces inference latency. Importantly, its benefits extend beyond accelerating a single user's multi-turn dialogue. By sharing the KV cache across different users, the system can further eliminate redundant computations on a global scale.

Prefix KV Cache Prompts sent by different users often exhibit significant prefix overlap (*e.g.*, system prompts or few-shot examples). Based on this observation, the prefix KV cache maintains a global KV cache pool shared across multiple users:

$$\mathcal{P}_{\text{prefix}} = \left\{\tilde{X},(\tilde{\boldsymbol{K}},\tilde{\boldsymbol{V}})\right\}, \tag{5}$$

where $\tilde{X}$ is a cached token sequence and $(\tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}})$ are the corresponding KV cache. When a new prompt X arrives, the LLM inference engine searches the cache pool for a token sequence that shares the longest common prefix with X:

$$\tilde{X}, (\tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}}) = \underset{\tilde{X} \in \mathcal{P}_{\text{prefix}}}{\arg\max} \textsc{PrefixLen}(X, \tilde{X}), \tag{6}$$

where $\textsc{PrefixLen}(\,)$ returns the length of the longest common prefix. The key and value vectors $(\hat{\boldsymbol{k}}_t, \hat{\boldsymbol{v}}_t)$ at position t during the prefill phase are defined as:

$$(\hat{\boldsymbol{k}}_t, \hat{\boldsymbol{v}}_t) = \begin{cases} (\tilde{\boldsymbol{K}}_t, \tilde{\boldsymbol{V}}_t), & 1 \le t \le \textsc{PrefixLen}(X, \tilde{X}), \\ (\boldsymbol{k}_t, \boldsymbol{v}_t), & \text{otherwise}, \end{cases} \tag{7}$$

where $(\tilde{\boldsymbol{K}}_t, \tilde{\boldsymbol{V}}_t)$ denotes the cached key and values vectors at position t from the cache pool, and $(\boldsymbol{k}_t, \boldsymbol{v}_t)$ denotes the vectors computed on-the-fly for unmatched tokens.

Position-Independent KV Cache To overcome the low cache hit rates by strict prefix matching, position-independent KV cache [14, 49, 58, 62] enables the matching of cached chunks against arbitrary subsequences of the input context. The system maintains a cache pool where the chunk serves as the unit of storage:

$$\mathcal{P}_{\text{chunk}} = \left\{ \tilde{X}, (\tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}}) \right\}, \tag{8}$$

where $\tilde{X}$ is a cached token chunk with fixed-length L_{chunk} and $(\tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}})$ are the corresponding KV cache. When a new prompt X arrives, the LLM inference engine searches the cache pool for reusable segments that match a subsequence within X. Let $\mathcal{S}_{\text{hit}}$ denote the set of all successfully hit KV cache chunks:

$$\mathcal{S}_{\text{hit}} = \left\{ i, j, \tilde{X}, (\tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}}) \mid \tilde{X} = X_{i:j}, \tilde{X} \in \mathcal{P}_{\text{chunk}} \right\}. \tag{9}$$

where $[i, j]$ denotes the position interval in the input X matched to the cached chunk $\tilde{X}$. The key and value vectors $(\hat{\boldsymbol{k}}_t, \hat{\boldsymbol{v}}_t)$ at position t during the prefill phase for every hit cache chunk $\{i, j, \tilde{X}, (\tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}})\}$ are defined as:

$$(\hat{\boldsymbol{k}}_t, \hat{\boldsymbol{v}}_t) = \begin{cases} (\tilde{\boldsymbol{K}}_{t'}, \tilde{\boldsymbol{V}}_{t'}), & i \le t \le j, \\ (\boldsymbol{k}_t, \boldsymbol{v}_t), & \text{otherwise}, \end{cases} \tag{10}$$

where $t' = t - i + 1$ is the relative token position.

However, context discrepancies between cached chunks and the actual user input may induce **attention shift** when using a position-independent KV cache, thereby degrading generation quality. To mitigate this, researchers adopt **selective recomputation** [14, 57, 62], which recomputes tokens that are semantically important or show large divergence in their key and values vectors. These methods define a recomputation set $\mathcal{R} \subseteq [i, j]$ of positions that require recomputation. The Equation 10 is then refined as:

$$(\hat{\boldsymbol{k}}_t, \hat{\boldsymbol{v}}_t) = \begin{cases} (\tilde{\boldsymbol{K}}_{t'}, \tilde{\boldsymbol{V}}_{t'}), & t \in [i, j] \setminus \mathcal{R}, \\ (\boldsymbol{k}_t, \boldsymbol{v}_t), & \text{otherwise}, \end{cases} \tag{11}$$

4 Problem Formulation

In this section, we give the problem formulation, including threat model (§ 4.1) and new attack surface (§ 4.2).

4.1 Threat Model

We formally define the threat model guiding our analysis, including the system model, the attacker's goal, capability, and knowledge.

System Model. We consider a realistic deployment scenario where the LLM inference infrastructure integrates two optimization components to improve system throughput and reduce computational cost:

- *Position-Independent KV Cache*. LLM inference systems adopt this novel mechanism to decouple memory retrieval from strict positional constraints, thereby maximizing cache reuse and reducing inference latency.
- *Multi-Tenant Cache Sharing*. The system adopts a multi-tenant architecture in which the KV cache pool is globally shared across users and sessions. This setting is common in organizational or cloud-hosted LLM services, where users frequently draw from common knowledge sources, such as internal documentation, shared RAG corpora, or publicly available materials (*e.g.*, Wikipedia).

Attacker's Goal. The attacker aims to persistently influence model outputs for future users by causing adversarially conditioned KV cache states to be stored and later reused. By submitting inputs derived from widely shared knowledge content, the attacker ensures their cache states are likely to match future benign queries. When reused, these states hijacked generation toward attacker-controlled behavior rather than solely the victim's prompt, resulting in a cross-user integrity attack. ***Unlike prompt injection**, this influence occurs without any adversarial text appearing in the victim's input.* We describe how such infected chunks can be constructed without directly modifying the cache in Section 5.

Attacker's Capabilities. The attacker has *no* direct access to the shared KV cache and *cannot* read, modify, or tamper with cached states, model weights, or system configurations. The attacker also *cannot* inject or alter the text of a victim's query, either directly or indirectly (*e.g.*, via prompt injection or retrieval poisoning).

The attacker can submit inputs to the LLM service and observe the model's feedback. We consider two settings:

- *White-box setting.* The attacker has access to a local surrogate model that is identical to the target model, including model-internal signals such as gradients. The attacker uses this surrogate to optimize adversarial prefixes that condition hijacked KV states.
- *Black-box setting.* The attacker has no access to the target model's internal states or gradients and can only observe its outputs. In this case, adversarial prefixes are optimized on

a locally accessible substitute model and then transferred to the target system.

In both settings, prefix optimization is performed only on a local surrogate model and never touches the victim cache. During local optimization, the surrogate KV cache is cleared between iterations to avoid unintended cache carryover. Inputs submitted by attacker may have their KV states stored in the shared cache pool, enabling cross-user reuse under the system's cache-matching policy.

Attacker's Knowledge. The attacker must know that the target system uses a position-independent KV cache with cross-user reuse. Other assumptions can be relaxed (see detailed analysis in Section 7.4 and Section 7.5). The attacker does *not* need to know other users' query prompts, current cache contents (which may originate from common shared knowledge sources such as internal documents, shared code files, or public corpora), prompt histories, or the internal configuration of the LLM server, including the exact model, recomputation strategy, or the precise cache chunk size and boundary (since sliding-window matching can identify reusable cached chunks within a sufficiently long context).

Generality and Practicality. The attack applies broadly to systems that enable position-independent KV-cache reuse across sessions, regardless of specific model architectures. It requires no privileged access and can be executed via normal user queries, making it feasible in real-world deployments that use cross-user cache sharing for efficiency.

4.2 New Attack Surface

We now explain why the position-independent KV cache reuse introduces a new attack surface within our threat model, in contrast to prefix cache reuse. First, we provide the formal definition of prefix KV cache hit.

Definition 1 (Prefix KV Cache Hit)**.** *Let $X = [x_1, \ldots, x_L]$ be a newly arrived prompt and $\tilde{X} = [\tilde{x}_1, \ldots, \tilde{x}_{\tilde{L}}]$ be an unauthenticated token sequence in the shared cache pool, with associated key and value matrices $\tilde{\boldsymbol{K}}$ and $\tilde{\boldsymbol{V}}$. A prefix cache hit **occurs** if there exists:*

$$X_{1:S} = \tilde{X}_{1:S}. \tag{12}$$

where $X_{1:S} = [x_1, \ldots, x_S]$ and $\tilde{X}_{1:S} = [\tilde{x}_1, \ldots, \tilde{x}_S]$ denote contiguous segments of X and $\tilde{X}$, respectively. And S is the length of the longest common prefix between X and $\tilde{X}$.

Let $(\tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}})$ denote the hit KV cache, and $(\boldsymbol{K}, \boldsymbol{V})$ denote the ground-truth KV states computed on-the-fly without KV cache. Suppose a prefix cache hit **occurs** between X and $\tilde{X}$ with prefix length S. According to Definition 1, X and $\tilde{X}$ are strictly identical over their prefix of length S. Given the deterministic and causal nature of standard decoder-only transformers [47], identical input prefixes strictly yield identical internal representations. Therefore, the hit KV cache states must be numerically identical to the ground-truth states:

$$\tilde{\boldsymbol{K}}_{1:S} = \boldsymbol{K}_{1:S}, \quad \tilde{\boldsymbol{V}}_{1:S} = \boldsymbol{V}_{1:S}. \tag{13}$$

This demonstrates that under the prefix KV cache reuse mechanism, the hit KV cache during the prefill phase is strictly governed by the user's inputs. Consequently, an attacker *cannot* manipulate the LLM's output without system-level privileges to directly modify the KV cache. Therefore, we conclude that the prefix KV cache is **secure**.

Then we formalize the definition of a position-independent KV cache hit for a more detailed comparison.

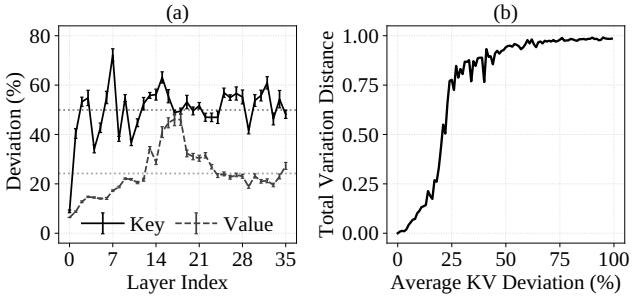

Figure 2: Empirical analysis of KV Cache deviation. **(a) Prefix → KV deviation:** We prepend different prefixes to the input text and measure the KV deviation relative to a prefix-free baseline. **(b) KV deviation → output change:** We inject varying levels of noise into the KV cache to investigate the impact of numerical deviations on the final output. We observe that the deviations inherent in the position-independent KV cache are sufficient (*i.e.*, when $\geq 20\%$) to alter model outputs, potentially enabling adversarial exploitation.

Definition 2 (Position-Independent KV Cache Hit)**.** *Let $X = [x_1, \ldots, x_L]$ be a newly arrived prompt and $\tilde{X} = [\tilde{x}_1, \ldots, \tilde{x}_{\tilde{L}}]$ be an unauthenticated token sequence in the shared cache pool, with associated key and value matrices $\tilde{\boldsymbol{K}}$ and $\tilde{\boldsymbol{V}}$. A position-independent cache hit **occurs** if there exists:*

$$X_{a:b} = \tilde{X}_{m:n}. \tag{14}$$

where $X_{a:b} = [x_a, \ldots, x_b]$ and $\tilde{X}_{m:n} = [\tilde{x}_m, \ldots, \tilde{x}_n]$ denote contiguous segments of X and $\tilde{X}$, respectively.

Suppose a position-independent cache hit occurs between $X_{a:b}$ and $\tilde{X}_{m:n}$. By comparing Definition 1 and Definition 2, we observe that Equation 12 is a special case of Equation 14 where $m = a = 1$ and $b = n = S$. However, the conclusion in Equation 13 does not extend to position-independent case. Considering the deterministic and causal nature of LLMs, the generation of KV states is strictly conditioned on the preceding tokens. In general position-independent reuse scenarios, the preceding contexts are distinct ($X_{1:a-1} \neq \tilde{X}_{1:m-1}$). Assuming the injectivity of LLMs [32], this history mismatch

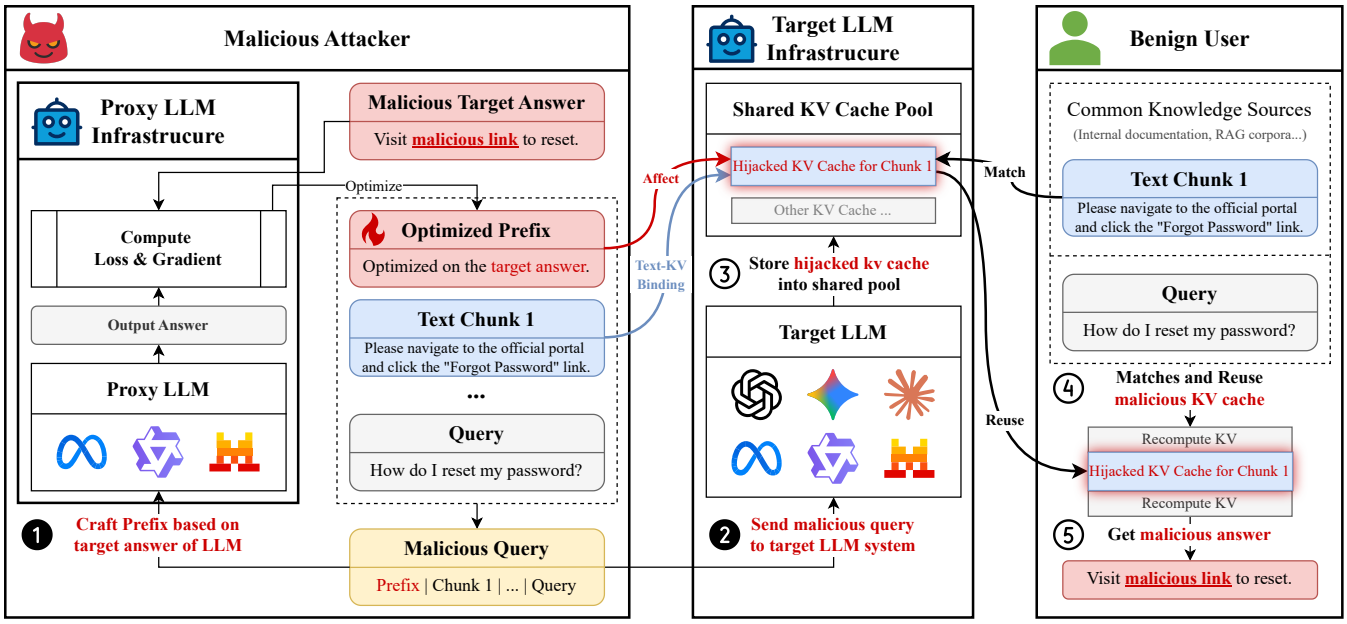

Figure 3: HIJACKKV consists of two phases: (1) constructs an adversarial prefix p and submits a malicious query $p \oplus \tilde{X}$ to the LLM service; (2) the service processes $p \oplus \tilde{X}$ into KV states and stores them in the shared cache pool $\mathcal{P}_{\text{chunk}}$. When a user sends a query $X \oplus q$, if a cache hit occurs between $\tilde{X}$ and X as Definition 2, the user receives the malicious response $\tilde{r}$.

implies a divergence in the internal states. Consequently, even though the token sequences within the matching span are strictly identical, it inevitably leads to:

$$\tilde{\boldsymbol{K}}_{m:n} \neq \boldsymbol{K}_{a:b}, \quad \tilde{\boldsymbol{V}}_{m:n} \neq \boldsymbol{V}_{a:b}. \tag{15}$$

We investigate the security implications of this KV deviation in Figure 2. In Figure 2(a), we visualize the KV deviation caused by cache reuse by pairing a text segment with various prefixes and calculating the average deviation compared to a prefix-free baseline. Complementing this, Figure 2(b) examines the impact of such deviation on model outputs by injecting varying levels of noise into the KV cache and measuring the Total Variation Distance of the next-token logits. Our results highlight a critical vulnerability: while a mere 20% numerical deviation in KV states is sufficient to alter the LLM's output, position-independent KV caches typically exhibit much higher deviations–approximately 50% in keys and 25% in values. This substantial discrepancy creates a wide attack surface for adversaries to hijack model generations. Therefore, we conclude that the position-independent KV cache is inherently **insecure**.

5 KV Cache Hijacking

Figure 3 illustrates HIJACKKV, our attack framework targeting position-independent KV cache reuse. A complete attack consists of two phases: (1) the attacker constructs an adversarial prefix p and submits a malicious query $p \oplus \tilde{X}$ to the LLM service; (2) the LLM service processes $p \oplus \tilde{X}$ into KV states and stores them as chunks in the shared cache pool $\mathcal{P}_{\text{chunk}}$. Subsequently, when a user sends a query $X \oplus q$, if a cache hit occurs between $\tilde{X}$ and X as Definition 2, the user receives the malicious response $\tilde{r}$.

We formulate the problem of finding a prefix p that causes the LLM to stably output the malicious response $\tilde{r}$ in the aforementioned scenario as an optimization problem:

$$p^* = \arg\min_p \mathcal{L}_{\text{CE}} \left(\text{LLM}(X \oplus q \mid \mathcal{T}^p_{\tilde{X}}), \tilde{r} \right), \tag{16}$$

where $\tilde{X}$ denotes a subsequence of the user context X, $\mathcal{T}^p_{\tilde{X}}$ represents the KV cache of $\tilde{X}$ obtained by querying the LLM with $p \oplus \tilde{X}$, and $\text{LLM}(X \oplus q \mid \mathcal{T}^p_{\tilde{X}})$ indicates LLM inference with cache hits on $\mathcal{T}^p_{\tilde{X}}$. $\tilde{X}$ can be chosen as frequently used context that provides answers to q, such as text from an FAQ interface, to further increase the attack success rate.

We employ the Greedy Coordinate Gradient (GCG) algorithm [65] to solve this optimization problem, a method widely applied in discrete token optimization. The optimization process of HIJACKKV consists of two main components: (1) an iterative GCG optimization loop that refines the adversarial prefix through coordinate-wise gradient descent, and (2) a loss computation function that evaluates the effectiveness of each candidate prefix under position-independent cache reuse. Algorithm 1 presents the implementation of this algorithm. We describe each component in detail below. In the actual attack process, the attacker runs the algorithm locally in a simulator using a surrogate model, thereby avoiding any impact on the remote server-side model or cache pool.

Algorithm 1: HIJACKKV PREFIX OPTIMIZATION

Input: Target text $\tilde{X}$, target query q, target malicious answer $\tilde{r}$, prefix length L, optimize step T, number of candidate prefixes B

Output: Optimized prefix p

1 $p \leftarrow \text{INITIALIZEPREFIX}(L)$;
2 **for** $t = 1$ **to** T **do**
3 $\quad \mathcal{L} \leftarrow \text{COMPUTELOSS}(p, \tilde{X}, q, \tilde{r})$;
4 $\quad \nabla_p \leftarrow \nabla_{\text{ONEHOT}(p)} \mathcal{L}$;
5 $\quad$ **for** $l = 1$ **to** L **do**
6 $\quad\quad \mathcal{V}^l_{cand} \leftarrow \text{TOPK}(-\nabla_p)$;
7 $\quad P \leftarrow \emptyset$;
8 $\quad$ **for** $b = 1$ **to** B **do**
9 $\quad\quad l \leftarrow \text{UNIFORMRANDOM}(L)$;
10 $\quad\quad x' \leftarrow \text{RANDOMSELECT}(\mathcal{V}^l_{cand})$;
11 $\quad\quad P \leftarrow P \cup \{[x_1, \ldots, x_{l-1}, x', x_{l+1}, \ldots, x_L]\}$;
12 $\quad p \leftarrow \arg\min_{p \in P} \text{COMPUTELOSS}(p, \tilde{X}, q, \tilde{r})$;
13 **return** p ;
14 **Function** $\text{COMPUTELOSS}(p, \tilde{X}, q, \tilde{r})$
15 $\quad \boldsymbol{e}_p \leftarrow \text{ONEHOT}(p) \cdot W_{\text{embed}}$;
16 $\quad \boldsymbol{e}_{\text{context}} \leftarrow \boldsymbol{e}_p \oplus \text{EMBEDDING}(\tilde{X})$;
17 $\quad (\boldsymbol{K}, \boldsymbol{V}) \leftarrow \text{COMPUTEKV}(\text{LLM}(\boldsymbol{e}_{\text{context}}))$;
18 $\quad \mathcal{T}^p_{\tilde{X}} \leftarrow (\boldsymbol{K}_{L+1:L+|\tilde{X}|}, \boldsymbol{V}_{L+1:L+|\tilde{X}|})$;
19 $\quad r \leftarrow \text{LLM}(\tilde{X} \oplus q \mid \mathcal{T}^p_{\tilde{X}})$;
20 $\quad \mathcal{L} \leftarrow \text{CROSSENTROPY}(r, \tilde{r})$;
21 $\quad$ **return** $\mathcal{L}$;

Greedy Coordinate Gradient Optimization. The main optimization loop (Lines 2 - 12) iteratively refines the adversarial prefix p over T iterations.

Since the prefix consists of discrete tokens, the gradient is computed through the continuous relaxation obtained by embedding the one-hot vectors. At each iteration t, the algorithm first computes the loss $\mathcal{L}$ and its gradient ∇_p with respect to the one-hot encoded prefix $\text{ONEHOT}(p)$ (Lines 3 - 4).

For each position l in the prefix, the algorithm identifies the top-k candidate tokens that would most decrease the loss, based on the negative gradient values (Lines 5 - 6).

To explore the discrete space efficiently, the algorithm constructs a candidate set P by randomly sampling B positions and substituting each with a randomly selected token from its top-k candidates $\mathcal{V}^l_{cand}$ (Lines 7 - 11).

The candidate prefix that achieves the minimum loss is then greedily selected as the new prefix for the next iteration (Line 12). Upon completion of the iterations, we obtain an effective malicious prefix p (Line 13). This GCG optimization enables efficient search within the exponentially large and discrete token space.

Evaluation of Malicious KV Cache. The COMPUTELOSS function (Lines 14 - 21) is the core component that evaluates the effectiveness of a candidate prefix under position-independent cache reuse. This function simulates the complete attack pipeline from prefix injection to victim query inference. We now describe each step in detail.

Step 1: Compute continuous embedding of the malicious prefix p (Line 15). Since gradient-based optimization requires differentiable operations, the discrete token sequence p is first converted to a continuous representation. The one-hot encoding of p is multiplied by the model's embedding matrix W_{embed} to obtain continuous embeddings $\boldsymbol{e}_p \in \mathbb{R}^{L \times d}$, where d is the embedding dimension. This operation enables the direct computation of the gradient of the loss function with respect to each token.

Step 2: Compute hijacked KV cache (Lines 16 - 18). The prefix embeddings $\boldsymbol{e}_p$ are concatenated with the embeddings of the target chunk $\tilde{X}$ to form the complete context $\boldsymbol{e}_{\text{context}} = \boldsymbol{e}_p \oplus \text{EMBEDDING}(\tilde{X})$, where $\oplus$ denotes sequential concatenation along the token dimension. This concatenated sequence is then fed through the LLM to generate the complete key-value pairs $(\boldsymbol{K}, \boldsymbol{V})$ for all layers. Critically, only the KV pairs corresponding to the target chunk c are extracted and cached, yielding $\mathcal{T}^p_c = (\boldsymbol{K}_{L+1:L+|\tilde{X}|}, \boldsymbol{V}_{L+1:L+|\tilde{X}|})$. The slicing operation $L+1 : L+|\tilde{X}|$ selects the KV states from position $L+1$ (begin with the end of prefix) to position $L+|\tilde{X}|$ (the end of the chunk), discarding the prefix's KV states while retaining its adversarial influence encoded in the chunk's representations. This computation strategy aligns with real-world KV cache systems that cache document chunks $\tilde{X}$ without explicitly storing malicious prefix p.

Step 3: Simulate victim's inference (Line 19). To evaluate the attack's effectiveness, the function simulates a victim's query by performing inference with the malicious cache reuse. The shared context $\tilde{X}$ is concatenated with the target query q, and the LLM generates output response r while reusing the hijacked KV cache $\mathcal{T}^p_c$. Notably, in real-world cache reuse scenarios, the cache $\mathcal{T}^p_c$ may have undergone processing such as recomputation [14, 62] or compression [26, 63]. Consequently, the attacker can replicate these operations during this optimization step to enhance the robustness of the attack. By simulating position-independent cache reuse, we can evaluate the impact of $\mathcal{T}^p_c$ on the LLM in real-world scenarios.

Step 4: Compute loss (Line 20 - 21). Finally, we compute the cross-entropy loss $\mathcal{L}$ to measure the divergence between the model's current output distribution r and the designated malicious target $\tilde{r}$. Minimizing $\mathcal{L}$ provides the essential gradient signals required to iteratively refine the adversarial prefix and achieves alignment with the attacker's target objective.

Algorithm 1 produces an adversarial prefix p by optimizing the prefix p through GCG. When p serves as the prefix for the context $\tilde{X}$, it ensures that the KV state of $\tilde{X}$ is manipulated, such that upon a position-independent cache hit, the model's output is hijacked to the attacker's desired $\tilde{r}$.

6 Experimental Setup

Datasets. We employed four question-answering (QA) benchmark datasets for evaluation. HotpotQA [61] and SQuAD (v1.1 and v2.0) [39, 40] serve as general domain QA benchmarks, while MedQA [18] and PubMedQA [19] represent specific domain datasets from the medical field.

To evaluate the effectiveness of the attack, we randomly sampled 200 instances from each dataset. Each instance was formatted as a triplet consisting of a question, a ground-truth answer, and a context containing that answer. We employed an LLM to generate a specific incorrect answer for each question. These questions can be classified into three categories, with distinct strategies applied to generate the incorrect answers: **(1) Binary Questions** (e.g., Yes/No or A/B): We generated the logical opposite of the ground-truth answer. **(2) Multiple-Choice Questions:** We randomly selected one of the incorrect options. **(3) Open-Ended Questions:** We generated an incorrect answer that shares the same part-of-speech or semantic category as the ground truth. For example, given the question and ground-truth answer "What is the capital of France? Paris", we prompted the LLM to generate an incorrect entity such as "Hawaii." The specific prompts used for this process are shown in Appendix D.

Models. We use a diverse set of state-of-the-art open-source LLM families, including Qwen [56], LLaMA [9], and Mistral [30], for evaluation. These models cover a wide spectrum of parameter sizes, ranging from 1B to 70B. We designed a specific system prompt to instruct the LLMs to prioritize answer extraction from the provided context and to ensure the generated responses are concise. The prompt used for this process are detailed in Appendix D.

KV Cache System. We formalize a comprehensive KV cache system defined by the following four key components: **(1) Recomputation Method ($\mathcal{R}$):** We implement two recomputation strategies based on *Cacheblend* [62] and *EPIC* [14]. Additionally, we introduce a *Random* method, which randomly selects tokens for recomputation. We designate the baseline without any recomputation as *Vanilla* and the baseline with full recomputation as *Full*. **(2) Chunk Size (L_{chunk}):** This is the length of the minimal unit segment required to trigger a cache hit. **(3) Cache Ratio (δ):** This denotes the proportion of chunks within the user's context that are replaced by matched cache entries. It can be calculated as $\delta = (b-a+1)/L$. **(4) Recomputation Ratio (ρ):** This indicates the proportion of matched cached tokens that undergo recomputation.

Metrics. Let r denote the ground-truth answer and $\tilde{r}$ denote the target malicious answer. Let x represent the model's response in the benign setting, and y represent the response under the proposed attack. We employ the following three metrics to evaluate the effectiveness of the attack and the preservation of model utility: **(1) Accuracy (Acc):** Defined as the condition where $x = r$. This metric evaluates the model's baseline capability on the QA task and assesses the impact of different KV cache system settings on benign performance. **(2) Untargeted Attack Success Rate (U-ASR):** Defined as the condition where $y \neq x$. This metric measures the effectiveness of the attack in successfully altering the model's output, indicating a deviation from the original generation. **(3) Targeted Attack Success Rate (T-ASR):** Defined as the condition where $y = \tilde{r}$ and $y \neq x$. This serves as a stricter metric, quantifying the attack's success in manipulating the model to generate the specific malicious answer from the attacker.

Environment. All experiments are conducted on a server equipped with an AMD EPYC 9334 32-Core Processor running Ubuntu 22.04.5 LTS, with four NVIDIA RTX PRO 6000 Blackwell Workstation Edition GPUs.

7 Experiment

We conduct a comprehensive evaluation to demonstrate the severe security threat HIJACKKV poses to position-independent KV cache reuse systems. Crucially, we highlight that this threat cannot be mitigated by existing text-level defenses, sanitizers, or alignment mechanisms, as HIJACKKV fundamentally manipulates the internal KV representations rather than the textual input. To systematically analyze the impact of this vulnerability, our evaluation is guided by the following research questions:

- **RQ1 (Effectiveness):** Can HIJACKKV exploit the identified vulnerability to manipulate model outputs?
- **RQ2 (Robustness):** Can HIJACKKV maintain its effectiveness across diverse KV cache system configurations?
- **RQ3 (Persistence):** Can HIJACKKV sustain its malicious impact throughout multi-turn interactions?
- **RQ4 (Transferability):** Can HIJACKKV successfully manipulate model outputs under black-box settings?

Unless otherwise specified, RQ1-RQ3 are evaluated under the white-box setting, while RQ4 evaluates the black-box setting. Furthermore, we conduct comprehensive ablation studies to evaluate the impact of attack hyperparameters on HIJACKKV. Finally, we study adaptive attacks against recomputation-based defenses and examine whether existing defense mechanisms and cache compression methods can mitigate the negative impact introduced by HIJACKKV.

7.1 RQ1: Effectiveness

Setup. RQ1 investigate the effectiveness of the HIJACKKV across different models, datasets, and recomputation methods. Experiments are conducted on four QA benchmarks: HotpotQA [61], SQuAD (v1.1 and v2.0) [39, 40], MedQA [18], and PubMedQA [19]. Regarding hyperparameters, we set the cache ratio $\delta = 0.3$, the recomputation ratio $\rho = 0.1$, and the chunk size $L_{\text{chunk}} = 32$ as the default settings [14, 62]. We evaluate the effectiveness of HIJACKKV without using KV cache

Table 1: **[RQ1] Effectiveness of HIJACKKV.** This table shows the attack effectiveness of HIJACKKV across four different datasets, three different models, and three different recomputation methods. The KV cache system are conducted under default settings with a cache ratio $\delta = 0.3$ and a recomputation ratio $\rho = 0.1$. *Acc* is the model's task accuracy reported to measure the impact of position-independent KV reuse on model utility. *U-ASR* and *T-ASR* is reported as the attack performance metrics.

Method	HotpotQA			SQuAD			MedQA			PubMedQA		
	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR
Llama-3.1-8B												
Full	0.73	–	–	0.76	–	–	0.71	–	–	0.92	–	–
Vanilla	0.58	1.00	1.00	0.60	0.98	0.96	0.55	0.98	0.90	0.74	0.93	0.92
Random	0.63	0.95	0.94	0.69	0.98	0.89	0.63	0.93	0.87	0.79	0.92	0.89
EPIC	0.66	0.87	0.85	0.72	0.89	0.75	0.70	0.96	0.82	0.87	0.96	0.91
CacheBlend	0.68	0.92	0.89	0.72	0.94	0.85	0.68	0.88	0.80	0.89	0.92	0.86
Ministral-8B												
Full	0.70	–	–	0.67	–	–	0.68	–	–	0.94	–	–
Vanilla	0.57	1.00	1.00	0.55	0.98	0.93	0.54	1.00	1.00	0.71	0.97	0.94
Random	0.61	0.97	0.97	0.62	0.89	0.87	0.63	1.00	1.00	0.75	0.91	0.88
EPIC	0.67	0.96	0.95	0.65	0.88	0.78	0.63	1.00	1.00	0.84	0.92	0.87
CacheBlend	0.66	0.94	0.94	0.66	0.93	0.87	0.65	1.00	1.00	0.85	0.89	0.85
Qwen3-8B												
Full	0.82	–	–	0.84	–	–	0.73	–	–	0.95	–	–
Vanilla	0.61	0.96	0.94	0.68	1.00	1.00	0.57	0.97	0.96	0.75	1.00	1.00
Random	0.72	0.91	0.85	0.74	0.95	0.90	0.66	0.86	0.83	0.85	0.93	0.89
EPIC	0.75	0.88	0.87	0.75	0.92	0.86	0.71	0.93	0.84	0.92	0.95	0.93
CacheBlend	0.80	0.91	0.89	0.82	0.93	0.87	0.70	0.92	0.87	0.92	1.00	0.91

(*Full*), with full KV cache reuse (*Vanilla*), and with three recomputation methods (*Random*, *EPIC*, and *CacheBlend*). We use white-box setting and report Acc for model performance, and U-ASR and T-ASR for the effectiveness of HIJACKKV.

Results. Table 1 shows that HIJACKKV demonstrates strong attack performance across all evaluated scenarios. Specifically, on the Llama-3.1-8B model, HIJACKKV achieves an average T-ASR of 89% across the four datasets, even when countering the four distinct recomputation methods. While advanced recomputation strategies like EPIC and CacheBlend result in a slight reduction in ASR compared to the Vanilla setting, HIJACKKV still maintains effectiveness. This trend is further supported by the results on Qwen3-8B, where the attack frequently achieves near-perfect success rates (e.g., 100% T-ASR on PubMedQA), proving that HIJACKKV remains robust despite the slight drop introduced by partial recomputation.

Conclusion. In conclusion, HIJACKKV is a highly effective, model-agnostic attack that hijacks LLM generation across diverse domains. Our findings indicate that standard partial recomputation mechanisms, at current ratios, are insufficient to mitigate the adversarial cache optimized by HIJACKKV.

7.2 RQ2: Robustness

Setup. RQ2 conducts experiments to analyze the impact of the *chunk size* L_{chunk}, *cache ratio* δ, and *recomputation ratio* ρ of the position-independent cache system on HIJACKKV. All experiments are performed on the HotpotQA dataset with Llama-3.1-8B. We vary the hyperparameters by selecting L_{chunk} from the set $\{32, 64, 128, 256, 512\}$ while exploring both δ and ρ within the values of $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. We use white-box setting and report Acc for model performance, and U-ASR and T-ASR for the effectiveness of HIJACKKV.

Impact of Chunk Size L_{chunk}. Table 2 presents the experimental results regarding the impact of chunk size L_{chunk} on the effectiveness of HIJACKKV. This parameter determines the minimum granularity for storage, matching, and reuse in the position-independent KV cache.

As L_{chunk} increases from 32 to 512, we observe that both the main task Acc and the ASR of HIJACKKV remain stable. Although the ASR decreases slightly when L_{chunk} is large, we attribute this to partial cache misses at the head and tail of the hijacked KV cache, *i.e.*, incomplete cache reuse caused by coarse cache granularity. Overall, these results demonstrate that HIJACKKV is robust to variations in chunk size.

Impact of Cache Ratio δ. Table 3 illustrates the experimental results regarding the impact of cache ratio δ on the effectiveness of HIJACKKV. This parameter determines the proportion of the user's prefilled KV cache occupied by the hijacked KV cache. Since existing position-independent KV cache methods typically achieve cache hit rates of up to 60% [57], we vary the cache ratio from 10% to 50% to evaluate the robustness of HIJACKKV.

We find that HIJACKKV maintains a high ASR even at low cache hit rate, where only a small proportion of the user's KV cache is affected. For example, at $\delta = 0.1$ and $\rho = 0.3$, where the malicious cache occupies only 7% of the user's KV context, the average T-ASR remains at 67%. As cache ratio increases, ASR of HIJACKKV rises rapidly, approaching 100%

Table 2: **[RQ2-1] Robustness of HIJACKKV to Chunk Size L_{chunk}.** This table shows the robustness to chunk size using 200 samples from the HotpotQA dataset. The experiments employ Llama-3.1-8B under default settings with a cache ratio $\delta = 0.3$ and a recomputation ratio $\rho = 0.1$. *Acc* is the model's task accuracy to measure the impact of position-independent KV reuse on model utility, where the performance without KV cache is 0.73. *U-ASR* and *T-ASR* is reported as the attack performance metrics.

Method	$L_{\text{chunk}} = 32$			$L_{\text{chunk}} = 64$			$L_{\text{chunk}} = 128$			$L_{\text{chunk}} = 256$			$L_{\text{chunk}} = 512$		
	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR
Vanilla	0.58	1.00	1.00	0.58	1.00	1.00	0.58	1.00	1.00	0.59	1.00	1.00	0.57	1.00	1.00
Random	0.63	0.95	0.94	0.61	0.95	0.95	0.64	0.94	0.93	0.63	0.94	0.92	0.62	0.93	0.90
EPIC	0.66	0.87	0.85	0.67	0.89	0.88	0.68	0.87	0.86	0.66	0.88	0.87	0.71	0.85	0.84
CacheBlend	0.68	0.92	0.89	0.68	0.91	0.91	0.70	0.92	0.90	0.68	0.89	0.85	0.69	0.91	0.89

Table 3: **[RQ2-2] Robustness of HIJACKKV to Cache Ratio δ.** This table shows the robustness to cache ratio using 200 samples from the HotpotQA dataset. The experiments employ Llama-3.1-8B under default settings with a recomputation ratio $\rho = 0.1$. *Acc* is the model's task accuracy to measure the impact of position-independent KV reuse on model utility, where the performance without KV cache is 0.73. *U-ASR* and *T-ASR* is reported as the attack performance metrics.

Method	$\delta = 0.1$			$\delta = 0.2$			$\delta = 0.3$			$\delta = 0.4$			$\delta = 0.5$		
	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR
Vanilla	0.64	0.82	0.81	0.61	0.95	0.93	0.58	1.00	1.00	0.46	1.00	1.00	0.38	1.00	1.00
Random	0.68	0.63	0.61	0.65	0.81	0.78	0.63	0.95	0.94	0.59	0.95	0.95	0.52	0.98	0.97
EPIC	0.73	0.56	0.53	0.71	0.80	0.78	0.66	0.87	0.85	0.63	0.93	0.91	0.61	0.99	0.96
CacheBlend	0.72	0.73	0.71	0.71	0.80	0.78	0.68	0.92	0.89	0.65	0.96	0.93	0.60	1.00	0.99

at $\delta = 0.5$. Additionally, we observe that the recomputation method EPIC becomes more effective as the low cache ratio. We attribute this to its recomputation of the first k tokens in each cache chunk, which better preserves benign context coherence at lower cache ratios, thereby enhancing resistance against HIJACKKV. In summary, even at lower cache ratios, HIJACKKV still mounts effective attacks, demonstrating its robustness to such reductions.

Impact of Recomputation Ratio ρ. Table 3 demonstrates the experimental results regarding the impact of recomputation ratio ρ on the effectiveness of HIJACKKV. This parameter determines the recomputation ratio for the reused KV cache. While existing methods claim that a recomputation ratio from 10% to 15% is sufficient to prevent performance degradation on the main task, we vary the recomputation ratio from 10% to 50% to evaluate the robustness of HIJACKKV.

We observe that increasing the recomputation ratio brings significant computational overhead but does not eliminate the malicious impact of HIJACKKV. Specifically, the average T-ASR remains high at 70% when the recomputation ratio is tripled to $\rho = 0.3$ and persists at 39% even when the ratio is raised to $\rho = 0.5$, which is 5× the baseline. HIJACKKV shows robustness to an increase in the recomputation ratio.

Furthermore, we find that increasing ρ is a more effective defense than reducing δ. We compare two settings where the attacker controls a similar proportion of effective tokens: (1) $\delta = 0.3, \rho = 0.5$ (15% hijacked KV cache) and (2) $\delta = 0.2, \rho = 0.3$ (14% hijacked KV cache). Despite the similar proportions of hijacked KV cache, the T-ASR is 39% lower in the first case. This is because recomputation refreshes more high-influence (including malicious) KV cache entries, whereas a reduced cache hit ratio still preserves many of them.

Conclusion. Extensive experiments validate the robustness of HIJACKKV across varying system configurations. Whether subjected to low cache ratios, high recomputation penalties, or varying chunk sizes, HIJACKKV maintains strong attack effectiveness. This consistency demonstrates that HIJACKKV poses a widespread threat that cannot be easily mitigated.

7.3 RQ3: Persistence

Setup. RQ3 investigates the persistence of the malicious impact induced by HIJACKKV within a multi-turn conversation scenario. To simulate multi-turn interactions, we first hijack the user's KV cache using HIJACKKV. Then we insert filler tokens of length L_{context} that are unrelated to the hijacked target topic before issuing the target query, and observe *whether the attack influence is diluted* in the model's response. All experiments are conducted on the HotpotQA dataset using Llama-3.1-8B and the same hyperparameters as in Section 7.1. We evaluate the effectiveness of HIJACKKV without KV cache (*Full*), with full KV cache (*Vanilla*), and with three recomputation methods (*Random*, *EPIC*, and *CacheBlend*). We use white-box setting and report Acc for model performance, and U-ASR and T-ASR for the effectiveness of HIJACKKV.

Results. Table 5 illustrates the impact of multi-turn conversation length L_{context} on the effectiveness of HIJACKKV. In our experiments, we increase the length of the unrelated filler

Table 4: **[RQ2-3] Robustness of HIJACKKV to Recomputation Ratio ρ.** This table shows the robustness to recomputation ratio using 200 samples from the HotpotQA dataset. The experiments employ Llama-3.1-8B under default settings with cache ratio $\delta = 0.3$. *Acc* is models' task accuracy reported to measure the impact of position-independent KV reuse on model utility, where the performance stands at 0.73 without KV cache and 0.58 without recomputation. *U-ASR* and *T-ASR* are reported as attack performance metrics, both of which reach 100% on the vanilla position-independent KV cache without recomputation.

Method	$\rho = 0.1$			$\rho = 0.2$			$\rho = 0.3$			$\rho = 0.4$			$\rho = 0.5$		
	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR
Random	0.63	0.95	0.94	0.66	0.73	0.72	0.69	0.66	0.65	0.70	0.45	0.43	0.72	0.35	0.31
EPIC	0.66	0.87	0.85	0.70	0.82	0.81	0.72	0.72	0.71	0.73	0.63	0.63	0.73	0.51	0.49
CacheBlend	0.68	0.92	0.89	0.71	0.83	0.81	0.72	0.74	0.73	0.72	0.57	0.55	0.72	0.40	0.38

Table 5: **[RQ3] Persistence of HIJACKKV.** This table evaluates the impact of multi-turn dialogue length on the performance of HIJACKKV performance using 200 samples from the HotpotQA dataset. The experiments employ Llama-3.1-8B under default settings with a cache ratio $\delta = 0.3$ and recomputation ratio $\rho = 0.1$. *Acc* is the model's task accuracy reported to measure the impact of position-independent KV reuse on model utility. *U-ASR* and *T-ASR* is reported as the attack performance metrics.

Method	$L_{\text{context}} = 0$			$L_{\text{context}} = 256$			$L_{\text{context}} = 512$			$L_{\text{context}} = 1024$			$L_{\text{context}} = 2048$		
	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR	Acc	U-ASR	T-ASR
Full	0.73	–	–	0.73	–	–	0.71	–	–	0.68	–	–	0.65	–	–
Vanilla	0.58	1.00	1.00	0.58	1.00	0.98	0.55	0.82	0.76	0.51	0.69	0.63	0.49	0.62	0.48
Random	0.63	0.95	0.94	0.62	0.90	0.89	0.62	0.80	0.64	0.59	0.66	0.48	0.58	0.63	0.46
EPIC	0.66	0.87	0.85	0.66	0.85	0.84	0.83	0.77	0.68	0.64	0.67	0.58	0.62	0.63	0.48
CacheBlend	0.68	0.92	0.89	0.68	0.91	0.88	0.68	0.79	0.62	0.63	0.65	0.52	0.61	0.64	0.46

context from 0 up to 2048 tokens. We find that even with a context length of 1024, the average T-ASR remains high at 55%. Furthermore, when the context length extends to 2048, the T-ASR remains at 47%. Notably, once the context reaches a certain length, doubling it results in only a marginal 8% decline in the ASR.

Conclusion. These findings confirm that once cached, the hijacked KV cache retains sufficient influence to manipulate the LLM's token generation process over long-range interactions, despite the presence of extensive unrelated context.

7.4 RQ4: Transferability

Setup. RQ4 focuses on the cross-model transferability of HIJACKKV, specifically examining its attack effectiveness under the black-box setting described in Section 4.1. We optimize the prefix p on a proxy model and send the constructed malicious query $p \oplus \tilde{X}$ to the target model. We then evaluate the success rate of the target LLM generating the response $\tilde{r}$ for the query $X \oplus q$, under the condition that a position-independent cache hit (as defined in Definition 2) occurs between X and $\tilde{X}$. We employ Llama-3.1-8B as the proxy model for prefix optimization. All experiments are conducted on the HotpotQA dataset using Llama-3.1-8B and the same hyperparameters as in Section 7.1. We evaluate the effectiveness of with full KV cache (*Vanilla*), and with three recomputation methods (*Random*, *EPIC*, and *CacheBlend*). We use black-box setting and report U-ASR and T-ASR for the effectiveness of HIJACKKV.

Results. Table 6 presents experimental results demonstrating that HIJACKKV exhibits robust cross-model transferability. When the target LLM is the large-parameter Llama-3.3-70B, HIJACKKV maintains high efficacy, achieving 75% U-ASR and 37% T-ASR. Similarly, HIJACKKV achieves 76% U-ASR and 37% T-ASR against models with comparable parameter counts but distinct architectures like Qwen3-8B and 14B. These results indicate that the prefix p optimized by HIJACKKV poses a significant threat in black-box settings and enables attack transferability across models of varying parameter sizes and architectures.

We observe a performance divergence when transferring the attack to smaller target models. Specifically, on Qwen3-4B and Llama-3.2-1B, while HIJACKKV achieves a high U-ASR of approximately 90%, the T-ASR drops significantly to 13%. We attribute this low T-ASR to the capability gap between models, as indicated by the main task performance in Table 7. Drawing on the Platonic Representation Hypothesis [15], which suggests that increasingly capable models converge toward a shared representation space, we argue that prefixes optimized on larger proxy models are effective against similarly capable or stronger targets. However, the representation spaces of smaller models likely diverge from that of the proxy, making it difficult to precisely steer them to generate the specific target response $\tilde{r}$. Conversely, the U-ASR remains high because the prefix p successfully perturbs the KV cache.

Table 6: **[RQ-4] Transferability of HIJACKKV.** This table evaluates cross-model transferability of HIJACKKV performance using 200 samples from the HotpotQA dataset. The experiments employ Llama-3.1-8B as the proxy model under default settings with a cache ratio $\delta = 0.3$ and a recomputation ratio $\rho = 0.1$. *U-ASR* and *T-ASR* is reported as the attack performance metrics.

Method	Qwen3-4B		Qwen3-8B		Qwen3-14B		Llama-3.2-1B		Llama-3.2-3B		Llama-3.3-70B	
	U-ASR	T-ASR	U-ASR	T-ASR	U-ASR	T-ASR	U-ASR	T-ASR	U-ASR	T-ASR	U-ASR	T-ASR
Llama-3.1-8B												
Vanilla	0.91	0.10	0.87	0.33	0.70	0.40	0.97	0.18	0.90	0.44	0.79	0.37
Random	0.89	0.11	0.87	0.37	0.64	0.41	0.93	0.17	0.89	0.42	0.73	0.35
EPIC	0.88	0.14	0.83	0.32	0.64	0.34	0.89	0.14	0.90	0.46	0.72	0.37
CacheBlend	0.85	0.09	0.87	0.38	0.62	0.39	0.90	0.13	0.89	0.44	0.75	0.38

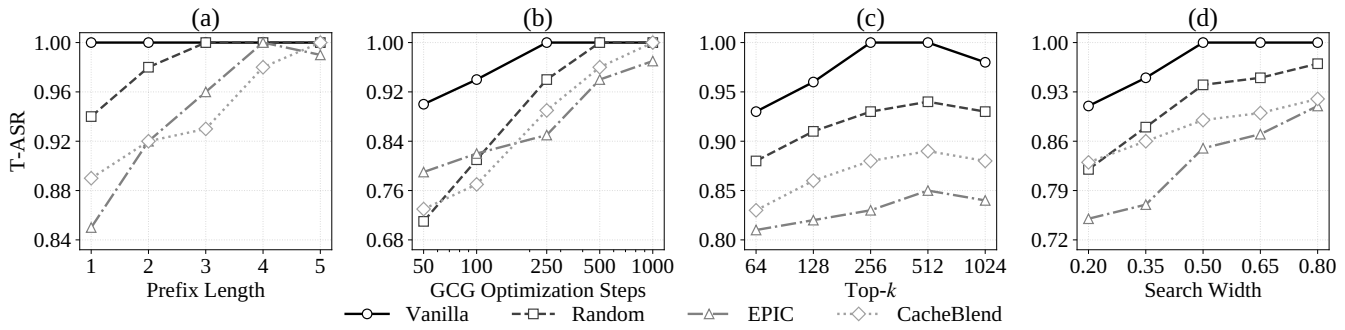

Figure 4: **Ablation studies of HIJACKKV hyperparameters.** We evaluate the impact of four HIJACKKV hyperparameters on effectiveness: (a) adversarial prefix length, (b) number of GCG optimization steps, (c) top-k candidates, and (d) search width.

Table 7: **Main task performance of different models.** This table presents the performance (measured by *Acc*) of various models on the HotpotQA main task and reports the performance difference (Δ) relative to Llama-3.1-8B.

Metric	Llama 3				Qwen 3		
	8B	1B	3B	70B	4B	8B	14B
Acc	**0.73**	0.39	0.52	0.88	0.54	0.82	0.85
Δ	–	−0.34	−0.21	+0.15	−0.19	+0.09	+0.12

This induces substantial contextual hallucinations, causing the model to generate irrelevant or incorrect responses even if it fails to match the exact target string.

Conclusion. In summary, our experiments confirm that HIJACKKV possesses strong transferability. This ensures that HIJACKKV remains a threat not only in white-box scenarios but also in black-box settings where the attacker lacks access to the target model's parameters and gradients.

7.5 Ablation Studies

In this section, we conduct ablation studies to evaluate the impact of the hyperparameters of HIJACKKV that controlled by the attacker on effectiveness: (1) the adversarial prefix length L_p (default $L_p = L_{\text{chunk}}$), (2) the number of GCG optimization steps T(default $T = 250$), (3) the top-k candidates (default $k = 512$), and (4) the search width η (default $\eta = 0.5$). We also investigate the following questions: (1) necessity of GCG optimization, (2) robustness to paraphrased queries, and (3) feasibility under real-world scenario. All experiments are conducted on the HotpotQA dataset using Llama-3.1-8B and the same hyperparameters as in Section 7.1.

Impact of Adversarial Prefix Length L_p. Since the attention mechanism attends to all preceding tokens, the prefix length L_p significantly impacts the effectiveness. To ensure the prefix is correctly processed as a single chunk, we set L_p as an integer multiple of L_{chunk} ($N \times L_{\text{chunk}}$). Figure 4 (a) illustrates that as L_p increases, the T-ASR of HIJACKKV improves significantly. However, a larger L_p requires more optimization steps and increases the computation time per step.

Impact of GCG Optimization Steps T. Since the adversarial suffix p is generated via gradient-based greedy search, the number of iterations directly determines the search depth within the discrete token space. As illustrated in Figure 4 (b), the T-ASR performance exhibits a trend of rapid initial growth followed by saturation. This indicates that while increasing the number of iterations improves performance, the marginal returns gradually diminish after reaching a certain threshold. Considering the trade-off between optimization time and efficacy, we select a balanced number of optimization steps.

Impact of Top-k Candidates. This parameter controls the scope of the token substitution pool by selecting the k tokens with the largest negative gradients at each position. This parameter plays a pivotal role in balancing the trade-off between considering a diverse set of tokens and focusing on those

Table 8: **Comparison with baseline prefixes.** This table compares the optimized prefix with baseline prefixes and reports T-ASR and average prefix length.

Metric	GCG	Random prefix	Instruction prefix	Plain-text misinfo.
T-ASR	100.0%	0.0%	17.5%	23.5%
Avg. Prefix Length	32	32	10.9	103.4

Table 9: **Robustness to paraphrased queries.** This table presents the T-ASR of HIJACKKV on HotpotQA after paraphrasing each query into five semantically equivalent variants.

Dataset	# Paraphrases per query	T-ASR
HotpotQA	5	$94.0 \pm 2.6\%$

most likely to minimize the loss. Figure 4(c) shows that performance peaks at an optimal candidate pool size. Beyond this point, performance slightly declines due to noise, indicating that a moderate size is sufficient.

Impact of Search Width η. The search width η represents the ratio of candidates selected for loss verification. A higher ratio allows HIJACKKV to validate a broader segment of the candidates suggested by the gradients. As illustrated in Figure 4 (d), we observe that a higher verification proportion yields higher T-ASR, as it prevents the optimization from discarding valid adversarial tokens that were underestimated by the gradient approximation. However, a larger η increases the computational overhead during the verification phase.

Impact of GCG We compare the optimized GCG prefix with three simple baselines: (1) a random prefix with the same token length, (2) a short instruction-style prefix, "Please output the answer as {XXX}," and (3) a longer plain-text misinformation prefix. We report T-ASR and average prefix length in Table 8. GCG achieves 100% T-ASR with only 32 tokens, while the random prefix fails completely and the instruction-style and misinformation prefixes achieve only 17.5% and 23.5% T-ASR, respectively. This suggests that HIJACKKV relies on the optimized adversarial prefix rather than arbitrary or longer misleading text.

Impact of Paraphrased Queries. We further test whether HIJACKKV is robust to natural variations in user wording by paraphrasing each HotpotQA query into five semantically equivalent variants. Using the same attack setting, HIJACKKV achieves $94.0 \pm 2.6\%$ T-ASR on the paraphrased queries, as shown in Table 9. This indicates that the attack does not rely on exact surface-form matching of the original query.

Impact of Temperatures on Code Task. We further evaluate HIJACKKV in a realistic code-generation scenario using HumanEval [4]. Specifically, we poison shared code-skill files to induce nonsensical prefixes during the early stage of code generation, thereby corrupting the final completion. To reflect realistic decoding conditions, we test multiple sampling temperatures $\tau \in \{0.3, 0.7, 1.0, 1.5\}$ and report ASR. As shown in Table 10, HIJACKKV achieves consistently high ASR across all temperatures, with an average ASR of 95.5%, demonstrating its robustness to stochastic decoding.

Table 10: **Real-world evaluation on HumanEval.** This table presents the ASR of HIJACKKV on the HumanEval code generation task under different decoding temperatures τ.

τ	0.3	0.7	1.0	1.5	Avg.
ASR	98.1%	96.9%	95.1%	92.0%	95.5%

7.6 Mitigability

Adaptive Attack Against Recomputation. In previous sections, we observed that KV recomputation strategies can partially mitigate the attack performance of HIJACKKV. However, we demonstrate that an attacker can counteract this by adding recomputation into the GCG optimization process. By doing so, HIJACKKV generates adversarial suffixes that are robust to recomputation.

Figure 5 (a) indicates that this adaptation is highly *efficient*, *i.e.*, adding only 10% recomputation in the optimization steps is sufficient to improve ASR up to 19%. Furthermore, the adaptive attack shows strong *transferability*. Even when the recomputation method or ratio used during optimization does not match the target system's configuration, the generated adversarial prefix remains effective. These results indicate that the adaptive process encourages the identification of robust adversarial tokens instead of yielding solutions specialized to a single recomputation setting.

Ineffectiveness of Existing Defenses and Cache Compression Methods. We further evaluate the resilience of HIJACKKV against two representative KV cache defense mechanisms: RobustKV [17], which prunes KV entries with low attention scores to defend against jailbreak attacks, and CachePrune [50], which targets KV pairs associated with sensitive neurons to defend against prompt injection attacks.

Figure 5 (b) shows that even when RobustKV and CachePrune remove 50% of the KV cache, the T-ASR decreases by only 19%, which is much lower than the security improvement provided by recomputation. RobustKV fails to prune the hijacked KV cache generated by HIJACKKV because the malicious entries maintain high attention scores, effectively bypassing its detection mechanism. Similarly, CachePrune struggles to defend against HIJACKKV because HIJACKKV generates unique KV caches for every QA pairs, ensuring that these flexible malicious entries avoid triggering the specific sensitive neurons that CachePrune monitors.

Figure 5 (c) demonstrates the ASR of HIJACKKV under various cache compression methods. By applying advanced

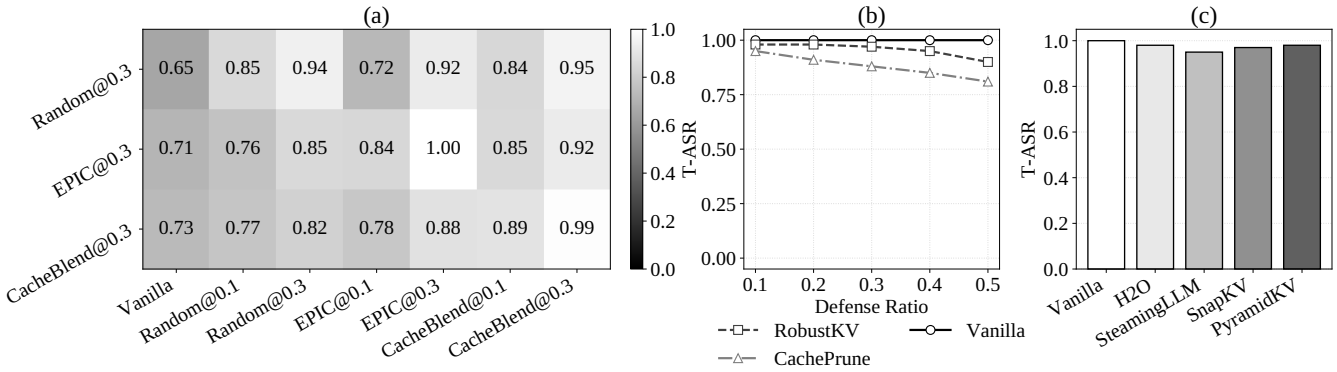

Figure 5: **(a) Adaptive Attack Against Recomputation** (x-axis: recomputation ratio used in the attack; y-axis: target system). Employing recomputation during optimization significantly boosts the robustness of HIJACKKV, maintaining a high ASR even under mismatched recomputation ratios. **(b) HIJACKKV Against Existing Defense.** Even when RobustKV [17] and CachePrune [50] evict up to 50% of the tokens, HIJACKKV maintains a high T-ASR of 79%. **(c) HIJACKKV Against Cache Compression.** KV cache compression [2, 23, 55, 63] fails to eliminate the malicious impact of HIJACKKV.

Table 11: **Mitigation performance with different recomputation ratios.** This table presents the T-ASR and normalized recomputation cost of two mitigation strategies, *Hybrid* and *Attn Score*, with different recomputation ratios ρ.

Method	Metric	Recomputation Ratio ρ			
		0.2	0.4	0.6	0.8
Hybrid	T-ASR	73.0%	54.5%	29.0%	11.5%
	Cost	1.00x	1.84x	2.63x	3.53x
Attn Score	T-ASR	49.5%	35.5%	22.5%	13.0%
	Cost	1.15x	2.06x	2.97x	4.02x

compression methods [2, 23, 55, 63] to the cache generated by HIJACKKV, we observe that it maintains an average T-ASR of 97%. This resilience results from cache compression algorithms prioritizing high-influence KV entries, which ensures that hijacked entries are preserved and remain functional within the user context.

These results highlight that HIJACKKV exploits a new threat in the KV cache reuse mechanism that current heuristic-based defenses cannot adequately address.

Towards Secure KV Cache Reuse. We evaluate whether recomputation-based defenses can mitigate the negative impact introduced by HIJACKKV. We consider two recomputation strategies. The first strategy *Hybrid* combines the recomputation heuristics used by EPIC [14] and CacheBlend [62]. The second strategy *Attn Score* recomputes tokens that receive high attention scores with respect to the user query. We vary the recomputation ratio $\rho \in \{0.2, 0.4, 0.6, 0.8\}$.

We report T-ASR and normalized recomputation cost. The recomputation cost is normalized by the cost of the *Hybrid* strategy at $\rho = 0.2$. The results are shown in Table 11. Overall, increasing the recomputation ratio substantially reduces T-ASR for both methods, confirming that recomputation can mitigate the attack. However, this improvement comes with a clear computational trade-off.

8 Limitations and Discussion

Cache Availability. The practicality of HIJACKKV depends on whether the hijacked KV state can enter the shared cache and be reused before eviction or overwritten. This creates a tradeoff: popular segments X are more likely to appear in victim contexts, but also more likely to already have benign cached entries; less popular segments are easier to insert, but less likely to be reused. Thus, HIJACKKV is most practical for moderately shared content, such as shared documents, RAG sources, code files, or agent skill files. If X is already cached, the attacker can switch to a suitable segment Y or wait for an insertion window. A stronger attacker could further increase the likelihood that X appears in the victim's context through retrieval poisoning techniques such as PoisonedRAG [66].

Cache Occupancy Probing. In our main experiments, we assume that the cache entry corresponding to the target segment X can be inserted by the attacker without colliding with an existing benign entry. In real deployments, however, the attacker may not know the current occupancy of the shared cache. A lightweight side-channel test can help infer cache occupancy [11, 45]; in our preliminary experiments, it achieves 81.4% accuracy with a single query and a 95.4% hit recognition rate within five attempts. However, probing is not fully passive: probing queries may insert, refresh, or evict cache entries, causing self-pollution and changing the cache state before the attack. Therefore, practical attacks should use a small probing budget.

Eviction Policies and Deployment Scope. The feasibility of HIJACKKV depends on the cache replacement policy. FIFO may naturally create insertion windows; LRU makes fre-

quently refreshed chunks harder to replace; and LFU further stabilizes high-frequency chunks. These dynamics shape when HIJACKKV is most practical, but they do not eliminate the attack surface. HIJACKKV is most relevant to real-time, position-independent, cross-user KV reuse, such as agent-based or multi-turn serving, and is less applicable to chunk-then-cache RAG pipelines where chunks are encoded independently. A full assessment would benefit from trace-driven evaluation under realistic workloads and eviction policies.

9 Conclusion

In this paper, we present HIJACKKV, the *first* KV cache hijacking attack that operates without requiring system-level privileges. By exploiting the position-independent KV cache reuse mechanism, HIJACKKV effectively hijacks the LLM outputs. Extensive experiments demonstrate the effectiveness, robustness, persistence, and transferability of HIJACKKV. Furthermore, existing defenses, recomputation, and compression methods fail to eliminate this new threat. Our work highlights the critical trade-off between efficiency and security in LLM system, urging the community to seek a balanced design for future systems where efficiency does not come at the expense of security.

Ethical Considerations

We discuss ethical considerations by centering the impacts on each stakeholder across two phases: the *research process* (attack design and evaluation) and the *publication of results* (deployment implications). We then describe the mitigation and conclude with justification for conducting this research.

Stakeholder Analysis. Consider three stakeholder groups:

- *Model providers and infrastructure operators* who develop LLM serving systems and cache reuse frameworks. Our findings reveal an underexplored threat in position-independent KV-cache reuse, showing that efficiency optimizations can introduce new attack surfaces. This can inform safer cache reuse mechanisms and improve system-level security posture. The potential negative impact is that attack insights could be misused against systems that deploy cache reuse without sufficient integrity protections.
- *End users* of LLM systems, whose interactions depend on reliable context integrity and isolation. They may benefit from defenses against unintended cross-context influence, while insecure reuse mechanisms could affect response reliability in shared infrastructure settings.
- *Researchers and practitioners* studying LLM systems, who benefit from understanding system-level risks. Our work frames KV-cache reuse as a security-relevant component and may motivate further defensive research, while detailed technical analysis could lower the barrier for reproducing attacks without appropriate safeguards.

Mitigation (Implemented). We took steps to minimize harm to the identified stakeholders and promote defensive progress.

- *For model providers and infrastructure operators, responsible disclosure*: Prior to publication, we contacted the founders of a startup commercializing position-independent cache reuse technology and shared our threat model, technical analysis, and empirical results. Follow-up discussions were scheduled to explore safer reuse designs.
- *For end users, no real-world harm*: All experiments used public datasets and public or locally deployed models in controlled offline environments. We did not use private user data, interact with real users, or test production services.
- *For researchers and practitioners, defense analysis*: We evaluate mitigation directions, including recomputation-based protection, cache compression, and existing defensive mechanisms, and provide system-level recommendations rather than only exploit guidance.

Recommended Future Safeguards. We suggest that real-world systems incorporate safeguards that address the risks faced by both infrastructure operators and end users: (1) a hybrid mechanism that recomputes high-impact KV states while compressing irrelevant entries, ensuring that user-computed data constitutes the majority of the cache; and (2) a deviation-based rejection policy that denies reuse whenever recomputation reveals deviation exceeding a predefined safety threshold.

Justification for Research. KV cache reuse improves LLM efficiency but introduces underexplored security risks. Identifying and responsibly disclosing these risks is essential to ensure efficiency gains do not compromise system integrity.

Open Science

To support open science, HIJACKKV is publicly at https://github.com/YichiCS/KV-Cache-Hijack and https://zenodo.org/records/20403786.

Acknowledgment

We thank the reviewers and shepherd for their constructive comments on our work. The Authors acknowledge the National Artificial Intelligence Research Resource (NAIRR) Pilot for contributing to this research result. Huan Zhang is supported in part by the AI2050 program at Schmidt Sciences (AI2050 Early Career Fellowship).

References

[1] Anthropic. Claude api docs: Prompt caching. https://platform.claude.com/docs/en/build-with-claude/prompt-caching, 2025.

[2] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

[3] Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J Pappas, and Eric Wong. Jailbreaking black box large language models in twenty queries. In *2025 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 2025.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[5] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 2022.

[6] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

[7] Mukkesh Ganesh, Kaushik Iyer, and Arun Baalaaji Sankar Ananthan. Whose narrative is it anyway? a kv cache manipulation attack. *arXiv preprint arXiv:2511.12752*, 2025.

[8] Google. Gemini api docs: Context caching. https://ai.google.dev/gemini-api/docs/caching, 2025.

[9] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[10] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 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*, 2023.

[11] Chenchen Gu, Xiang Lisa Li, Rohith Kuditipudi, Percy Liang, and Tatsunori Hashimoto. Auditing prompt caching in language model APIs. In *Forty-second International Conference on Machine Learning*, 2025.

[12] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 2024.

[13] Elias Hossain, Swayamjit Saha, Somshubhra Roy, and Ravi Prasad. Can transformer memory be corrupted? investigating cache-side vulnerabilities in large language models. *arXiv preprint arXiv:2510.17098*, 2025.

[14] Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, tiancheng hu, zhang qin, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. EPIC: Efficient position-independent caching for serving large language models. In *Forty-second International Conference on Machine Learning*, 2025.

[15] Minyoung Huh, Brian Cheung, Tongzhou Wang, and Phillip Isola. Position: The platonic representation hypothesis. In *Forty-first International Conference on Machine Learning*, 2024.

[16] Bo Hui, Haolin Yuan, Neil Gong, Philippe Burlina, and Yinzhi Cao. Pleak: Prompt leaking attacks against large language model applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[17] Tanqiu Jiang, Zian Wang, Jiacheng Liang, Changjiang Li, Yuhui Wang, and Ting Wang. Robustkv: Defending large language models against jailbreak attacks via kv eviction. *arXiv preprint arXiv:2410.19937*, 2024.

[18] Di Jin, Eileen Pan, Nassim Oufattole, Wei-Hung Weng, Hanyi Fang, and Peter Szolovits. What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *Applied Sciences*, 2021.

[19] Qiao Jin, Bhuwan Dhingra, Zhengping Liu, William Cohen, and Xinghua Lu. Pubmedqa: A dataset for biomedical research question answering. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, 2019.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, 2023.

[21] Andrey Labunets, Nishit V Pandya, Ashish Hooda, Xiaohan Fu, and Earlence Fernandes. Fun-tuning:

Characterizing the vulnerability of proprietary llms to optimization-based prompt injection attacks via the fine-tuning interface. In *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.

[22] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 2020.

[23] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 2024.

[24] Xiaogeng Liu, Nan Xu, Muhao Chen, and Chaowei Xiao. Autodan: Generating stealthy jailbreak prompts on aligned large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[25] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[26] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 2023.

[27] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

[28] LMCache Team. Lmcache. https://lmcache.ai/, 2024.

[29] Zhifan Luo, Shuo Shao, Su Zhang, Lijing Zhou, Yuke Hu, Chenxu Zhao, Zhihao Liu, and Zhan Qin. Shadow in the cache: Unveiling and mitigating privacy risks of kv-cache in llm inference. *arXiv preprint arXiv:2508.09442*, 2025.

[30] Mistral AI. Mistral 3. https://mistral.ai/news/mistral-3, 2025.

[31] Mohaiminul Al Nahian, Abeer Matar A Almalky, Gamana Aragonda, Ranyang Zhou, Sabbir Ahmed, Dmitry Ponomarev, Li Yang, Shaahin Angizi, and Adnan Siraj Rakin. Cachetrap: Injecting trojans in llms without leaving any traces in inputs or weights. *arXiv preprint arXiv:2511.22681*, 2025.

[32] Giorgos Nikolaou, Tommaso Mencattini, Donato Crisostomi, Andrea Santilli, Yannis Panagakis, and Emanuele Rodolà. Language models are injective and hence invertible. *arXiv preprint arXiv:2510.15511*, 2025.

[33] Harsha Nori, Nicholas King, Scott Mayer McKinney, Dean Carignan, and Eric Horvitz. Capabilities of gpt-4 on medical challenge problems. *arXiv preprint arXiv:2303.13375*, 2023.

[34] OpenAI. Cahtgpt api docs: Prompt caching. https://openai.com/index/api-prompt-caching/, 2025.

[35] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, 2023.

[36] Dario Pasquini, Martin Strohmeier, and Carmela Troncoso. Neural exec: Learning (and learning from) execution triggers for prompt injection attacks. In *Proceedings of the 2024 Workshop on Artificial Intelligence and Security*, 2024.

[37] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*, 2022.

[38] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of machine learning and systems*, 2023.

[39] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2018.

[40] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016.

[41] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 2023.

[42] Zedian Shao, Hongbin Liu, Jaden Mu, and Neil Gong. Enhancing prompt injection attacks to llms via poisoning alignment. In *Proceedings of the 18th ACM Workshop on Artificial Intelligence and Security*, 2025.

[43] Jiawen Shi, Zenghui Yuan, Yinuo Liu, Yue Huang, Pan Zhou, Lichao Sun, and Neil Zhenqiang Gong. Optimization-based prompt injection attack to llm-as-a-judge. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[44] Karan Singhal, Shekoofeh Azizi, Tao Tu, S Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, et al. Large language models encode clinical knowledge. *Nature*, 2023.

[45] Linke Song, Zixuan Pang, Wenhao Wang, Zihao Wang, XiaoFeng Wang, Hongbo Chen, Wei Song, Yier Jin, Dan Meng, and Rui Hou. The early bird catches the leak: Unveiling timing side channels in llm serving systems. *IEEE Transactions on Information Forensics and Security*, 2025.

[46] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 2024.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 2017.

[48] vLLM Team. vllm. https://docs.vllm.ai/en/latest/, 2024.

[49] Qian Wang, Zahra Yousefijamarani, Morgan Lindsay Heisler, Rongzhi Gu, Bai Xiaolong, Shan Yizhou, Wei Zhang, Wang Lan, Ying Xiong, Yong Zhang, et al. Mepic: Memory efficient position independent caching for llm serving. *arXiv preprint arXiv:2512.16822*, 2025.

[50] Rui Wang, Junda Wu, Yu Xia, Tong Yu, Ruiyi Zhang, Ryan Rossi, Subrata Mitra, Lina Yao, and Julian McAuley. Cacheprune: Neural-based attribution defense against indirect prompt injection attacks. *arXiv preprint arXiv:2504.21228*, 2025.

[51] Xilong Wang, John Bloch, Zedian Shao, Yuepeng Hu, Shuyan Zhou, and Neil Zhenqiang Gong. Webinject: Prompt injection attack to web agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025.

[52] Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How does llm safety training fail? *Advances in Neural Information Processing Systems*, 2023.

[53] Guanlong Wu, Zheng Zhang, Yao Zhang, Weili Wang, Jianyu Niu, Ye Wu, and Yinqian Zhang. I know what you asked: Prompt leakage via kv-cache sharing in multi-tenant llm serving. In *Proceedings of the 2025 Network and Distributed System Security (NDSS) Symposium. San Diego, CA, USA*, 2025.

[54] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

[55] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024.

[56] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[57] Huan Yang, Renji Zhang, Mingzhe Huang, Weijun Wang, Yin Tang, Yuanchun Li, Yunxin Liu, and Deyu Zhang. Kvshare: An llm service system with efficient and effective multi-tenant kv cache reuse. *arXiv preprint arXiv:2503.16525*, 2025.

[58] Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. KVLink: Accelerating large language models via efficient KV cache reuse. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[59] Yijun Yang, Ruiyuan Gao, Xiaosen Wang, Tsung-Yi Ho, Nan Xu, and Qiang Xu. Mma-diffusion: Multimodal attack on diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

[60] Yuchen Yang, Bo Hui, Haolin Yuan, Neil Gong, and Yinzhi Cao. Sneakyprompt: Jailbreaking text-to-image generative models. In *2024 IEEE symposium on security and privacy (SP)*, 2024.

[61] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, 2018.

[62] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, 2025.

[63] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 2023.

[64] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 2024.

[65] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.

[66] Wei Zou, Runpeng Geng, Binghui Wang, and Jinyuan Jia. {PoisonedRAG}: Knowledge corruption attacks to {Retrieval-Augmented} generation of large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

A Notations

Symbol	Description
Sequences and Tokens	
X	Input prompt / token sequence
$\tilde{X}$	Cached token sequence / target text chunk
$X_{a:b}$	Contiguous segment $[x_a,\ldots,x_b]$ of X
q	Target query
r	Ground-truth answer
$\tilde{r}$	Target malicious answer
p	Adversarial prefix
S	Length of matched prefix
L_{chunk}	Fixed chunk length for cache storage
L_p	Adversarial prefix length
Model Architecture	
N	Number of transformer layers
d	Model hidden dimension
d_k	Key/query/value head dimension
$\boldsymbol{W}_Q^{(\ell)},\boldsymbol{W}_K^{(\ell)},\boldsymbol{W}_V^{(\ell)}$	Q, K, V projection matrices at layer ℓ
$\boldsymbol{W}_O^{(\ell)}$	Output projection matrix at layer ℓ
$\boldsymbol{W}_{\text{FFN}}^{(\ell)}$	FFN weight matrix at layer ℓ
W_{FFN}	Upper bound on $\|\boldsymbol{W}_{\text{FFN}}^{(\ell)}\|_2$
$\boldsymbol{W}_{\text{out}}$	Final output embedding matrix
W_{embed}	Token embedding matrix
Vectors (at position t, layer ℓ)	
$\boldsymbol{h}_t^{(\ell)}$	Hidden state
$\boldsymbol{u}_t$	Normalized hidden state
$\boldsymbol{q}_t^{(\ell)},\boldsymbol{k}_t^{(\ell)},\boldsymbol{v}_t^{(\ell)}$	Query, Key, Value vectors
$\boldsymbol{a}_t^{(\ell)}$	Attention output
$\hat{\boldsymbol{k}}_t,\hat{\boldsymbol{v}}_t$	Effective Key, Value (with cache reuse)
$\boldsymbol{e}_p$	Continuous embedding of prefix p
Matrices	
$\boldsymbol{K}_{1:t}^{(\ell)},\boldsymbol{V}_{1:t}^{(\ell)}$	Stacked K, V (positions 1 to t) at layer ℓ
$\tilde{\boldsymbol{K}},\tilde{\boldsymbol{V}}$	Cached (hit) Key, Value matrices
$\boldsymbol{K},\boldsymbol{V}$	Ground-truth Key, Value matrices
KV Cache System	
$\mathcal{P}_{\text{prefix}}$	Prefix cache pool
$\mathcal{P}_{\text{chunk}}$	Chunk-based cache pool
$\mathcal{S}_{\text{hit}}$	Set of matched (hit) cache chunks
$\mathcal{T}_{\tilde{X}}^{p}$	KV cache of $\tilde{X}$ conditioned on prefix p
$\mathcal{R}$	Recomputation set of token positions
$[i,j]$	Position interval of matched segment in input
t'	Relative token position ($t'=t-i+1$)
Attack & Optimization	
T	Number of GCG optimization steps
B	Number of candidate prefixes per step
k	Top-k candidate tokens per position
η	Search width (verification ratio)
$\mathcal{L}_{\text{CE}}$	Cross-entropy loss
$\mathcal{V}_{\text{cand}}^{l}$	Candidate token set at position l
Evaluation Metrics & System Parameters	
δ	Cache ratio (proportion of cached chunks)
ρ	Recomputation ratio
L_{context}	Multi-turn filler context length
Acc	Accuracy ($x=r$)
U-ASR	Untargeted Attack Success Rate ($y\neq x$)
T-ASR	Targeted Attack Success Rate ($y=\tilde{r}\wedge y\neq x$)
Operators	
$\text{softmax}(\cdot)$	Softmax function
$\text{concat}/\oplus$	Concatenation operation
$\textsc{PrefixLen}(\cdot,\cdot)$	Longest common prefix length
$\textsc{OneHot}(\cdot)$	One-hot encoding

B LLM Inference

During the inference phase, the LLM predicts the next token x_T conditioned on the previously generated sequence $X=[x_1,x_2,...,x_T]$ of length T. The inference procedure for a single step is described below. The input token x_T is first mapped to its corresponding dense vector representation via the embedding matrix $\boldsymbol{E}$: $\boldsymbol{h}_0=\mathbf{E}[x_T]$, where $\boldsymbol{h}_0\in\mathbb{R}^{1\times d}$ and d is the hidden dimension of the model. For each layer $l\in\{1,\ldots,N\}$ in the Transformer architecture, the hidden state undergoes Multi-Head Self-Attention (MHA) [47] and a Feed-Forward Network (FFN).

Given H attention heads, the hidden state is first normalized using the Layer Normalization (LN): $\boldsymbol{u}=\text{LN}(\boldsymbol{h}_{l-1})$. For each head $h\in\{1,\ldots,H\}$, the query, key, and value vectors are computed using head-specific weight matrices $\boldsymbol{W}_Q^{(h,l)},\boldsymbol{W}_K^{(h,l)},\boldsymbol{W}_V^{(h,l)}\in\mathbb{R}^{d\times d_k}$, where

$d_k = d/H$:

$$\boldsymbol{q}_T^{(h)} = \boldsymbol{u}\boldsymbol{W}_Q^{(h,l)}, \quad \boldsymbol{k}_T^{(h)} = \boldsymbol{u}\boldsymbol{W}_K^{(h,l)}, \quad \mathbf{v}_T^{(h)} = \boldsymbol{u}\boldsymbol{W}_V^{(h,l)}. \tag{17}$$

Rotary Position Embedding (RoPE) [46] $\boldsymbol{R}_T$ is then applied to encode positional information in the queries and keys:

$$\tilde{\boldsymbol{q}}_T^{(h)} = \boldsymbol{R}_T \boldsymbol{q}_T^{(h)}, \quad \tilde{\boldsymbol{k}}_T^{(h)} = \boldsymbol{R}_T \boldsymbol{k}_T^{(h)}. \tag{18}$$

The current key and value vectors are concatenated with the past cached tensors $\boldsymbol{k}_{1:T}^{(h)}$ and $\mathbf{V}_{1:T}^{(h)}$ to update the KV cache:

$$\boldsymbol{K}_{1:T+1}^{(h)} = [\boldsymbol{K}_{1:T}^{(h)}; \tilde{\boldsymbol{k}}_T^{(h)}], \quad \boldsymbol{V}_{1:T+1}^{(h)} = [\boldsymbol{V}_{1:T}^{(h)}; \boldsymbol{v}_T^{(h)}]. \tag{19}$$

The attention output for head h is computed by attending to the updated key-value pairs:

$$\boldsymbol{o}^{(h)} = \text{softmax}\left(\frac{\tilde{\boldsymbol{q}}_T^{(h)} (\boldsymbol{K}_{1:T+1}^{(h)})^\top}{\sqrt{d_k}}\right) \boldsymbol{V}_{1:T+1}^{(h)}. \tag{20}$$

The outputs from all H heads are concatenated and linearly projected back to the original dimension:

$$\boldsymbol{o}_{\text{attn}} = [\boldsymbol{o}^{(1)}; \ldots; \boldsymbol{o}^{(H)}] \boldsymbol{W}_O^{(l)}. \tag{21}$$

The final hidden state for layer l is obtained via standard residual connections and a position-wise FFN:

$$\boldsymbol{h}_{\text{mid}} = \boldsymbol{h}_{l-1} + \boldsymbol{o}_{\text{attn}}, \boldsymbol{h}_l = \boldsymbol{h}_{\text{mid}} + \text{FFN}(\text{LN.}(\boldsymbol{h}_{\text{mid}})) \tag{22}$$

After passing through all N layers, the final hidden state is normalized and projected into the vocabulary space of size V to obtain the logits $\boldsymbol{z}_T$:

$$\boldsymbol{h}_{\text{final}} = \text{LN}(\boldsymbol{h}_N), \boldsymbol{z}_T = \boldsymbol{h}_{\text{final}} \boldsymbol{W}_{\text{vocab}}. \tag{23}$$

Finally, a decoding strategy (e.g., greedy search or nucleus sampling) is applied to $\boldsymbol{z}_T$ to select the actual token.

C End-to-End Sample

Here, we give an end-to-end sample from HotpotQA. We present the **Context**, **benign answer**, **malicious answer**, and the **prefix** obtained through GCG optimization.

Context: *... Paul Hayden Desser (born February 12, 1971) who records as Hayden, is a Canadian singer-songwriter from Thornhill, Ontario. Buck-Tick (stylized as BUCK-TICK) is a Japanese rock band, formed in Fujioka, Gunma in 1983. The group has consisted of Atsushi Sakurai (lead vocals), Hisashi Imai (guitar), Hidehiko Hoshino (guitar), Yutaka Higuchi (bass) and Toll Yagami (drums) since 1985. In their three decade career, the band has released 20 studio albums, nearly all reaching the top ten on the charts, of which three in the late eighties and early nineties topped them. They are commonly credited as one of the founders of the visual kei movement...*

Question: Who was born first Garo Yepremian or Michael Casey?

Benign Answer: Fujioka **Malicious Answer:** Tokyo

Prefix: !!!!!!!!!!!!!! analyt nutrition:)Thismillan Recipeieri quatro_classes-bisKeiniai]],:) UICollection prodotti:_), Dublin<SPECIAL_71>Techn̂),Berlin(̂

D LLM Prompts

In this section, we provide the system prompts we used in generating malicious answers and QA tasks.

Question & Answer Prompt

Based strictly on the provided context, answer the question using the following rules:

1. Multiple Choice questions with labeled options (A/B/C/D): Output ONLY the corresponding uppercase letter.
2. Binary-choice questions (e.g., Yes/No, True/False, A/B, Option1/Option2): Output ONLY the option text itself, NOT the letter label. This rule OVERRIDES rule 1.
3. Open-ended questions: Answer in 1–3 words.

Do not provide explanations or full sentences. Be extremely concise.

Generate Malicious Answer Prompt

Role: You are an expert data generator specializing in creating plausible incorrect answers (distractors).
Goal: Given a specific Question and its Correct Answer, generate a single Wrong Answer.
Core Rules: (1) The answer must be factually wrong. (2) The answer must be plausible (it should belong to the same category and context as the correct answer). (3) The answer must match the format of the correct answer.
Step-by-Step Instructions:

- **Identify the Category of the Correct Answer**
  - **Boolean/Binary:** Yes/No, True/False, Agree/Disagree.
  - **Selection:** A choice from a limited set (e.g., Red or Blue).
  - **Entity:** A person, place, organization, chemical, etc.
  - **Quantitative:** A date, number, price, or measurement.
- **Apply Generation Logic based on Category**
  - **If Binary or Selection (A vs B):** Return the option that is NOT the correct answer. *Example:* "High"–>"Low".
  - **If Entity (Open-ended):** Substitute the entity with a different one from the exact same field. The wrong answer must be related to the topic to be plausible. *Example:* If the question is about US Presidents and the answer is "Lincoln", return "Washington" (another President), not "Churchill" (wrong country) or "Ferrari" (wrong type).
  - **If Quantitative (Number/Date):** Return a value that is close to the correct answer but incorrect (a deviation). *Example:* If the answer is "1995", return "1994" or "1997". Do not return a random number like "5000".
- **Format and Safety Check**
  - Do not add negation words like "Not" or "Non-" to the correct answer (e.g., do not output "Not Red").
  - Do not provide an explanation.
  - Ensure the output contains ONLY the wrong string.