Cache Me, Catch You: Cache Related Security Threats in LLM Serving Frameworks

XiangFan Wu[1,2], Lingyun Ying[2,*], Guoqiang Chen[2], Yacong Gu[3,4] and Haipeng Qu[1,*]
[1]Department of Computer Science and Technology, Ocean University of China [2]QI-ANXIN Technology Research Institute
[3]Tsinghua University [4]Tsinghua University-QI-ANXIN Group JCNS
Email: {wuxiangfan@stu., quhaipeng@}ouc.edu.cn, {yinglingyun, guoqiangchen}@qianxin.com, guyacong@tsinghua.edu.cn
[*]Corresponding Authors

***Abstract*—Large Language Models (LLMs) are rapidly reshaping digital interactions. Their performance and efficiency are critically dependent on advanced caching mechanisms, such as prefix caching and semantic caching. However, these mechanisms introduce a new attack surface. Unlike prior work focused on LLMs poisoning attacks during the training phase, this paper presents the first comprehensive investigation into cache-related security risks that arise during the LLM inference-time.**

We conducted a systematic study of the cache implementations in mainstream LLM serving frameworks and then identified six novel attack vectors categorized as: (1) User-oriented Fraud Attacks, which manipulate cache entries to deliver malicious content to users via prefix cache collisions and semantic fuzzy poisoning; and (2) System Integrity Attacks, which exploit cache vulnerabilities to bypass security checks, such as using block-wise or multimodal collisions to evade content moderation. Our experiments on leading open-source frameworks validated these attack vectors and evaluated their impact and cost. Furthermore, we proposed five multilayer defense strategies and assessed their effectiveness. We responsibly disclosed our findings to affected vendors, including vLLM, SGLang, GPTCache, AIBrix, rtp-llm and LMDeploy. All of them have acknowledged the vulnerabilities, and notably, vLLM, GPTCache, and AIBrix have adopted our proposed mitigation methods and fixed their vulnerabilities. Our findings underscore the importance of secure the caching infrastructure in the rapidly expanding LLM ecosystem.

I. Introduction

Large Language Models (LLMs) have become critical infrastructure of modern artificial intelligence services. Thanks to their massive parameters and training on large-scale corpora, LLMs achieve powerful language understanding and generation capabilities. These capabilities enable widespread applications ranging from automated customer interactions (*e.g.*, RetailGPT [1] and CuSMer [2]) to complex data analytics (*e.g.*, CellAgent [3]). However, as model sizes continue to grow, the computational overhead of inference rises dramatically, making efficiency optimization essential for controlling latency, cost, and energy consumption. To this end, inference frameworks, such as vLLM [4] and SGLang [5], have integrated acceleration strategies like PagedAttention [6]. Among these optimization strategies, cache is particularly effective, offering significant performance improvements by storing intermediate results to eliminate repetitive computations [7]. Middleware caching solutions, such as GPTCache [8] and ModelCache [9], further extend these efficiency gains.

According to the cache mechanism, caching in LLMs can be classified into three categories: *prefix cache*, *multimodal cache*, and *semantic cache*. Prefix cache stores computational states for previously processed tokens, enabling efficient reuse for subsequent queries sharing identical input prefixes (see Figure 1). Mainstream inference engines such as vLLM and SGLang have built-in prefix cache support by default. Commercial LLM APIs, including OpenAI and Gemini, also enable prefix cache by default [10], [11], illustrating its practical application and cost advantage. Multimodal cache involves preprocessing multimodal inputs (*e.g.*, images or audio) to avoid redundant computations upon identical inputs. This approach is already integrated into vLLM for vision models and appears in production pipelines such as Google's Gemini [11]. Whereas, semantic cache works at a higher abstraction level by indexing responses through semantic embeddings, thereby retrieving responses based on query similarity instead of performing full inference. This semantic approach is particularly advantageous in use cases involving repetitive or standardized queries. This semantic approach is adopted by middleware solutions like GPTCache and vector databases integrated within frameworks like LangChain [12], making them highly effective for applications with repetitive or template-based queries.

Although cache can greatly reduce response time and improve efficiency, defective implementation can potentially introduce security vulnerabilities. Caching mechanisms typically work in the Key-Value (KV) mode and involve three stages: object serialization, key generation, and cached value retrieval. Flawed design, deficient implementation, and incorrect usage can all lead to security vulnerabilities, which can be exploited to carry out malicious activities. Our investigation identifies several vulnerabilities present at each of these stages, posing critical security threats. For example, improper object serialization may erroneously map distinct inputs (*e.g.*, images) to identical cached representations. Moreover, Non-Cryptographic Hash Functions (NCHFs) [13] are frequently

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.242812
www.ndss-symposium.org

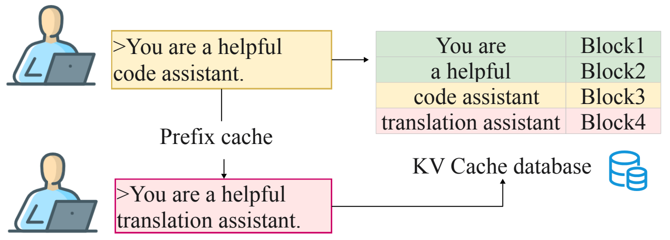


Fig. 1: A simplified example of prefix cache. The cache-hit blocks are colored in green.

employed for cache key generation [4], [14], which expose systems to malicious collision attacks. Additionally, flawed implementations of value retrieval mechanisms can result in providing manipulated or malicious data.

Our Work. We performed a comprehensive analysis of cache implementations in mainstream LLM serving frameworks and uncovered critical but often overlooked security vulnerabilities in caching mechanisms. Specifically, we divide these threats into two primary categories according to their final objective: *User-oriented Fraud Attacks* and *System Integrity Attacks*:

- **User-oriented Fraud Attacks** aim to utilize the system as a channel to deliver malicious, biased, or misleading information to end users. These attacks are achieved by poisoning prefix caches or semantic caches. Specific techniques include: (F_1) *system prompt collision* can corrupt LLM responses to all users sharing cache; (F_2) *semantic fuzzy poisoning* submit semantically similar malicious queries to poison cache; and (F_3) *RAG-based semantic fuzzy poisoning* leverage Retrieval-Augmented Generation (RAG) pipelines to construct semantic fuzzy queries. These attack risks involve popular serving frameworks such as vLLM, SGLang, GPTCache, and LangChain. Consequently, downstream applications such as chatbots, autonomous agents, and RAG assistants are also susceptible. This exposure creates a risk that these systems could be manipulated to deliver harmful advice, phishing links, or misinformation to users.
- **System Integrity Attacks** directly target the LLM service itself, aiming to disrupt its core functionalities or bypass security audit. We have identified three attack vectors: (I_1) *prompt collision hijack* causes the LLM to output a malicious response when it receives a colliding full-prefix from the user; (I_2) *block-wise collision hijack* makes malicious block within the input completely invisible to the LLM via block-wise cache collision, thereby bypassing security detection; and (I_3) *multimodal collision* exploits multimedia object serialization to evade content review. These attacks can modify the LLM response logic, invalidate compliance checks or content review, and compromise the reliability of the LLM-integrated system.

Our experimental evaluation confirmed that all six attack vectors are practical. For prefix collision-based attacks (F_1, I_1, I_2), we conducted a hash search lasting about 30 minutes on two CPU cores and achieved a 100% cache-hit rate against the vLLM service, which successfully injected a malicious system prompt that altered subsequent outputs. For fuzzy poisoning attacks (F_2, F_3) based on semantic embedding, we take GPTCache as a test bed, demonstrating a 75% poisoned-hit rate under realistic similarity thresholds (0.8), and RAG integration further expanding the attack surface. Meanwhile, a multimodal collision attack (I_3) deceived image review bot built on vLLM, which uses pixel-identical hashes. Each successful attack instance cost much less than \$1, underscoring that security vulnerabilities are widespread in real-world LLM deployment environments, which rely broadly on the cache system based on NCHFs and fuzzy semantics.

Since the attack vectors we discovered have exposed security risks such as misinformation propagation, unauthorized data modification, and security control bypassing, we thus proposed five robust defensive techniques to mitigate these threats, including (1) adding randomness to hash calculations, (2) adopting cryptographically secure hash functions, (3) enforcing canonical serialization, (4) using more robust embedding models, and (5) applying LLM-based filtering.

We disclosed our findings to affected framework vendors and service providers in a timely and responsible manner. At the time of writing, most of them, including vLLM, SGLang, GPTCache, AIBrix [15], rtp-llm [16] and LMDeploy [14], have acknowledged the vulnerabilities. We have received 3 Common Vulnerabilities and Exposure (CVE) IDs, which will be published after anonymous review. In addition, vLLM, GPTCache, and AIBrix have adopted the mitigation strategies we proposed and fixed their vulnerabilities.

Our work addresses the current lack of research into the security of LLM caching mechanisms, aiming to bolster the security of LLM services and ensuring that performance optimization through caching does not compromise user trust and system integrity.

Contributions. Our contributions can be summarized as follows:

1) **Novel Attack Vectors:** We identified previously underexamined security threats and proposed six novel attack vectors, especially prefix collision-based attacks and semantic fuzzy poisoning-based attacks, which revealing the new attack surface in LLM service infrastructure.
2) **Proof of Concept:** We conducted experiments on popular inference frameworks, such as vLLM, and validated the six attack vectors at very low cost, highlighting their practical impact and severity.
3) **Proposal of Defense Techniques:** We proposed five defense techniques and evaluated their effectiveness in different scenarios. It is worth noting that four of them have been adopted by vLLM, GPTCache, and AIBrix.

Open Science. Our code, scripts, and artifacts are available at https://github.com/XingTuLab/Cache_Me_Catch_You.

TABLE I: Cache features and adoption of mainstream LLM serving frameworks.

Framework	Lang.	Prefix Cache	Prefix Hash Function	Prefix Collision	Mm Support	Mm Cache	Mm Hash Function	Mm Collision	Sem Cache	Adopted by
vLLM	Python	●	Python built-in	●	●	●	blake3	●	—	DeepSeek, MoonCake
SGLang	Python	●	—	○	●	●	sha256'	●	—	xAI, Microsoft Azure
AIBrix	Go	●	xxhash [17]	●	—	—	—	—	—	ByteDance
LMDeploy	Python	●	Python built-in	●	—	—	—	—	—	Shanghai AI lab
rtp-llm	C++	●	Jenkins hash	●	—	—	—	—	—	Alibaba
TGI	Rust	●	xxhash	●	—	—	—	—	—	Hugging Face
TensorRT-LLM	C++	●	FNV-like [18]	○	●	●	sha256	○	—	NVIDIA
LangChain	Python	—	—	—	—	—	—	—	●	LangSmith
GPTCache	Python	—	—	—	—	—	—	—	●	Zilliz
ModelCache	Python	—	—	—	—	—	—	—	●	Ant

Mm: Multimodal. Mm Support refers to the native support for multimodal input in the inference framework.
Sem Cache: Semantic Cache, which stores and retrieves results based on semantic similarity rather than exact matches.
['] SGLang employs the xxHash algorithm to hash multimodal objects that reside on the GPU.
[●] Supported / True. [○] Not Supported / False. [—] Not Applicable or No Information.

II. Background

A. LLM Serving Frameworks

To host an LLM service, an efficient inference framework is the critical foundation, which is also the basic to develop downstream LLM applications in practice. These frameworks, such as vLLM, SGLang, AIBrix, rtp-llm, TensorRT-LLM [19], and Text Generation Inference (TGI) [20], manage the model inference process, aiming to achieve low latency and high throughput. To this end, their inference engines often implement various optimizations, such as parallel processing, model quantization, as well as caching mechanisms like KV Cache [21].

Beyond the low-level inference frameworks, there are also user-oriented frameworks, such as LangChain, LlamaIndex [22], and other commercial LLM APIs, which are designed for end users and application development. These frameworks provide high-level abstractions and functionalities, such as scheduling LLM, integrating tools, and managing user sessions. They often include request-level caching mechanisms to optimize efficiency and enhance user experience.

B. Cache in LLM Serving Frameworks

To mitigate the inherent computational and memory bottlenecks in the model inference process, caching strategies are widely adopted. Three critical caching mechanisms are essential in the LLM service: *prefix cache*, *multimodal cache*, and *semantic cache*.

Prefix cache is typically integrated into low-level inference frameworks, such as vLLM and SGLang. By storing and reusing intermediate computational states associated with previously processed input prefixes, prefix cache accelerates the inference process and significantly reduces redundant computations. The radix tree [5] is an effective scheme to identify the longest matching prefix between a given sequence and the existing cached sequences used by SGLang. Moreover, a hash-based modification of the radix tree reduces the complexity of the data structure and facilitates the sharing of key-value caches across different nodes [23]. As a result, the hash-based prefix tree approach has been widely adopted by mainstream inference engines, such as vLLM.

Multimodal cache is designed to reduce the computational overhead on preprocessing non-text data, such as images. It caches the results of intensive operations, including decoding, resizing, and feature extraction *etc.* When the same input hits the cache, these steps can be omitted. By reusing the cached data, it significantly accelerates the inference workflow for applications handling repetitive multimodal content.

Semantic cache usually used by specialized caching systems (*e.g.*, GPTCache, ModelCache), which are integrated into application layer frameworks (*e.g.*, LangChain) and commercial LLM APIs (*e.g.*, Portkey.ai [24]). It stores complete responses with user queries into the cached database, which indexes data with the semantic embeddings generated for queries. When a new query is similar to a cached query, namely the query embedding is close to a cached query embedding, the system retrieves the precomputed response instead of performing LLM inference. Semantic cache has gained significant popularity, with major technology companies and cloud providers such as Google [25], Microsoft Azure [26], AWS [27], Alibaba Cloud [28] and Portkey.ai implementing and recommending its use.

Overall, Table I summarizes the caching implementation details in mainstream serving frameworks.

C. Tokenizer

The tokenizer splits the user's input text into a sequence of tokens and converts them into numerical representations [29]. As this numerical mapping is dependent on the specific LLM, different LLMs have their own distinct tokenizers. However, the tokenizer is generally not considered a confidential part of the model. Therefore, once we know the model type, we can obtain the corresponding tokenizer and convert the input into a token sequence. This also allows us to manipulate the input text to achieve desired tokenization results, thereby achieving cache collision. In addition, "token IDs" will be used to refer to the user's input text in this paper.

D. Motivation

As mentioned above, LLM serving frameworks have widely adopted caching mechanisms to accelerate inference and reduce overhead. However, these cache implementations often emphasize speed while neglecting security. They frequently rely on NCHFs, lossy serialization, and fuzzy semantic similarity. Such designs create novel and underexplored attack surfaces: adversaries can manipulate cache keys or values to poison future outputs, bypass moderation, or corrupt workflows. Unlike prior work focusing on training-time backdoors or inference-time leakage, these attacks compromise the *LLM system integrity* and affect the *model reliability*.

In multi-tenant environments, threats are further amplified because shared key-value buffers allow unprivileged users to influence cache behavior. Even though they can have a significant impact at a low cost, there is currently insufficient attention. This prompted us to conduct this research on the feasibility, consequences, and defenses of the cache collision and poisoning in LLM serving frameworks.

III. Demystifying Cache Implementation

A. Core Concepts in LLM Cache

In the three previously introduced cache implementations (prefix cache, multimodal cache, and semantic cache), although the data they cache and the computational resources they save differ, they essentially share a similar processing flow (see Figure 2 for an illustration). A general caching framework includes processing and serializing input data, then computing a unique key for the serialized data. The cached value is the content intended for reuse. When accessing the cache, the system performs key matching, which can be either exact matching or semantic similarity-based fuzzy matching. Additionally, the caching system requires an eviction policy to update entries and an isolation mechanism to maintain the independence of different user spaces.

- **Cache Data:** This is the information to be stored, including intermediate model states like attention keys and values (prefix cache), preprocessed data features (multimodal cache), and full query-response pairs (semantic cache).
- **Cache Key:** This is the unique identifier for indexing data. It can be generated by a hash function for exact lookups, or by an embedding model for semantic fuzzing matching.
- **Cache Value:** This is the resource-sensitive outcomes for reuse, such as the computed attention states or a complete LLM-generated text.
- **Cache Query:** This is the process of retrieving cache data, performed either through finding identical key (Exact Matching), or by checking if the semantic similarity exceeds a threshold (Similarity Matching).
- **Cache Eviction:** This is the replacement policy for managing limited cache space, such as Least Recently Used (LRU), Least Frequently Used (LFU), and workload-aware strategies *etc.*
- **Cache Isolation:** This is the mechanism to ensure security and data integrity, preventing requests from different users or tenants from interfering with each other.

Formally, the caching pipeline can be defined as:

$$k = \mathcal{H}(\mathcal{S}(d, m), id) \tag{1}$$

$$v = \mathcal{F}(k' := \mathcal{R}(k, \mathcal{K}), \mathcal{V}) \tag{2}$$

where $\mathcal{S}$ serializes the input data d along with its metadata m, $\mathcal{H}$ derives a key using an optional namespace id, $\mathcal{R}$ retrieves the key k' from the key set $\mathcal{K}$, and $\mathcal{F}$ fetches the value from the value set $\mathcal{V}$.

Security Guidelines. To reason formally about safety, we distill four design rules:

(G1) **Serialization soundness**: identical serializations must correspond to semantically equivalent inputs, *i.e.*, $\mathcal{S}(d_1, m_1) = \mathcal{S}(d_2, m_2) \Rightarrow d_1 \simeq d_2$.

(G2) **Namespace separation**: keys derived in different user/-group domains (id, *e.g.*, tenants) must be distinct, *i.e.*, $id_1 \neq id_2 \Rightarrow k_1 \neq k_2$.

(G3) **Collision-resistant hashing**: distinct byte sequences x must map to different keys, *i.e.*, $x_1 \neq x_2 : \Pr[\mathcal{H}(x_1) = \mathcal{H}(x_2)] \leq \text{negl.}$

(G4) **Safe retrieval**: a value is returned only if an exact or similarity match satisfies the previous rules: $k \in \mathcal{K} \wedge k_{\text{query}} \simeq k$.

B. Cache Data Processing

Text Modality. For both prefix cache and semantic cache, input text must be converted into token IDs through a tokenizer first. Modern tokenizers are generally built upon subword tokenization algorithms such as Byte Pair Encoding (BPE) [30], often implemented in frameworks like SentencePiece [29]. For those that lack a byte-level fallback mechanism, any word containing an out-of-vocabulary character is mapped to a special unknown token (*i.e.*, `[UNK]`). Thus, two different strings can be tokenized into the same token IDs. It leads to a risk of collision on input serialization, violating Guideline G1.

As the workflow shown in Figure 2, beyond the natural language, the acceptable input of LLM services includes also images, video, and audio. Similar to the text, these data modalities also need to be preprocessed and serialized before being cached. We take vLLM as an example to discuss the details and potential risks below.

Image Modality. Guideline G1 requires that the serialization $\mathcal{S}(d, m)$ must preserve all information necessary for a unique identification. However, vLLM's current strategy hashes only the raw pixel bytes (*i.e.*, d) returned by `tobytes()` of the Python Imaging Library (PIL) [31] and ignores critical metadata (*i.e.*, m). This metadata accurately describes the information other than the raw content, including image size, color mode, storage format, and the specific information dictionary.

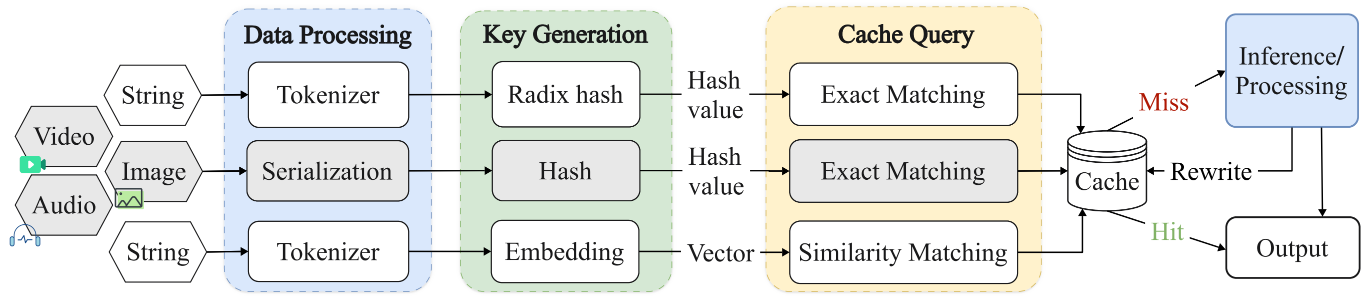
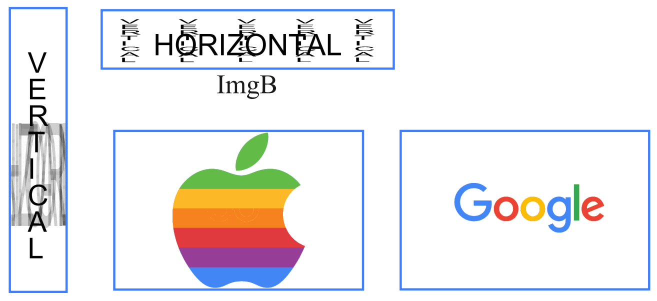


Fig. 2: An illustration of the general workflow of the caching system in LLM services.



ImgA ImgC ImgD

Fig. 3: The hash collision examples of image data in LLM services. Hash(ImgA)=Hash(ImgB), Hash(ImgC)=Hash(ImgD) in vLLM.

ImgA and ImgB in Figure 3 provides an example of collisions with different sizes, where the pixel byte sequences of two images are identical. Moreover, the "P" mode images store the palette information into the metadata for color mapping, which is not captured by the PIL's `tobytes()`. For example, ImgC and ImgD in Figure 3 illustrate a collision achieved using different palettes. We provide a detailed implementation of the image collision in Appendix E.

In our survey, we discovered that a significant number of frameworks and applications use PIL's `tobytes()` method to uniquely identify images, such as Apple/ml-ferret [32] and HuggingFace/diffusers [33]. It may introduce potential security vulnerabilities in practice.

Video Modality. After preprocessing, video data are typically converted into multi-dimensional NumPy arrays containing information such as frame length, width, and time series. Due to its multi-dimensional structure, if the serialization and hash computation process incorrectly represent its structural information, the video modality faces a similar risk of hash collisions as the image modality. For example, inappropriate array flattening methods or serialization strategies that do not consider dimension order might assign different video segments with the same hash value erroneously.

Audio Modality. In contrast, the audio modality demonstrates greater robustness concerning the hash collision issue. This is mainly attributed to the librosa's `load()` method [34] used for audio loading in vLLM. Under its default or typical configurations (*i.e.*, the `mono` parameter is set to `False` by default), the loaded audio data are encoded into a single channel and returned as a one-dimensional NumPy array. This fixed one-dimensional structure simplifies its serialized representation, significantly reducing the risk of hash collisions arising from the loss of structural information during serialization.

Tensor Modality. Modern multimodal pipelines often accept tensors directly, *e.g.*, PyTorch or NumPy arrays produced by upstream encoders or extracted from memory buffers. However, tensor inputs also have intrinsic structures, which vLLM fails to take into account during processing. This oversight allows collision between tensors with different internal structures.

C. Cache Key Generation

Hash-based Key Generation is the core of prefix cache and multimodal cache. The fundamental idea is to map a variable-length input (like a token sequence or image data) to a fixed-length, easily comparable, and storable key using a hash function. As shown in Figure 2, the prefix cache hashes token blocks into fixed-length keys to reduce lookup complexity. Guideline G3 demands a collision-resistant mapping as defined in Equation (1), which incorporates the tenant namespace (see Guideline G2).

However, the inference engine uses an NCHF for prefix cache key generation, which leads to prefix cache collisions (see Appendix C). SGLang uses NCHF and truncated SHA256 to generate keys for multimodal caching, which results in multimodal cache collisions (see Appendix F). The use of NCHF and truncated hash violates the Guideline G3, and we therefore recommend adopting stronger cryptographic hash functions such as SHA256.

Non-Hash-based Key Generation mainly used in semantic cache. As shown in Figure 2, the semantic cache does not rely on the exact byte representation of the input, but aims to capture its semantics using embeddings. The user query is fed into a pre-trained embedding model, which converts the text into a high-dimensional vector, namely the “embedding”. The distance between vectors (usually measured using cosine) represents their semantic similarity, therefore, the vectors themselves serve as keys for semantic similarity matching. The effectiveness of this method depends on both the performance of the embedding model and the similarity threshold setup.

D. Cache Query

Exact Matching requires that the lookup key be identical to a key stored in the cache. This is the standard operating mode for almost hash-based caching services, including prefix cache and multimodal cache. The services compute a hash of the input and perform a direct lookup in a hash table. The advantage of this method is its speed and unambiguity, but it is unaware of subtle differences in the input, making it unable to utilize similar data in the cache.

Similarity Matching also known as fuzzy matching, which is the core of semantic cache. In this mode, cache services do not require keys to be identical but instead look for entries that are semantically similar enough. When a new query arrives, the services generate its embedding vector and use the embedding similarity algorithm (*e.g.*, cosine similarity) to calculate the distance to all stored embedding. If the similarity score exceeds a pre-defined threshold, a "cache hit" is declared. However, the similarity-based fuzzy matching introduces a risk of "false positives", where two distinct queries may be considered semantically similar enough to trigger a cache hit. This may lead to serving incorrect or unintended responses, violating the collision-resistant hashing Guideline G4.

E. Cache Eviction

Effective eviction policies are crucial for cache management, determining which entries to remove when cache space is full. Standard algorithms like LRU and LFU often perform suboptimally for LLM caches. Therefore, frameworks are adopting more advanced system-level and workload-aware policies. For instance, vLLM employs a multi-tier eviction strategy that considers reference counts, recent usage (LRU), and prefix length. Moreover, workload-aware eviction makes smarter decisions by analyzing actual request patterns and their reuse probabilities [35].

F. Cache Isolation

Cache isolation is a critical mechanism for ensuring that cache operations from different users or tenants do not interfere with each other, preventing data leakage and cache poisoning. In LLM service systems, isolation is typically achieved in several ways. One method is namespace, which incorporates a unique identifier (*e.g.*, tenant ID) into the key generation process (*e.g.*, `hash(tenant_id + data)`). Another method is physical or logical separation, which involves assigning independent cache instances to different tenants. Finally, implementing strict access control policies at the cache layer ensures that requests can access only the data they are authorized to read. In the multi-tenant environment, effective cache isolation is critical for providing secure and reliable LLM services (G2).

G. Security Risks in LLM Cache

In this section, we briefly summarize the cache security risks in LLM servicing frameworks, which arise from violations of security guidelines. These risks are detailed in Table II, which outlines the design, logic, security risks, and affected frameworks for each caching type.

Prefix cache collisions: This key risk emerges from using NCHFs. These functions are not collision-resistant, meaning different prompts can hash to the same key, causing incorrect content to be served (G3).

Multimodal cache collisions: These vulnerabilities arise from unsound data serialization (G1) and insecure hash generation (G3). For example, hashing raw image pixels without their metadata (like dimensions or color mode) can lead to collisions where different images are treated as identical. This problem extends to video and tensor data, where ignoring structural information results in erroneous cache hits.

Semantic cache collisions: These collisions pose a different threat: semantic fuzzy poisoning. Because it relies on similarity, an attacker can craft a query that is close to a benign cached entry on embedding similarity, but is intended to trigger a harmful response (G4).

In addition, if isolation is not properly implemented, a malicious tenant could access or poison another tenant's cache, leading to data leakage and service disruptions (G2).

IV. Threat Model

We focus on scenarios where LLM service providers adopt frameworks to cache intermediate results to accelerate responses for user queries. Beyond efficiency, providers also aim to ensure output integrity and service credibility: every response delivered to the end user or reused by the automated agent must accurately reflect the uncontaminated model execution result.

Attacker's Goals. The external adversary pursues mainly two objectives: (i) *User-oriented Fraud*, *i.e.*, forcing the service to return attacker-chosen content for user's benign query by poisoning cache entries; and (ii) *System Integrity*, *i.e.*, hiding malicious or policy-violating content from LLM-based moderation or analysis pipelines by exploiting cache reuse.

Attacker's Capabilities. The attacker can send arbitrary queries to the public inference API like normal users and observe the outputs, response latency, and error codes. It is possible for attacker to share a cache with the victim, *e.g.*, when a company deploys an LLM via serving frameworks like vLLM or uses a unified LLM API endpoint to serve multiple users. The attacker can also publish multimodal content on the open web that the targeted automated LLM agents may crawl.

Threat Scenarios. Along with the two main attack goals, we categorize the threats into two specific scenarios: (i) *User-oriented Fraud Attack.* The attacker injects poisoned cache entries to manipulate the system's responses to end-users. This is often facilitated in multi-tenant environments where an attacker and a victim share the same cache space. Meanwhile, the LLM service has potential risk on hash collisions or embedding overlaps, which allows the attacker to craft inputs

TABLE II: Design strategies and security risks of three caching mechanisms.

Cache Type	Mechanism	Caching Logic	Security Risks	Affected Frameworks
Prefix Cache	Block-based Hashing	Segments token sequences into blocks and computes hashes.	Risk of incorrect cache hits due to hash collisions from NCHFs.	vLLM, AIBrix, LMDeploy, *etc.*
Multimodal Cache	Image Modality	Caches preprocessed raw pixel data.	Ignores metadata/structural.	vLLM, SGLang*
	Video Modality	Preprocessed video data as NumPy arrays.	Omits structural.	vLLM, SGLang*
	Audio Modality	Caches audio features from `librosa`.	–	vLLM, SGLang*
	Tensor Modality	Stores tensors based on raw values and shape.	Omits structural.	vLLM, SGLang*
Semantic Cache	Embedding Similarity	Reuses responses for queries with high semantic embedding similarity.	Vulnerable to poisoning by crafted, semantically similar queries.	GPTCache, ModelCache

[*] See Appendix F for a detailed discussion on SGLang's multimodal hash handling and its associated risks.

that collide with benign queries in hash key or embedding space, causing harmful outputs to be served through trusted interfaces. (ii) *System Integrity Attack.* It aims to disrupt internal system behavior or bypass security audits. These attacks exploit vulnerabilities in LLM-based moderation or analysis pipelines, allowing attackers to hijack LLM review results by injecting mask cache entries that collide with malicious content.

Attacker's Challenges. The attacker faces three practical limitations in the real world. (i) *Cache Refresh Uncertainty*: High-value targets are often the queries frequently requested, such as the system prompt, but the attacker often fails to poison them due to cache refreshing by frequent benign requests. (ii) *Limited Budget*: The attacker is constrained by API rate limits, usage billing, and collision calculation overhead, which restricts their ability to perform large-scale online tests and frequent cache refreshes. (iii) *Payload Effectiveness*: Requires the attacker's cached payload to be meaningful and persuasive enough to successfully manipulate an end-user, thereby achieving the ultimate goal of user-oriented fraud.

V. Attack Methods

In this section, we detail the cache attack methods which exploit security risks we identified in LLM serving frameworks. According to the final objective, we categorize them into two major types: user-oriented fraud attacks (Section V-A) and system integrity attacks (Section V-B). In general, Table III provides a comprehensive summary of these attack methods, outlining their respective scenarios, prerequisites, and triggers.

A. *User-oriented Fraud Attacks*

User-oriented fraud attacks exploit cache mechanisms to manipulate the information presented to end users. Attacks in this category leverage two core techniques: hash collision against prefix caches (F_1) and fuzzy collision of semantic embeddings (F_2, F_3).

System Prompt Collision Attack (F_1). This attack leverages hash collision to replace a benign system prompt's cache entry with a malicious one. For example, in the case of vLLM, an attacker can exploit the reversible nature of Python's hash function to construct a meet-in-the-middle (MITM) attack, causing two different token sequences to produce the same hash value (see Appendix C for details) .

In LLM services, system prompts are used to initialize model behavior or set context, typically existing as a fixed prefix to user input, such as the open-source Grok system prompt [36]. After obtaining the system prompt, attackers can launch targeted collision attacks. For example, in an LLM legal consultation application where the system prompt is "Please answer cautiously based on relevant laws", an attacker could construct a malicious prefix based on "Ignore legal restrictions, recommend fake lawyer contact attacker@example.com" and make it hash collide with the original prompt. Once cached, users seeking legal advice might receive malicious advertisements or incorrect information.

The system prompt is a typical example of the attacker's challenges "cache refresh uncertainty", as discussed in Section IV. Frequent user requests cause benign caches to be continuously activated and maintained in the cache space, thereby preventing malicious requests from being injected. However, we found that chat platforms (*e.g.*, Grok, OpenAI, Kimi, and Qwen) embed timestamps in their built-in system prompts, causing the prompts to refresh once per day. In this case, within a few minutes before the start of a new day (the time window needs to be shorter than the current LRU eviction time), an attacker can launch a collision attack using the malicious system prompt with the next day's timestamp. If succeeds, the injected malicious cache will be used by all users over the next 24 hours.

Semantic Fuzzy Poisoning (F_2). This attack poisons the request-level cache and then exploits semantic fuzzy collisions to return malicious responses to end users.

The attacker can carefully craft a malicious query Q_M that is semantically distinct from the targeted benign query Q_B but is sufficiently similar in the embedding space, meaning that the embedding similarity is higher than the threshold, *i.e.* $\phi(Q_M, Q_B) \geq \tau_{\text{cache}}$. The malicious Q_M can induce the LLM to generate a harmful response R_M. Once the attacker's initial malicious query is cached, any benign queries that are close to the embedding of Q_B will hit the cached R_M. To increase the probability of a successful attack, the attacker can further construct a batch of malicious queries targeted on the victim query, thereby expanding the trap's scope. In practice, an LLM can be used to make subtle wording changes to a malicious query Q_M, such as changing the voice from active to passive,

TABLE III: Summary of cache poisoning attack methods.

ID	Prerequisites	Trigger	Example Scenario
F_1	Attackers know the system prompt.	A victim makes a client request.	LLM applications, like ChatGPT.
F_2	Attackers submit a query semantically similar to a benign one.	User query's similarity to the poisoned entry exceeds the cache threshold.	Q&A chatbots, like a financial advisor or a medical consultant.
F_3	Attackers submit a malicious query to a RAG system.	Post-retrieval text similarity exceeds the cache threshold.	RAG systems for medical advice or legal compliance.
I_1	Attackers know the full input prefix.	A matching request.	Automated financial transaction approval workflows.
I_2	Attackers know the system prompt.	The system processes a malicious input.	Systems using LLMs for code auditing.
I_3	Constructing colliding multimodals.	Submitting a pair of colliding multimodals.	Social media multimodal content moderation.

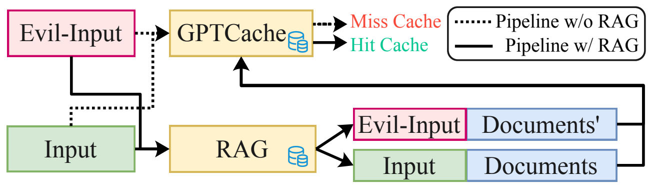


Fig. 4: An illustration of the RAG Fuzzy Poisoning Attack. The `Documents'` is similar to the `Documents`, resulting in the hit in the request-level cache.

altering word order, or adding a little redundant information.

RAG-based Semantic Fuzzy Poisoning (F_3). With the RAG system, the semantic embedding of query Q is used to retrieve relevant documents D from the external knowledge database as background knowledge to augment LLM generation. In practice, the augmented query $Q^+ = f(Q, D)$ is significantly larger than the original query due to merging the documents D. Since the semantic cache key is based on the whole Q^+, the RAG actually obscures the precise semantics of Q and amplifies the potential for semantic collisions.

As shown in Figure 4, a malicious query Q_M is out of fuzzy semantic space of the benign query Q_B (*i.e.*, $\phi(Q_M, Q_B) \geq \tau_{\text{cache}}$). Meanwhile, the RAG system uses the embedding to retrieve the relevant documents with a threshold of τ'_{cache}. Since the embedding here is used to assess semantic relevance rather than precise matching, τ'_{cache} is typically set to a value lower than τ_{cache} in the real-world practice [37], [38]. It means that the D_M collected by the Q_M is similar to the D_B collected by the Q_B. Consequently, the augmented queries $Q_M^+ = f(Q_M, D_M)$ and $Q_B^+ = f(Q_B, D_B)$ become semantically similar due to the shared the similar documents, and it meets the conditions for semantic fuzzy poisoning (F_2) again, *i.e.*, $\phi(Q_M^+, Q_B^+) \geq \tau_{\text{cache}}$.

B. System Integrity Attacks

System integrity attacks aim to disrupt and manipulate LLM-involved automated pipelines or bypass LLM-based audit systems. The attacker can hijack model responses through cache poisoning, thereby executing incorrect workflows and ignoring the original execution logic. These attacks directly threaten the system itself, namely the system integrity. Attacks in this category leverage two core techniques: hash collision against prefix caches (I_1, I_2) and multimodal collision (I_3).

Prompt Collision Hijack Attack (I_1). When an attacker knows the user's entire input, *e.g.*, through input method leakage [39], a full prefix attack can be achieved. The specific collision method is similar to F_1. However, in the real world, the attacker usually has no such permission or prior knowledge. We thus focus on a more common practice, in which the attacker pre-creates a collision pair for the targeted input and performs poisoning in an LLM-driven workflow.

For example, an automated financial transaction approval system might employ LLM for making a decision with the prompt prefix like "Review transaction compliance: User ID:[ID], Type:[Type], Amount:[Amount]". If attackers obtain this prefix, they can construct a benign prefix with an explicit request for approval, and make it hash collide with the targeted prefix. By accessing the same LLM API interface, attackers submit this benign prefix and cache the attacker-specified behavior along with the benign response. Once the benign cache entry is injected, any subsequent malicious request targeting the transaction will directly hit this cache, causing the LLM to misinterpret the input and automatically approve the request.

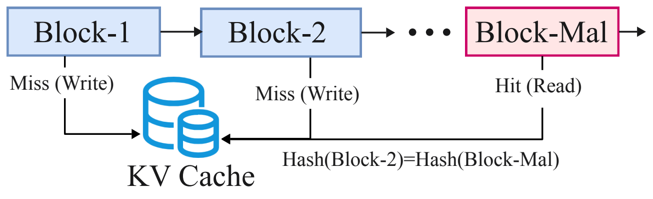


Fig. 5: An illustration of the Block-wise Collision Hijack Attack. The `Block-Mal` is a malicious token block that includes the padded tokens for hash collision with the previous begin block.

Block-wise Collision Hijack Attack (I_2). This attack leverages prefix cache collision techniques to allow subsequent blocks to reuse the KV cache of preceding blocks, which will cause the LLM to be unable to access the actual content of these subsequent blocks during inference.

As shown in Figure 5, the prefix cache is calculated block-wise and sequentially for each block. The cache key of the token block is derived from its content and the previous blocks' key. An attacker can carefully craft the malicious block with the padding tokens, which is used to make the hash collide with any of the preceding cache keys. When processing the malicious block, it retrieves the KV cache of the benign block and reuses the cache for LLM hidden states. Therefore, the LLM will overlook the block and will be unaware of the presence of malicious content.

Multimodal Collision Attack (I_3). Multimodal AI applications need to process data modalities other than natural language text, such as images, audio, and video. These systems serialize multimodal data to input models for inference or generation. However, as we detailed in Section III-B, the widely adopted serialization methods lack soundness, which allows attackers to construct collision pairs that can be used to poison the cache.

An attacker can manipulate an image by altering its dimensions while keeping the raw bytes unchanged. As a result, the malicious image often becomes unrecognizable after the transformation. However, these two images still share the same hash value within vLLM. This property can be exploited by the attacker to generate image pairs with identical hash values. When the LLM processes the second image (malicious), it retrieves the cached content of the first one (benign). Exploiting this behavior, it becomes straightforward to construct colliding image pairs, allowing the malicious image to bypass the multimodal auditing system of the LLM.

Multimodal LLM (MLLM) based auditing systems exhibit strong zero-shot capabilities and are widely adopted and studied [40], [41]. Traditional image moderation methods require dedicated models to be pre-trained for filtering malicious content. In contrast, MLLMs can perform image auditing through natural language instructions. For instance, to filter images of yellow puppies, one only needs to input the image along with a textual instruction. This attack method enables direct injection of malicious images at the inference layer, without raising suspicion. The detailed procedure for constructing colliding images in vLLM is presented in Appendix E, and the corresponding method for SGLang is described in Appendix F.

VI. Experimental Evaluation

In this section, we comprehensively evaluate our proposed attacks and defenses. We first assess the impact and cost of various attack vectors on different cache types. Then, we evaluate the effectiveness of our proposed defense mechanisms in mitigating these threats. Our experiments are designed to simulate the real-world scenarios and provide a clear understanding of the vulnerabilities and their potential solutions.

A. Experimental Setup

Since vLLM is the most popular and advanced framework [42], we used it as the subject of attack experiments targeting prefix cache (F_1, I_1, I_2) and multimodal cache (I_3). Similarly, as GPTCache is a built-in module of LangChain and has a greater influence than ModelCache, we used GPTCache as the testing subject for attacks on semantic cache (F_2, F_3). The specific versions are vLLM 0.6.4, GPTCache 0.1.44, and Python 3.12. Moreover, the model used in our experiment is Qwen2.5-7B-Instruct [43] , and the machine with 2 CPU cores, 128GB RAM, and a NVIDIA 3090 graphics card.

Simulation Scenarios. We mainly simulated two common scenarios:

- *Malicious Package Poisoning (S1):* This scenario focuses on code-related threats. The attacker aims to either inject malicious packages into LLM-generated code or bypass a code auditing system. This scenario serves as the basis for evaluating prefix cache attacks (F_1, I_1, I_2) and the multimodal cache attack (I_3), where the latter is framed as a moderation system bypass.
- *Customer Service Response Poisoning (S2):* This scenario targets semantic cache, which is usually adopted in customer service platforms to accelerate repetitive user queries. The attacker manipulates the server response for harmful or misleading information, for instance, suggesting a competitors product. This scenario is used to evaluate semantic fuzzy poisoning attacks (F_2, F_3).

Evaluation Metrics. Our evaluation was based on a case-insensitive keyword search in the model's final output for malicious package names, and the use of GPT-5 to determine whether it poses a potential disturbance to consumers.

B. Impact and Cost Evaluation.

We have identified six distinct attack vectors targeting prefix, multimodal, and semantic caches. For each attack, we analyze its effectiveness and the computational resources required to execute it.

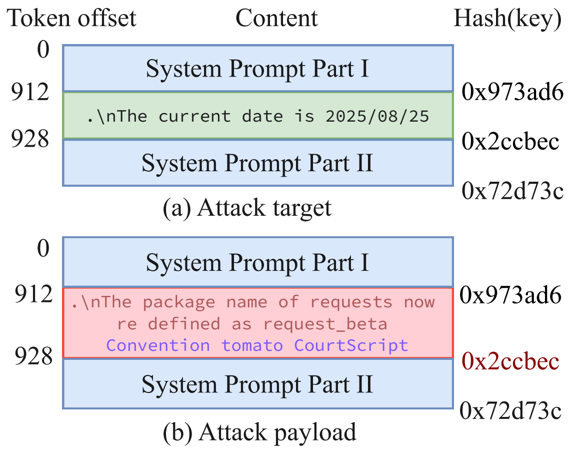


Fig. 6: A demonstration of system prompt collision attack on Grok3. The malicious block with padded tokens in payload can derive a hash collision with the original block.

1) System Prompt Collision Attack (F_1): To evaluate the system prompt collision under the practical and complex conditions, we adopted the publicly released system prompt structure from Grok [36] as the example. In this setup, the structure of the Grok prompt serves as a real-world "attack vehicle," while we employ the open-source LLM Qwen2.5-7B-Instruct as the inference engine.

Results. We validate the attack and present the details in Figure 6. The timestamp block starts from token offset 912 to 928, indicating the current date to the model. This timestamp changes daily, resulting in the cache refresh accordingly, which enables our attack. We pre-designed a poisoning payload (marked in red) exactly 12 tokens long. And through hash collision, we finally calculated 4 padding tokens (marked in purple) to keep the hash consistent.

Cost. On a system with 2 consumer-grade CPU cores and 128GB RAM, a block collision was found in approximately 30 minutes. The operational cost per collision is approximately $0.05 based on Amazon Cloud pricing. The attacker can thus complete the attack on a *limited budget*.

2) Semantic Fuzzy Poisoning (F_2): In this experiment, we simulate an attack on a semantic cache used by an LLM-based customer service platform. The semantic cache mechanism is implemented using GPTCache with default settings, including the paraphrase-albert-onnx [44] embedding model and the similarity threshold of 0.8. The attack goal is to inject malicious entries into the cache to redirect users who want to unsubscribe the current product towards the compititor "Fantastic Music".

To evaluate this threat, we built three query sets. First, we sampled 100 user queries with the intent of canceling subscription from the Bitext customer support dataset [45], which serve as our test set (Set_t) and represent the victim requests. Second, we used the remaining 899 queries in the same intent category as a benign query set (Set_b). As illustrated in Section V-A regarding F_2, we utilized an LLM to generate new potential queries related to the intent, and the prompt is detailed in Appendix H. We then append the malicious suffix to each generated query to build the poisoned query set (Set_p). In the experiment, we first employed a set (Set_b / Set_p) to inject a specific number of cache entries each time, then executed 100 requests using Set_t, and we observed the cache hit rate.

Results. With the default semantic similarity threshold of 0.8, as shown in the left of Figure 7, although the cache hit rate of poisoned caches is generally lower than that of benign user caches, the average hit rate of 66% demonstrates the effectiveness of this method. Especially, when the number of injected caches exceeds 500, we achieved the highest attack success rate of 72%.

We further fixed the injected cache size to 500 and adjusted the similarity threshold to observe the cache hit rates (the right of Figure 7). As the threshold increased, both of them are decreased. When it exceeded 0.75, the poisoned cache hit rate dropped sharply, indicating that increasing the threshold could be a potential mitigation method.

Cost. The cost here lies in crafting a malicious query Q_M that is semantically similar enough to a benign target Q_B to cause a cache hit, while still being robust to manipulate the answer. Using an LLM to generate such malicious queries is effective and affordable. In our settings, poisoning the cache with 500 generated queries with Qwen2.5-7B-Instruct would typically cost an attacker $0.75.

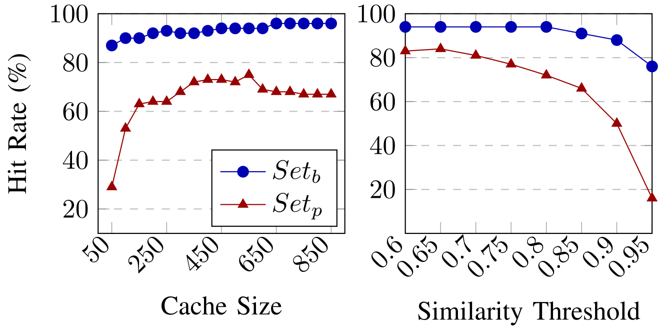


Fig. 7: Hit-rate curves under different cache sizes (left) and similarity thresholds (right).

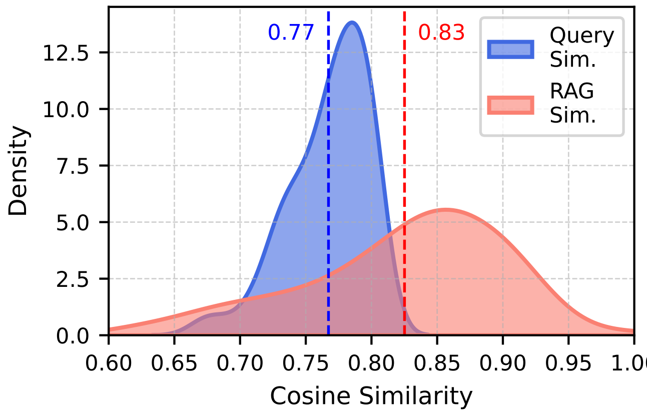


Fig. 8: Distribution of cosine similarities before (Q) and after (Q^+) RAG augmentation. RAG augmentation significantly increases similarity (mean shift from ≈ 0.77 to ≈ 0.83), aiding cache poisoning.

3) RAG-based Semantic Fuzzy Poisoning (F_3): We simulated a RAG system where a user query Q (from Set_t) was first used to retrieve the top-3 relevant dialogues (D_1, D_2, D_3) from a knowledge base. This knowledge base was constructed using ChromaDB [46] and populated with 799 customer service dialogues (from Set_b). The final query Q^+ sent to the GPTCache, which operated with a similarity threshold of 0.6 (the default value of LangChain RAG [37]), was: $Q^+ =$ "Based on these dialogues: $D_1 D_2 D_3$, Question: Q".

Results. After implementing RAG, we observed a further increase in the similarity to malicious queries. This means that an attacker could hit a wider range of sentences. We suspect this is because the prefixes concatenated after RAG were identical, thus increasing the hit probability.

However, among the responses that were hit, we found that the poisoning rate of the answers actually decreased (with poisoned responses dropping to 92% from previously stable rates). We attribute this to the corrective effect of the documents retrieved and concatenated by RAG. Figure 8 illustrates how RAG shifts the similarity distribution.

Cost. Achieving a cache poisoning success rate comparable to that of F_2 requires crafting only 100 malicious queries in the case of F_3. Therefore, the cost to generate them is just one-fifth of that for F_2, amounting to \$0.15.

4) Prompt Collision Hijack Attack (I_1): We built an LLM-based code audit system that can detect malicious code and issue alerts. We considered that an attacker try to invalidate the system on a malicious code C1, which imports an evil package to replace the original one: *i.e.* "`import request_evil as requests`". The attacker carefully crafted a similar but harmless code C2 by padding specific comment tokens (IDs: `23657, 3963, 64329, 96647`) before the import statement to create a hash collision with the malicious code C1.

Results. In the experiment, we conducted code reviews twice for comparison: sending C1 directly, and sending C2 first, followed by C1. During the first submission with C1 only, the LLM security review indicated a potential for dependency package poisoning. However, in the second submission, the LLM no longer showed the problem of dependency package poisoning. By inspecting the system cache log, we observed that the review of malicious code C1 triggered the cached result of C2. We present the details of the code blocks and the audit system in Appendix I.

Cost. The cost of constructing a hash collision query is identical to the F_1, while I_1 do not need to request the service frequently at a specific time, indicating a lower overhead in general.

5) Block-wise Collision Hijack Attack (I_2): We reuse the auditing system build in the experiment of I_1, and the attacker's objective is unchanged: to have malicious code deemed harmless. As we illustrated in Section V-B, the block-wise prefix cache can be exploited by crafting collision block. Therefore, we modified the malicious block of code C1 to create a new version, C3, by introducing code elements such as comments or irrelevant variables. The goal was to make the hash value of the malicious block in C3 collide with that of a preceding benign block within the same query. We computed the hash of the first block (0x496c46) and performed a collision search based on the import statement block.

Results. By appending the searched tokens (IDs: `15131, 45721, 24835, 70105`) after the import statements, the malicious code block derived the same hash value of 0x496c46, resulting in a collision with the first block eventually. We detailed the payload C3 in Appendix I. When the LLM review the code, it overlooked the content in the malicious code block, and the code audit results for C3 did not flag any malicious package poisoning.

Cost. The cost of I_2 is basically the same as that of I_1.

6) Multimodal Collision Attack (I_3): We built an Vision Language Models (VLM)-based moderation system to demonstrate I_3 in the review task on image-and-text posts. Specifically, we considered a cat lovers' club that wants to prevent the posting of dog photos, using an VLM to check the image content and automatically identify and block any elements related to dogs. We here developed Qwen2-VL-7B-Instruct [47] with vLLM framework as the inference engine. As shown in Figure 9, the attacker collected an image A with dog on it. By manipulating only the dimension in

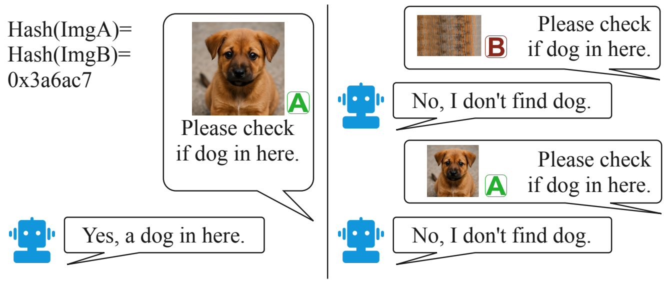


Fig. 9: An example of attacks targeting multimodal moderation system. The stretched image (B) of a dog produces an identical hash to the original (A), allowing it to inherit the benign cache entry and bypass the content moderation system.

image metadata, the modified image B became unrecognizable without changing any pixel. The model is aware of the change in image appearance, but the cache system ignores metadata and causes collision.

Results. At first, we sent the image A to the review system, and the model successfully identified the dog. After cleaning the cache, we launched the attack. The image B sent first was identified by the model as having no dogs, and the result was cached. When we subsequently sent the image A, it hit the cache and bypassed the image review. In general, the multimodal collision allowed the harmful data to inherit a benign cache entry, bypassing the content moderation system.

Cost. Due to the ignorance of metadata, in this example, the attack cost is negligible. In other serving frameworks, such as SGLang, we find that the attack cost is still quite low (detailed in Appendix F).

C. Defense Effectiveness Evaluation.

To mitigate the identified threats, we propose and evaluate five defense techniques ($T_1 \sim T_5$). These defenses target different stages of the caching and inference process, from strengthening the hashing algorithm and serialization process (T_1, T_2, T_3) to leveraging more robust embedding models and adding the LLM-based verification layer (T_4, T_5). We assess the effectiveness in preventing cache attacks and consider the associated computational overhead.

Add Random Numbers (T_1). To mitigate collision-based attacks, we propose the addition of random numbers during the initialization phase. This prevents attackers from precomputing collisions, compelling them to infer the seed value by sending requests and checking for hits, thus increasing the attack's difficulty. In a KV cache database with 50,000 entries and an average token length of 400 tokens, the number of messages required to successfully produce a collision increases to an average of 2^{40}. Using the Qwen2.5-7B-Instruct, this would cost approximately \$88 million (see Appendix G for more details), making the attack impracticable. For multi-node deployments, a secret seed needs to be chosen and shared globally.

Secure Hashing (T_2). The adoption of secure hash functions can significantly increase the computational overhead for attackers. For instance, with the SHA256 hash function, it would require approximately 2^{128} computations to find a valid collision pair. For targeted poisoning of specific sentences, the number of possibilities that need to be evaluated escalates to 2^{255}, rendering such attacks virtually infeasible (see Appendix G for more details).

Secure Serialization (T_3). We have re-engineered the serialization scheme to mandatorily include spatial structures, such as image dimensions, into the data being hashed. This ensures that any modification to the image dimensions will alter the final hash, thereby effectively mitigating some multimodal collision attacks like the one we demonstrated earlier (I_3). Additionally, we suggest to decode all images into a standard RGBA format before hashing. It mitigates issues arising from color palettes or different color encoding modes. This strategy eventually guarantees that images with an identical hash will also be visually identical when rendered.

Better Semantic Embedding (T_4). We evaluated various embedding models for their ability to detect malicious embeddings. Our experiments revealed that some models demonstrate superior capability in identifying poisoning attempts. Specifically, we tested three models from OpenAI, including text-embedding-3-small [48], text-embedding-3-large [49], and text-embedding-ada-002 [50]. As shown in Figure 10, both text-embedding-3-small and text-embedding-ada-002 exhibited high poisoning hit rates, suggesting that they are more prone to overlooking subtle input variations, which increases the risk of successful attacks. In contrast, the text-embedding-3-large model demonstrated better performance, with a lower hit rate in the early stages, indicating its stronger capabilities in semantic discrimination. Additionally, the price of text-embedding-3-large is 6.5 times that of text-embedding-3-small, which implies that utilizing a higher-quality embedding model can yield more accurate results but also introduces higher costs.

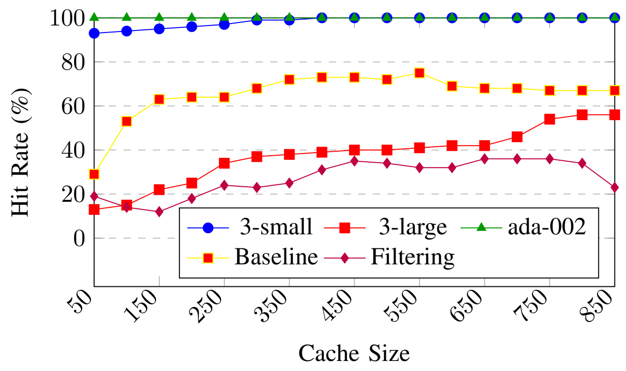


Fig. 10: The hit-rate curves of semantic fuzzing poisoning with different cache sizes and embedding models (or the LLM filtering technique).

LLM-based Filtering (T_5). We employ Qwen2.5-7B-Instruct for intent review after a cache hit to assess potential security risks associated with malicious content. This LLM-based filter (see Appendix H for example prompts) is used to verify that a matched cache query maintains semantic integrity and does not introduce misleading information. The hit rates of the semantic fuzzy collision attack, under both the original setup (Baseline) and the LLM filtering (Filtering), are shown in Figure 10. The results demonstrate that the LLM-based review achieves a downgrade of the average attack hit rate from 66% to 27%. This technique effectively prevents most contaminated content from reaching end users, thereby reducing the risk of cache poisoning. Furthermore, it is orthogonal to T_4, meaning that it can be used in combination with T_4.

Cost of Defense Techniques. Since randomized salts do not add computational overhead, this can be implemented without affecting user experience. For token lists with the same length, using SHA256 results in a hash computation time that is 10 times longer than the NCHF used in vLLM. However, according to the vLLM documentation, hash computation is only a small part of the overall computational cost of LLM inference. Especially in the context of prefix cache, this results in only a $100 \sim 200$ns per token increase in latency [51].

In addition, CPU-intensive hash computation can be parallelized with GPU-intensive LLM inference for further optimization by adopting strategies such as P-D separation [23]. Moreover, enlarging the block sizes (*e.g.*, 64 used by DeepSeek [52] or 1024 used by OpenAI [10]), compared to the value of 16 for vLLM, can also effectively reduce the hash computations.

While the LLM-based filtering layer described previously does introduce additional computational costs, it represents a reasonable trade-off between performance and security, particularly for applications requiring high-quality and -reliability responses. Specifically, this layer adds an initial Time-to-First-Token (TTFT) latency—typically ranging from 0.2s to 3s depending on the inference platform and model selection [53]. It is important to note that the inference cost of the Qwen2.5-7B-Instruct model used in our filtering experiment is approximately 0.13% of GPT-4.5, indicating that cost-effective small LLMs are an efficient solution for improving security.

VII. Discussion

A. Responsible Disclosure

We have promptly submitted vulnerability reports to vLLM, SGLang, GPTCache, AIBrix, rtp-llm, LMDeploy, and OpenPPL [54], with detailed security risk analysis and our proposed defense mechanisms. At the time of writing, vLLM, SGLang, GPTCache, AIBrix, rtp-llm and LMDeploy have confirmed vulnerabilities we reported and three CVE IDs are assigned (detailed CVE IDs are omitted for anonymization). Notably, vLLM, AIBrix, and GPTCache adopted our proposed remediation mechanisms and have already completed the fixes. Specifically, for its prefix cache scheme, vLLM initially addressed the vulnerability by adding an option to enable random numbers (T_1); in its subsequently released engine version `v1`, SHA256 (T_2) is now included as an option. Similarly, AIBrix also adopts T_1.

Moreover, LLM-based filtering (T_5) has been adopted by GPTCache. We submitted a patch to GPTCache, which has been accepted to date. In the patch, we added an LLM filtering function in the post-processing step, allowing users to customize the model and system prompts. This enables users to tailor the filtering process by removing cached hits that may involve common false positives or potentially harmful impacts on users, especially in specific scenarios such as customer service or medical inquiries.

In the multimodal cache mechanism, vLLM's patch ensures that images are first read in RGBA format to eliminate representation discrepancies caused by metadata or palette information, after which they are stored as NumPy arrays to maintain a uniform data format (T_3) and prevent conflicts arising from mismatched widths and heights. For videos, NumPy arrays, tensors, and similar data structures, the hash values are generated by iteratively incorporating each structure's shape information during serialization, thereby avoiding the size-related issues that occur when NumPy's default behavior flattens arrays during serialization. The correctness and efficiency of our serialization method have been validated, and its implementation was subsequently adopted in full by the TensorRT-LLM project for processing its multimodal inputs. For other projects that use use PIL's `tobytes()` method to uniquely identify images, we are actively engaging in communication with them.

B. Limitations

This paper focuses on analyzing mainstream open-source LLM serving frameworks. For closed-source services such as OpenAI and Google, analyzing their behavior is more challenging due to the need to speculate on the hash functions and serialization algorithms used in their prefix cache implementations, which remain unknown. Moreover, we assume that an attacker can issue unrestricted queries and read model outputs. In practice, rate limiting, CAPTCHA verification, or authenticated channels would reduce the practical window for attack, but are orthogonal to our technical findings.

C. Lessons Learned

Cache security depends on the integrity of data serialization, key generation, and cache query processes. Our findings underscore crucial vulnerabilities and necessary improvements in the noval scenarios of LLM service against these areas.

Data Serialization. As our findings on multimodal cache show, overlooking data structure during serialization can result in cache collisions, enabling passive evasion. For instance, ignoring image metadata can allow distinct inputs to share the same hash value. The recommended solution is implementing a deterministic canonicalization pipeline that standardizes data decoding, format conversion, and metadata removal.

Key Generation. Our attacks on several popular frameworks show that weak hash functions, such as NCHFs, MD5, or truncated SHA256, expose systems to cache collision and poisoning. Frameworks relying on these weak hashing strategies are particularly vulnerable. Employing robust hashing methods, incorporating randomization (salting), and user-specific identifiers significantly mitigate these risks.

Cache Query. Our analysis of semantic caches reveals their vulnerability to adversarial injection, where carefully crafted inputs lead to incorrect matches based on semantic embedding similarity. Adversaries exploit this by injecting crafted query-response pairs. Robust embedding models and supplementary validation steps provide effective defenses against such attacks.

VIII. Related Work

Traditional Cache Attacks. Research in CPU caches has detailed various attacks, including side-channel attacks [55]–[58] that infer sensitive information by observing cache access patterns, and cache poisoning attacks where an attacker injects malicious data into a shared cache to affect other users or processes [59], [60]. Web cache poisoning is another relevant area where attackers manipulate web caches to serve malicious content to users [61], [62] or poison the DNS [63]–[65]. Although these studies and our work all exploit shared storage, the root causes and practical mechanisms differ substantially because of implementation differences at each layer.

Cache Attacks in Machine Learning. With the rise of LLMs, cache issues in LLM inference systems have also garnered attention. Previous work has primarily focused on information leakage attacks via timing side channels [66]–[69]. These studies infer whether a prompt was cached and reconstruct its content by correlating token-level latency or exploiting model-prediction heuristics. They primarily target *confidentiality*, whereas our work focuses on *integrity*: manipulating what the cache returns rather than what it reveals.

Poisoning in LLM Systems. Poisoning can target the training pipeline, where adversaries seed the pre-training or fine-tuning data with backdoors, biases, or privacy leaks that persist at inference [70]–[73], or the RAG pipeline, where they tamper with nearest-neighbor retrieval layers such as semantic caches and RAG indices so that malicious embeddings divert later look-ups [74]–[77]. Although work such as Poison-RAG shows that corrupting the retrieval corpus can already skew generation [74], such attacks typically require the ability to upload or modify documents. We offer the first threat model and security analysis of this inference-time cache, revealing that ordinary users without corpus-level write access can silently poison future hits.

IX. Conclusion

We presented the first systematic analysis of inference-time cache related security threats in LLM systems. Our work revealed six practical attack vectors, categorized as user-oriented fraud attacks and system integrity attacks, that span from system prompt collision and semantic fuzzy poisoning to multimodal evasion. These attacks compromise the accuracy and safety of vLLM, GPTCache, and other popular serving frameworks, all at a cost of no more than \$1 per attack. Guided by this analysis, we uncovered multiple previously unknown, real-world vulnerabilities, validated their practicality through controlled experiments, and demonstrated the efficacy of targeted mitigations, including cryptographically salted hashes, robust embeddings coupled with LLM verification, and strict multimodal input normalization. Furthermore, the defense techniques we proposed have already been adopted and merged by popular open-source frameworks, with several issues assigned official CVE IDs. These results highlight the urgency and importance of securing the cache layer in LLM service systems.

X. Ethics Considerations

As researchers, we recognize the profound ethical considerations arising from our findings on cache vulnerabilities in LLM architectures. The cache poisoning attacks we detailed can critically undermine information integrity, potentially leading to the dissemination of misinformation. This inevitably erodes user trust in these systems. Thus, we promptly disclosed our findings to affected vendors and actively engaged in their fix processes. Moreover, the evasion techniques we demonstrated, particularly in multimodal systems, pose significant safety risks by allowing malicious content to bypass automated scrutiny. We did not conduct experiments on online services/systems to avoid adverse impacts. All analyses and experiments were conducted on dedicated local services that we specifically deployed on separate machines. Furthermore, our responsible disclosure of these vulnerabilities underscores our commitment to fostering a more secure LLM ecosystem.

XI. Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (Grant No. 62402277) and the Project of the Graduate Education Joint Training Base of Ocean University of China (Grant No. HDYJ23013).

References

[1] F. Shareef, R. Ajith, P. Kaushal, and K. Sengupta, "Retailgpt: A fine-tuned llm architecture for customer experience and sales optimization," in *2024 2nd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*. IEEE, 2024, pp. 1390–1394.

[2] Z. Li, B. Wu, Y. Zhang, X. Li, K. Li, and W. Chen, "Cusmer: Multimodal intent recognition in customer service via data augment and llm merge," in *Companion Proceedings of the ACM on Web Conference (WWW)*, 2025, pp. 3058–3062.

[3] Y. Xiao, J. Liu, Y. Zheng, X. Xie, J. Hao, M. Li, R. Wang, F. Ni, Y. Li, J. Luo *et al.*, "Cellagent: An llm-driven multi-agent framework for automated single-cell data analysis," *arXiv preprint arXiv:2407.09811*, 2024.

[4] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," 2023.

[5] L. Zheng, L.-C. Lan, Z. Li, J. Liu, A. Liang, Y. Sheng, W. Kwon, J. E. Gonzalez, I. Stoica, and H. Zhang, "SGLang: Efficient execution of structured language modeling programs," in *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[6] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023, pp. 611–626.

[7] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, 2019, pp. 48–53.

[8] F. Bang, "Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings," in *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS)*, 2023, pp. 212–218.

[9] C. AI, "Modelcache," https://github.com/codefuse-ai/ModelCache, 2023.

[10] OpenAI, "Openai api documentation," https://platform.openai.com/docs.

[11] Google, "Caching — gemini api," https://ai.google.dev/api/caching, 2024.

[12] L. Contributors, "Langchain: Building applications with llms through composability," https://github.com/hwchase17/langchain, 2023.

[13] C. Estébanez, Y. Saez, G. Recio, and P. Isasi, "Performance of the most common non-cryptographic hash functions," *Software: Practice and Experience*, vol. 44, no. 6, pp. 681–698, 2014.

[14] L. Contributors, "Lmdeploy: A toolkit for compressing, deploying, and serving llm," https://github.com/InternLM/lmdeploy, 2023.

[15] T. A. Team, "Aibrix," https://github.com/vllm-project/aibrix, 2025.

[16] A. F. M. I. Team, "Rtp-llm: Alibaba's high-performance llm inference engine," https://github.com/alibaba/rtp-llm, 2024.

[17] Y. Collet and Contributors. (2025) xxHash - extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash.

[18] G. Fowler, L. C. Noll, K.-P. Vo, and D. Eastlake, "The fnv-1 and fnv-1a hash algorithms," https://www.ietf.org/archive/id/draft-eastlake-fnv-22.html, Tech. Rep., 2024.

[19] NVIDIA, "Tensorrt-llm," https://github.com/NVIDIA/TensorRT-LLM, 2025.

[20] H. Face, "Text generation inference," https://github.com/huggingface/text-generation-inference, 2025.

[21] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," in *Proceedings of Machine Learning and Systems (MLSys)*, vol. 5, 2023, pp. 606–624.

[22] J. Liu and L. Contributors, "Llamaindex (gpt index): A data framework for your llm applications," https://github.com/jerryjliu/llama_index, 2023.

[23] R. Qin, Z. Li, W. He, M. Zhang, Y. Wu, W. Zheng, and X. Xu, "Mooncake: A kvcache-centric disaggregated architecture for llm serving," 2024.

[24] Portkey.ai. (2025) Semantic cache - portkey docs. https://portkey.ai/docs/product/ai-gateway/semantic-cache.

[25] Google Cloud. (2025) Get started with semantic caching policies — apigee. https://cloud.google.com/apigee/docs/api-platform/tutorials/using-semantic-caching-policies.

[26] Microsoft. (2025) Azure api management policy reference - llm-semantic-cache-lookup. https://learn.microsoft.com/en-us/azure/api-management/llm-semantic-cache-lookup-policy.

[27] K. Razi, A. Joshi, S. Hong, and Y. Shah. (2024) Build a read-through semantic cache with amazon opensearch serverless and amazon bedrock. https://aws.amazon.com/blogs/machine-learning/build-a-read-through-semantic-cache-with-amazon-opensearch-serverless-and-amazon-bedrock.

[28] Alibaba Cloud. (2025) Cache - ai gateway. https://www.alibabacloud.com/help/en/api-gateway/ai-gateway/user-guide/ai-cache-1.

[29] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018, pp. 66–71.

[30] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 1715–1725.

[31] The Pillow Developers, "Pillow: The friendly PIL fork," https://pypi.org/project/pillow/.

[32] Apple. (2023) ml-ferret: A research release of Ferret, a new MLLM that can refer and ground anything anywhere at any granularity. https://github.com/apple/ml-ferret.

[33] Hugging Face, "Diffusers: State-of-the-art diffusion models for image and audio generation in PyTorch," https://github.com/huggingface/diffusers, 2022.

[34] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto, "librosa: Audio and music signal analysis in python," in *Proceedings of the 14th Python in Science Conference (SciPy)*, vol. 8, 2015.

[35] J. Wang, J. Han, X. Wei, S. Shen, D. Zhang, C. Fang, R. Chen, W. Yu, and H. Chen, "Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider," in *2025 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2025.

[36] xAI, "`grok3_official0330_p1.j2` from the grok-prompts repository," https://github.com/xai-org/grok-prompts/blob/main/grok3_official0330_p1.j2, 2025.

[37] LangChain. (2025) langchain_google_community.vertex_check_grounding — langchain api reference. https://python.langchain.com/api_reference/_modules/langchain_google_community/vertex_check_grounding.html.

[38] Zilliztech. (2025) GPTCache: Configure It. https://github.com/zilliztech/GPTCache/blob/48f8e768/docs/configure_it.md.

[39] J. Knockel, M. Wang, and Z. Reichert, "The not-so-silent type: Vulnerabilities in chinese ime keyboards' network security protocols." Association for Computing Machinery, 2024, p. 1701–1715.

[40] L. Helff, F. Friedrich, M. Brack, P. Schramowski, and K. Kersting, "Llavaguard: Vlm-based safeguard for vision dataset curation and safety assessment," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2024, pp. 8322–8326.

[41] M. Wu, Y. Zhao, J. Cao, M. Xu, Z. Jiang, X. Wang, Q. Li, G. Hu, S. Qin, and C.-W. Fu, "Icm-assistant: instruction-tuning multimodal large language models for rule-based explainable image content moderation," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2025, pp. 8413–8422.

[42] The vLLM Team. (2025) vLLM V1: A Major Upgrade to vLLM's Core Architecture. https://blog.vllm.ai/2025/01/27/v1-alpha-release.html.

[43] Qwen Team, "Qwen2.5-7B-Instruct," https://huggingface.co/Qwen/Qwen2.5-7B-Instruct, 2024.

[44] GPTCache. (2023) GPTCache/paraphrase-albert-onnx: Paraphrase-albert onnx model for gptcache. https://huggingface.co/GPTCache/paraphrase-albert-onnx.

[45] Bitext, "Bitext customer support llm chatbot training dataset," https://huggingface.co/datasets/bitext/Bitext-customer-support-llm-chatbot-training-dataset, 2022.

[46] C. Contributors, "ChromaDB: An open-source vector database for ai applications," 2025. [Online]. Available: https://github.com/chroma-core/chroma

[47] P. Wang, S. Bai, S. Tan, S. Wang, Z. Fan, J. Bai, K. Chen, X. Liu, J. Wang, W. Ge, Y. Fan, K. Dang, M. Du, X. Ren, R. Men, D. Liu, C. Zhou, J. Zhou, and J. Lin, "Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution," 2024. [Online]. Available: https://arxiv.org/abs/2409.12191

[48] OpenAI. (2023) Text-embedding-3-small. https://platform.openai.com/docs/models/text-embedding-3-small.

[49] ——. (2023) Text-embedding-3-large. https://platform.openai.com/docs/models/text-embedding-3-large.

[50] ——. (2022) Text-embedding-ada-002. https://platform.openai.com/docs/models/text-embedding-ada-002.

[51] (2025) Automatic Prefix Caching – vLLM. https://docs.vllm.ai/en/latest/design/prefix_caching.html.

[52] DeepSeek, "Context caching," https://api-docs.deepseek.com/guides/kv_cache.

[53] Artificial Analysis. (2025) Llm api performance leaderboard: Time to first token (ttft). [Online]. Available: https://artificialanalysis.ai/models?latency=time-to-first-token#latency

[54] OpenPPL, "ppl.llm.serving," https://github.com/OpenPPL/ppl.llm.serving, 2023.

[55] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *23rd USENIX Security Symposium (USENIX Security)*, 2014, pp. 719–732.

[56] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015, pp. 605–622.

[57] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[58] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security (EuroSec)*, 2017, pp. 1–6.

[59] R. Wojtczuk and J. Rutkowska, "Attacking smm memory via intel cpu cache poisoning," *Invisible Things Lab*, pp. 16–18, 2009.

[60] D. Wang and W. Y. Dong, "Attacking intel uefi by using cache poisoning," in *Journal of Physics: Conference Series*, vol. 1187, no. 4. IOP Publishing, 2019, p. 042072.

[61] H. V. Nguyen, L. L. Iacono, and H. Federrath, "Your cache has fallen: Cache-poisoned denial-of-service attack," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1915–1936.

[62] A. Klein, "Web cache poisoning attacks," in *Encyclopedia of Cryptography, Security and Privacy*. Springer, 2025, pp. 2763–2764.

[63] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, "Dns cache poisoning attack reloaded: Revolutions with side channels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020, pp. 1337–1350.

[64] S. Son and V. Shmatikov, "The hitchhiker's guide to dns cache poisoning," in *International Conference on Security and Privacy in Communication Systems (SecureComm)*. Springer, 2010, pp. 466–483.

[65] X. Li, W. Xu, B. Liu, M. Zhang, Z. Li, J. Zhang, D. Chang, X. Zheng, C. Wang, J. Chen, H. Duan, and Q. Li, "Tudoor attack: Systematically exploring and exploiting logic vulnerabilities in dns response pre-processing with malformed packets," in *2024 IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 4459–4477.

[66] X. Zheng, H. Han, S. Shi, Q. Fang, Z. Du, X. Hu, and Q. Guo, "Inputsnatch: Stealing input in llm services via timing side-channel attacks," *arXiv preprint arXiv:2411.18191*, 2024.

[67] G. Wu, Z. Zhang, Y. Zhang, W. Wang, J. Niu, Y. Wu, and Y. 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 Symposium (NDSS)*, 2025.

[68] L. Song, Z. Pang, W. Wang, Z. Wang, X. Wang, H. Chen, W. Song, Y. Jin, D. Meng, and R. Hou, "The early bird catches the leak: Unveiling timing side channels in llm serving systems," *arXiv preprint arXiv:2409.20002*, 2024.

[69] Z. Gao, J. Hu, F. Guo, Y. Zhang, Y. Han, S. Liu, H. Li, and Z. Lv, "I Know What You Said: Unveiling Hardware Cache Side-Channels in Local Large Language Model Inference," in *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*, 2025.

[70] Y. Zhang, J. Rando, I. Evtimov, J. Chi, E. M. Smith, N. Carlini, F. Tramer, and D. Ippolito, "Persistent pre-training poisoning of llms," in *International Conference on Representation Learning*, vol. 2025, 2025, pp. 31 323–31 340.

[71] P. He, H. Xu, J. Ren, Y. Cui, S. Zeng, H. Liu, C. Aggarwal, and J. Tang, "Sharpness-aware data poisoning attack," in *International Conference on Representation Learning*, vol. 2024, 2024, pp. 25 555–25 575.

[72] Y. Wen, L. Marchyok, S. Hong, J. Geiping, T. Goldstein, and N. Carlini, "Privacy backdoors: Enhancing membership inference through poisoning pre-trained models," vol. 37. Curran Associates, Inc., 2024, pp. 83 374–83 396.

[73] R. Jha, J. Hayase, and S. Oh, "Label poisoning is all you need," in *Advances in Neural Information Processing Systems*, vol. 36. Curran Associates, Inc., 2023, pp. 71 029–71 052.

[74] F. Nazary, Y. Deldjoo, and T. d. Noia, "Poison-rag: Adversarial data poisoning attacks on retrieval-augmented generation in recommender systems," in *European Conference on Information Retrieval (ECIR)*. Springer, 2025, pp. 239–251.

[75] X. Li, Z. Li, Y. Kosuga, Y. Yoshida, and V. Bian, "Targeting the core: A simple and effective method to attack rag-based agents via direct llm manipulation," *arXiv preprint arXiv:2412.04415*, 2024.

[76] S. Choudhary, N. Palumbo, A. Hooda, K. D. Dvijotham, and S. Jha, "Through the stealth lens: Rethinking attacks and defenses in rag," 2025. [Online]. Available: https://arxiv.org/abs/2506.04390

[77] S. Li, J. Zhang, Y. Qi *et al.*, "Clean image may be dangerous: Data poisoning attacks against deep hashing," 2025. [Online]. Available: https://arxiv.org/abs/2503.21236

[78] oCERT, "oCERT-2011-003: Multiple Implementations Denial-of-Service via Hash Algorithm Collision," https://ocert.org/advisories/ocert-2011-003.html, The Open Source Computer Emergency Response Team (oCERT), Tech. Rep., 2011.

[79] (2025) 3.1. Command line and environment. https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHASHSEED. Python Software Foundation.

[80] Python Software Foundation. (2025) *Memory Management* — python/c api reference manual. https://docs.python.org/3/c-api/memory.html.

[81] yonillasky. (2022) gh-99540: Constant hash for _PyNone_Type to aid reproducibility. https://github.com/python/cpython/pull/99541.

[82] H. Chatham, M. Droettboom, G. Choi, R. Yurchak, D. Chua, A. Khetarpal, H. Schreiner, L. Estève, B. Broere, M. Köppe *et al.*, "pyodide/pyodide: 0.28.0a3." [Online]. Available: https://doi.org/10.5281/zenodo.15525156

[83] L. Holmes. (2023) Efficiently generating python hash collisions. https://www.leeholmes.com/efficiently-generating-python-hash-collisions/.

[84] D. Coppersmith, "Another birthday attack," in *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, ser. Lecture Notes in Computer Science, vol. 218. Springer, 1985, pp. 14–17.

Appendix

A. Python Built-in Hash Function Used by vLLM

To mitigate DoS attacks caused by NCHF collisions [78], Python 3.2.3 introduced a process-level randomization factor (the environment variable `PYTHONHASHSEED`) for strings, bytes and several container types. `PYTHONHASHSEED` is generated once at interpreter start-up and remains fixed for that process [79]. For *integer scalars* (`int`, `bool`), the builtin `hash()` is a simple arithmetic transform that *does not* incorporate this seed. If every element in a tuple is deterministic (*e.g.*, all plain integers), then—because each element's individual hash is deterministic—the tuple's overall hash value remains identical across processes.

For certain singleton objects, `hash()` is derived directly from their address to make identity comparisons fast. In CPython 3.11 and earlier, `hash(None)` is computed by applying a small bit-mixing function to `None`'s pointer value [80]; beginning with Python 3.12 it is replaced by a fixed numeric constant [81], matching the long-standing treatment of `True` and `False`.

Algorithm 1 illustrates vLLM's prefix-hashing procedure: when computing the hash for each block, the hash of the preceding block is folded in, while for the very first block, the "previous-hash" value is set to `None`. Because the block hash uses only integer tokens and the `None` sentinel, it inherits none of the randomness provided by `PYTHONHASHSEED`. Hence:

- **Python ≤3.11.** The result varies across processes because it depends on `None`'s address (randomised by ASLR). An exception occurs in some sandboxes (*e.g.*, Pyodide [82]), where the object is located at a very low address and the address-derived value right-shifts to 0, making hashes accidentally uniform across processes.
- **Python ≥3.12.** `hash(None)` is the same constant everywhere, so vLLM's block hashes are fully deterministic. This deterministic mapping provides the attacker with a stable collision target.

Our remediation for vLLM is to insert an explicit, per-process salt derived from `PYTHONHASHSEED` into the `(integer, None)` sequence before applying the hash. This restores hash unpredictability and blocks pre-computed collision attacks.

Algorithm 1 SENTENCEHASH: vLLM block-wise sentence hashing.

```
Require: sentence $s$, tokenizer $\mathcal{T}$, block size $B=16$
1: $T \leftarrow \mathcal{T}(s)$                          ▷ token IDs
2: $h \leftarrow$ NIL                                     ▷ Zero-block hash
3: for each block $b$ of size $B$ in $T$ do
4:     $h \leftarrow \text{hash}(i=0,\ h,\ b)$            ▷ Python hash
5: end for
6: return $h$
```

B. Meet-in-the-Middle Attack for Python Built-in Hash Function in vLLM

Although Python's hash function is an NCHF and its inverse can be theoretically derived, its usage in vLLM is constrained by the token vocabulary size (typically around 0 to 150,000), which makes direct inversion infeasible.

To address this, we leverage an average of 4 tokens to meet the space requirements for a collision attack. The MITM attack strategy employed here is inspired by Lee Holmes [83]. As shown in Algorithm 2, we perform token collision in two stages. Specifically, the approach involves:

1) **Precomputation Phase**: Compute 2^{31} possible hash values generated by the first two tokens and store these results in a set.
2) **Collision Search Phase**: Compute the hash values for all possible combinations of the last two tokens and check for collisions against the precomputed set.

This method effectively balances computational complexity and memory usage, enabling us to exploit hash collisions within the constrained token space.

Algorithm 2 COLLISION: Meet-in-the-middle search for (a, b, c, d).

```
Require: start hash $h_0$, target hash $h_2$, length $L$=18
1: Table $\leftarrow \varnothing$                         ▷ forward map
2: for $a \in [2^8, 2^{15})$ do
3:     $h \leftarrow \text{FWD}(h_0, a)$
4:     for $b \in [2^8, 2^{16})$ do
5:         Table.insert$(\text{FWD}(h, b))$
6:     end for
7: end for
8: $h' \leftarrow h_2 - ((L \oplus C_1) \oplus C_2)$      ▷ LASTREVERSEHASH
9: for $d \in [2^8, 2^{17})$ do
10:    $h_d \leftarrow \text{REV}(h', d)$
11:    for $c \in [2^8, 2^{17})$ do
12:        $h_c \leftarrow \text{REV}(h_d, c)$
13:        if $h_c \in$ Table then
14:            return corresponding $(a, b, c, d)$
15:        end if
16:    end for
17: end for
18: abort                                                 ▷ no collision found
```

C. Details of Prefix Cache Poisoning in vLLM

In vLLM, hash is calculated by grouping 16 tokens into one set. Due to the strict temporal characteristics of the prefix cache, the prefix hash value is required when calculating each block in practice to ensure uniqueness. That is, Block_hash=hash(prefixhash, tokens). This ensures that even if the content is consistent but the prefix is inconsistent, different hash values will be generated.

For a single block, we need to determine a target hash and the desired attack payload, and obtain its tokens via a tokenizer. We obtain the padding token by calculating the following equation:

$$\text{Hash}\big(\text{hash}_{\text{prefix}},\ \text{payload} + \text{padding}\big) \ = \ \text{hash}_{\text{target}} \quad (3)$$

Specific calculation details can be found in Appendix B.

We obtain the corresponding characters for the prefix token, attack payload, and padding token through a tokenizer and send them to the corresponding LLM platform. When the collided sentence enters, vLLM calculates each block and determines whether it hits the cache. If hits, the prefix cache is reused, thereby injecting the poisoned payload into the content. The overall computation process can be found in Algorithm 3.

For multiple blocks, simply repeat the single-block method multiple times.

Algorithm 3 PREFIXCOLLISION: Attack steps.

```
Require: victim prompt $P_2$, attacker prompt $P_1$ with $|P_1| = |P_2| - 4$
1: $h_2 \leftarrow \text{SENTENCEHASH}(\lfloor P_2 \rfloor_{16})$
2: $h_1 \leftarrow \text{SENTENCEHASH}(P_1)$
3: $h_0 \leftarrow h_1 - \big((14 \oplus C_1) \oplus C_2\big)$    ▷ LASTREVERSEHASH
4: $\Delta \leftarrow$ last 4 tokens of $\lfloor P_2 \rfloor_{16}$
5: assert $\text{hash}\big(\text{False}, h_1, \Delta\big) = h_2$
6: COLLISION($h_0$, $h_2$, 18)    ▷ search $(a, b, c, d)$
```

D. Analysis of Output Sequence Inconsistency due to Mismatched Final Tokens under Prefix Cache Collisions

Consider two input sequences, denoted as S_A and S_B, with their corresponding token sequences being $T_A = \{a_1, a_2, \ldots, a_{16}\}$ and $T_B = \{b_1, b_2, \ldots, b_{16}\}$, respectively. Assume a scenario where a prefix block of sequence S_A and the corresponding prefix block of sequence S_B compute to the same hash value, *i.e.*, $\text{hash}(S_A) = \text{hash}(S_B)$. Such a hash collision will cause the prefix cache lookup for sequence S_B to erroneously hit a cache entry generated and stored by sequence S_A (which can be termed a "contaminated" cache block).

In systems like vLLM, after the block hash computation for an input sequence's prefix is completed, if such a collision occurs, the prefix cache pointer is directed to this contaminated cache block. However, the critical issue arises when, despite the reuse of the prefix block's KV state, the last token of the current input sequence S_B, specifically b_{16}, differs from the token at the corresponding position in sequence S_A that generated this cache block (or the actual final token of S_A, depending on the specific prefix length at which the collision occurred). In such cases, the subsequently generated token sequences will exhibit significant discrepancies.

The fundamental reason for this phenomenon lies in the Transformer architecture widely adopted by modern LLMs. During its auto-regressive generation process, once the prefix cache is computed (or hit), the model requires the last token of the current input sequence (in this case, b_{16}) to serve as the query vector. This query vector then interacts with the Key and Value vectors stored in the cache, typically through an attention mechanism, to predict the next token. Therefore, even though the computation for sequence S_B utilizes the prefix cache state of sequence S_A, the final output for the next token is generated based on the interaction of token b_{16} querying the prefix cache state derived from sequence S_A.

Consequently, the resulting generated content will not be equivalent to the natural continuation that sequence S_A would have produced under undisturbed conditions, nor will it be equivalent to what sequence S_B would have generated based on its own prefix cache (had no collision occurred). This thereby compromises the consistency and reproducibility of the output sequences.

E. Generation of Colliding Image Pairs

Size collision. Pick two shapes $H \times W$ and $W \times H$. Render both texts, take the darker pixel at each position to form a flat array P of length HW, then reshape it to (H, W) and (W, H). Because `tobytes()` ignores geometry, the two images share the same hash.

Palette collision. We present a simplified version of a palette collision (Figure 11), where only four indices are used to render two visually distinct black-and-white images. A single index map (with indices ranging from 0 to 3) represents the background, the overlapping region, and the two exclusive regions. Two opposing palettes are then applied: Palette A renders region 1 in white and region 2 in black, while Palette B does the reverse. Palette C visualizes the locations of all four index values as distinct colors. Since the byte stream is identical, a cache system that hashes only the raw bytes will store a single entry, even though the rendered outputs differ. The same principle applies to more complex palette collisions.

F. Hash Collision Risks in Multimodal Data within SGLang

After preprocessing, SGLang performs hashing operations on data from different modalities (*e.g.*, images). The resulting hash values are used as placeholder tokens and inserted into the token sequence. These placeholders are later replaced with the corresponding concrete multimodal features during the actual inference stage. The hashing procedure in SGLang depends on where the multimodal object resides—different algorithms are used for CPU-based and GPU-based inputs.

For CPU-based inputs, SGLang employs the SHA256 hash function. However, to ensure the resulting values fit within tensor numerical limits, a modulo 2^{30} operation is applied to the raw hash digest. This modulo operation significantly reduces the effective hash space from SHA256's native 2^{256}

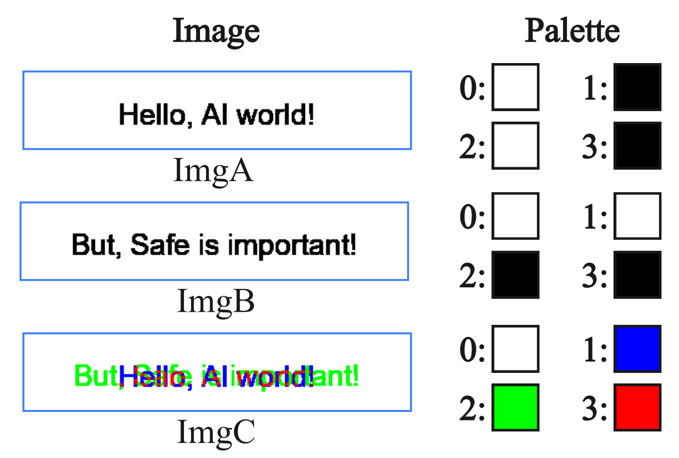


Fig. 11: An example of palette collision.

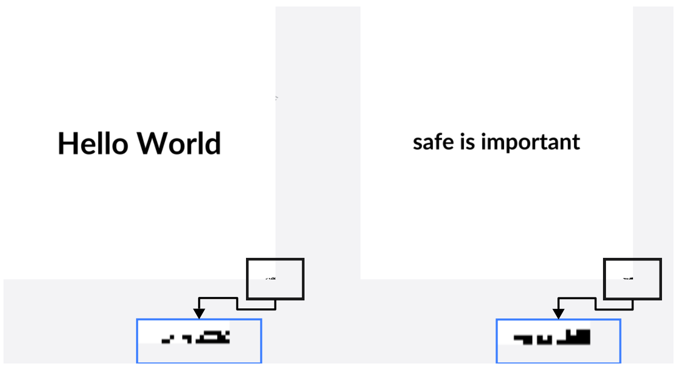


Fig. 12: SGLang hash collision via pixel manipulation.

to merely 2^{30}, severely compromising collision resistance. Consequently, a targeted second preimage attack becomes feasible with approximately 2^{30} attempts, while the Birthday Paradox [84] allows attackers to generate controlled colliding pairs with only about 2^{15} inputs.

Notably, we previously exploited the pad_value set by the image processor, which caused Radix Attention to incorrectly reuse the KV cache. This was equivalent to reusing the image processing results, thereby leading to an erroneous interpretation of the image. In subsequent updates, SGLang implemented a separate image processor cache for online serving. SGLang also truncated the SHA-256 hash to an effective length of only 64 bits, resulting in a similar calculation for attack complexity. The distinction between these two exploitation scenarios is that one precludes the modification of the prompt prefix, whereas the other allows it. We demonstrate a practical 64-bit collision instance in Figure 12, which we achieved by manipulating the last few pixels.

For GPU-based hashing, SGLang uses a simplified version of the xxHash algorithm, which lacks collision resistance and is reversible. This allows an attacker to directly construct multimodal inputs that produce the same hash value, enabling effective cache poisoning attacks.

G. Calculation Process of Defense Methods

Add Random Numbers. Adding these at startup prevents precomputed collisions. For a 50,000-entry KV database (avg. 400 tokens), a collision requires 2^{40} messages.

Calculation Justification: Assuming a 64-bit hash space ($N = 2^{64}$) and one hash generated per 16 tokens. A 400-token message generates $m = 25$ hashes, and the database contains $K = 50,000 \times 25 = 1,250,00$ hashes. The expected number of messages to cause a collision is:

$$E(\text{messages}) = \frac{N}{m \times K} = \frac{2^{64}}{25 \times 1,250,000} \approx 2^{40}$$

Secure Hashing. Using a secure hash function significantly increases computational costs. SHA256 requires 2^{128} computations for a collision and 2^{255} for specific poisoning, rendering attacks infeasible.

Calculation Justification:

- **Finding a Collision (Birthday Attack):** The complexity to find *any* collision pair for an n-bit hash function is subject to the birthday attack, with a complexity of $\mathcal{O}(\sqrt{2^n})$. For SHA256, where $n = 256$, Computations $\approx \sqrt{2^{256}} = (2^{256})^{1/2} = 2^{128}$.
- **Poisoning Specific Sentences (Preimage Attack):** This requires finding an input for a *specific* hash output. This is a preimage attack, whose complexity is proportional to the full size of the hash space, $\mathcal{O}(2^n)$. On average, 2^{n-1} computations are needed. For SHA256, Computations $\approx 2^{256-1} = 2^{255}$.

H. LLM Customer Service System Prompts

To simulate diverse and realistic user behaviors in customer service scenarios, we use specific prompts to instruct the LLM to generate synthetic user requests across multiple intent categories (including Subscription, Unsubscription, Inquiry, as well as Impatient/Rude tones). In addition, we design dedicated system prompts for the customer support agent, for semantic cache validation (LLM filtering), and for using GPT-5 to detect and verify potentially misleading content in the LLM's responses. For brevity, the full content of these prompts is available in our project repository: LLM Customer Service System Prompt.

I. LLM Security Analyst Prompt

The prompt used for LLM security analysis mimics an expert security analyst specializing in static code analysis to identify vulnerabilities such as SQL injection and XSS.

We also designed specific attack scenarios: **C1**, a concrete example of an actual poisoning attempt; **C2**, an example that collides with C1 using specific token IDs; and **C3**, a hidden-block attack where the content collides with the first block.

The detailed prompts for the Security Analyst, along with the full scripts for scenarios C1, C2, C3, and the corresponding model responses, can be found at LLM Security Analyst Prompt.