I Know What You Asked: Prompt Leakage via KV-Cache Sharing in Multi-Tenant LLM Serving

Guanlong Wu
SUSTech
santiscowgl@gmail.com

Zheng Zhang
ByteDance Inc.
2hang2heng@buaa.edu.cn

Yao Zhang
ByteDance Inc.
zhangyao.crypto@bytedance.com

Weili Wang
SUSTech
12032870@mail.sustech.edu.cn

Jianyu Niu*
SUSTech
niujy@sustech.edu.cn

Ye Wu
ByteDance Inc.
wuye.2020@bytedance.com

Yinqian Zhang*†
SUSTech
yinqianz@acm.org

***Abstract*—Large Language Models (LLMs), which laid the groundwork for Artificial General Intelligence (AGI), have recently gained significant traction in academia and industry due to their disruptive applications. In order to enable scalable applications and efficient resource management, various multi-tenant LLM serving frameworks have been proposed, in which the LLM caters to the needs of multiple users simultaneously. One notable mechanism in recent works, such as SGLang and vLLM, is sharing the Key-Value (KV) cache for identical token sequences among multiple users, saving both memory and computation.**

This paper presents the first investigation on security risks associated with multi-tenant LLM serving. We show that the state-of-the-art mechanisms of KV cache sharing may lead to new side channel attack vectors, allowing unauthorized reconstruction of user prompts and compromising sensitive user information among mutually distrustful users. Specifically, we introduce our attack, PROMPTPEEK, and apply it to three scenarios where the adversary, with varying degrees of prior knowledge, is capable of reverse-engineering prompts from other users. This study underscores the need for careful resource management in multi-tenant LLM serving and provides critical insights for future security enhancement.

I. INTRODUCTION

The rise of Large Language Models (LLMs) like GPT [25] or Llama [44] has enabled a variety of new applications, including universal chatbots [5], virtual assistants [4], and code generators [6], applicable to both large-scale cloud deployments and small-scale local setups. As LLM applications become widespread, effectively serving concurrent requests from multiple users has become a non-trivial research question [32], [61], [34]. In fact, processing a single LLM request is already costly, as it generates Key-Value (KV) cache [37] for each token during the inference phase, occupying a considerable amount of GPU memory [12]. With limited GPU memory capacity, the extensive size of the KV cache [12] restricts the ability to serve concurrent requests, becoming a critical bottleneck in multi-tenant scenarios.

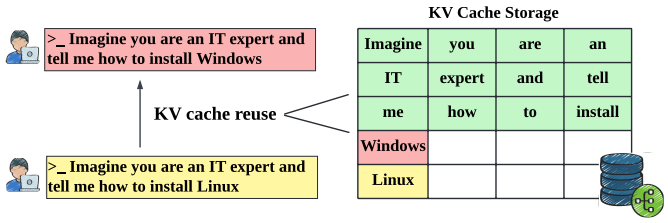

Figure 1: KV cache reuse.

One promising solution proposed by recent work (*e.g.*, vLLM, SGLang) [32], [61], [53] is to share the KV cache across the requests to reduce both computation and memory usage. The rationale is that identical tokens in different requests can generate the same KV cache if their preceding tokens are also identical. Figure 1 illustrates an instance of KV cache sharing. When the first user submits the query "Imagine you are an IT expert and tell me how to install Windows", the KV cache for each token is computed and stored on the LLM server. Thereafter, if another user issues a query "Imagine you are an IT expert and tell me how to install Linux", the initial segment of the sentence—"Imagine you are an IT expert and tell me how to install"—has an identical KV cache. Hence, the second user can directly utilize the KV cache previously computed for the first user, recalculating only the differing segment, "Linux". KV cache sharing prevents duplicate KV storage on the GPU, allowing more user requests to be served concurrently. More importantly, it reduces the serving time for individual requests by eliminating unnecessary calculations.

However, in this paper, we point out that the KV cache sharing mechanism is not secure. Our key insight is that the KV cache sharing may inadvertently create side channel information, which can be leveraged by the adversary to carefully craft requests sent to the LLM server to determine if its requests match the other users', thereby recovering other users' prompts.

In this paper, we dig into the current KV cache sharing

*Affiliated with the Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering

†Corresponding Author

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241772
www.ndss-symposium.org

strategies and demonstrate how these can be exploited to reconstruct user input prompts. In particular, we propose PROMPTPEEK, which leverages the changes of serving order as side channel information, to repeatedly extract other users' prompts from the LLM service. More specifically, PROMPTPEEK utilizes the side channel information to monitor the KV cache hits and extracts one token at a time from another user's prompt. By iteratively repeating this process, PROMPTPEEK can reconstruct the entire prompt from another user. We assess PROMPTPEEK across three scenarios, where the adversary possesses different levels of background knowledge: knowledge of the prompt template, knowledge of the prompt input, and no background knowledge. Our results show that the adversary can achieve an average success rate of 99% in fully or partially reversing the prompt input, 98% in reversing the prompt template, and 95% without additional background knowledge, when tested on a Llama2-13B model on an A100 80G GPU. From a higher level, our results show that the attack depends on three key factors: memory capacity, concurrent users' requests, and attack requests. Memory capacity sets the feasibility of the attack, while both users' and attack requests accelerate memory depletion. Besides, the attacker's background knowledge minimizes the number of requests needed to recover prompts.

Our contributions. To summarize, we make the following contributions in this paper:

- We are the first to touch on the security risks in multi-tenant LLM serving, identifying it as a new attack surface in LLM security. We not only investigate the risks associated with KV cache sharing but also highlight the broader implications for any future LLM serving frameworks. Our research emphasizes the need for careful management of shared resources in these environments.
- We propose PROMPTPEEK and assess its feasibility across three scenarios. Unlike previous studies that only approximate prompt content [41], [52], PROMPTPEEK accurately reconstructs prompts, which significantly increases privacy risks, as prompts may contain sensitive information like bank account numbers or health records. More importantly, we recognize that KV cache sharing is still in its early stages, so we outline three critical attack conditions that service providers and framework developers should consider in case of potential security risks.
- We simulate real-world LLM scenarios to evaluate the effectiveness and cost of our attack across three different environments, utilizing four distinct datasets on an A100 80G GPU. Our results reveal that our attack not only successfully uncovers prompt secrets but also at a low cost. For example, knowing the prompt template allows the adversary to uncover the prompt's secrets, including gender, age, weight, and height, with just 60 requests in total.

Ethical considerations. We responsibly disclosed our findings to both the framework developers (SGLang, our primary target) and service providers, including OpenAI, ByteDance, and Anthropic. So far, we have received a response from ByteDance acknowledging the new side-channel vector identified in our work. Also, we are actively engaging with SGLang to discuss updates to counter the attack.

II. LLM SERVING

In this section, we describe the background of LLM serving, with a particular emphasis on multi-tenant LLM serving and KV cache sharing strategies, setting the stage for our exploration of potential security risks. Specifically, LLM serving refers to deploying Large Language Models (LLMs) to offer inference service. Various frameworks, including vLLM [32], SGLang [61], LightLLM [34], and DeepSpeed [40], have been developed to serve LLMs.

A. LLM Inference

As the core of LLM serving, the inference mechanism is primarily based on Transformer blocks [47], which consist of multi-head attention mechanisms and feed-forward networks. Given a sequence of input tokens, LLMs sequentially produce output tokens through an autoregressive process. The inference process is segmented into two main phases: the prefill phase and the incremental decoding phase. In the prefill phase, LLMs simultaneously process all the input tokens, resulting in the production of the first output token. Subsequently, during the decoding phase, LLMs generate subsequent output tokens, each relying on the previously produced token, thus building the response iteratively.

KV cache. Through the inference process, each token generates a unique **K**ey-**V**alue cache (KV cache) [37], used for decoding further tokens and reducing computational overhead. Some notable characteristics of KV cache include:

- The computation of KV cache depends on all the previous tokens, which means that the same token does not always generate the same KV cache. For example, in the phrase "How do you do?", the two 'do's produce distinct caches.
- Conversely, when preceding tokens are identical, their resulting KV caches are also identical. This is observed in sentences like "I enjoy coding" and "I enjoy debugging", where the tokens 'I' and 'enjoy' generate the same KV cache in both instances, under the same LLM.
- KV cache poses a significant bottleneck in LLM serving, due to its considerable memory requirements. It consumes roughly 1MB per token [13], and can easily grow larger than the model weights [12]. Given the limited GPU memory, emerging research suggests storing the KV cache in CPU memory or on disk, and retrieving it as needed [29], [38].

B. Multi-tenant LLM Serving

Multi-tenant LLM architecture spans a wide spectrum from small-scale local setups to large-scale cloud deployments. A pervasive challenge across these scenarios comes from the constrained GPU memory compared to the extensive KV-cache demands of individual requests (Sec. II-A), which makes batching multiple users' requests difficult. One solution

Request 1:

```
Help me translate this
sentence into English,
...
```

Request 2:

```
Help me translate this
sentence into French,
...
```

(a) KV cache shared: "Help me translate this sentence into".

Request 1:

```
Help me translate this
sentence into English,
...
```

Request 2:

```
Translate this sentence
into English,
...
```

(b) No KV shared because the first tokens are different.

Figure 2: KV cache sharing policy.

proposed by prior works [32], [61], [53] is to share the KV cache among the users' requests. In this way, the GPU can process and batch more requests concurrently. Additionally, leveraging the prior stored KV cache can reduce the time required to serve the request [32], [61], [53].

KV cache sharing. As introduced in Sec. II-A, the KV cache for a given token remains consistent across different requests if all preceding tokens in those requests are identical; that is, the KV cache produced by a token in the request can be reused by another request, as long as all preceding tokens match. Figure 2 depicts scenarios of KV cache sharing, highlighting that sharing is feasible only when all preceding tokens match, and a mismatch in the initial token prevents any sharing. Automatic KV cache sharing has been enabled in many state-of-the-art LLM frameworks, such as SGLang [61] and vLLM [32]:

- In vLLM [32], [22], incoming requests generate KV Cache Blocks as usual (for both prefilling and decoding phase), each tagged with metadata including a hash of previous tokens, last accessed time, and a reference count tracking active sequences using the block. These caches are retained in memory for as long as possible. When memory limits are reached, the oldest KV caches are evicted to make space. For subsequent requests, the system assesses the potential for cache reuse by comparing the hashes of incoming request blocks against those of existing KV Cache Blocks, facilitating efficient cache sharing.
- SGLang [61], [20] presents a more thorough discussion on the KV cache sharing mechanism. It utilizes a radix tree for storing KV cache, also maintaining them in GPU memory as long as space permits. In particular, when multiple users send requests, the system queues them and forms a batch from the first few. The KV cache for these requests is then calculated and stored in the GPU using a radix tree.

  1) For cache eviction, it uses a Least Recently Used (LRU) strategy when GPU memory is full. Only the KV cache for tokens in the current running batch is kept, while all other entries are removed. For example, if "Imagine you are an IT expert" is stored and the running batch contains "Imagine you are an IT specialist," the shared part "Imagine you are an IT" is preserved, but "expert" is evicted to free memory.

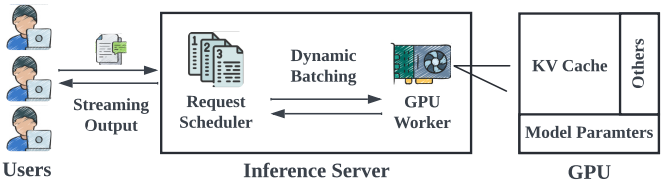

Figure 3: System model.

2) For scheduling policy, SGLang implements a state-of-the-art scheduling policy, namely Longest Prefix Match (LPM), to mitigate cache thrashing. This policy organizes incoming requests by sorting them based on the length of shareable tokens that match the previously stored KV cache entries. For example, if the KV cache for the request "Imagine you are an IT expert" is stored and there are two pending requests—"Imagine you are an IT specialist" and "Imagine you are an actor"—the system prioritizes the first request because it shares a longer token sequence with the stored KV cache.

III. Overview

As introduced in Sec. II-B, multi-tenant LLM serving spans a broad spectrum, from small-scale local deployments to large-scale cloud-based applications. Some critical factors may vary across these scenarios, posing distinct challenges in achieving a comprehensive model of each. As the first attempt to analyze LLM serving security, we establish a unified system model for all scenarios in Sec. III-A, discuss the considered threat model in Sec. III-B, and present the high-level overview of our proposed attack, PromptPeek, in Sec. III-C.

A. System Model

Figure 3 illustrates a high-level overview of our system model. In general, multiple users send requests to the LLM inference server independently. The scheduler aggregates the requests into batches within the request queue and dispatches them to the GPU worker for inference processing. Upon completion of the inference, the results are relayed back to the users by the scheduler.

Entities. This system model consists of three entities: users, scheduler, and GPU worker.

- **Users.** We assume N users utilize the LLM service, with each dispatching its request (denoted by r) to the inference server at a frequency f. N and f can vary significantly from local setups to cloud applications, which are thoroughly analyzed and evaluated in our research. Each request r consists of i tokens $\{t_1, t_2, \ldots, t_i\}$, where i varies and represents the length of each request. After being processed by the inference server, each user receives an output comprising j tokens $\{t_{i+1}, t_{i+2}, t_{i+3}, \ldots, t_{i+j}\}$, where j varies and can be specified through user-defined settings in the serving framework (*e.g.*, max_tokens in vLLM) [20], [22].
- **Scheduler.** The scheduler sorts the incoming requests through a predetermined scheduling policy P_S, and assembles the first m requests $\{r_1, r_2, \ldots, r_m\}$ into a batch b

directed to the GPU worker, adhering to a specified batching policy P_B. All requests within the same batch are processed concurrently, and their KV cache is managed by policy P_{KV}. After obtaining the results from the GPU worker, it ensures each outcome is delivered to the appropriate user following the output policy P_O.

- **GPU worker.** The GPU executes the inference task using the LLM loaded in its internal memory on the incoming batch b. For simplicity, we only consider hosting a single LLM on a single GPU in our system model. The GPU memory capacity M is divided into three primary components: model parameters M_{model}, KV cache M_{KV}, and others for activation M_{others} [32]. M_{model} statically persists in GPU memory throughout the serving process; M_{others} takes a small amount of memory space, while the remainder is all for M_{KV}, which is allocated per request during serving. Given that the size of each token's KV cache, denoted by m_{t_i} (m_{t_i} depends on the specific LLM), the maximum number of tokens, T_{max}, can be calculated as $T_{max} = \frac{M_{KV}}{m_{t_i}}$. As introduced in Sec. II-B, when the KV cache sharing is on, the KV cache can be preserved as long as there is still enough memory in the GPU. Once the number of processed tokens reaches max token capacity T_{max}, KV cache eviction is performed, in accordance with the eviction policy P_E.

Policy specifications. We specify the suite of policies adopted in our system model's operations as follows:

- **Scheduling policy P_S.** Scheduling policy refers to how the scheduler manages the incoming requests. In our system model, we consider the state-of-the-art scheduling policy tailored for KV cache sharing—Longest Prefix Match (LPM)—from SGLang (Sec. II-B), which is the only scheduling policy designed to enhance KV cache sharing efficiency. In this case, the waiting requests are sorted based on the length of matched tokens, and those having longer shared KV will be served earlier. Other policies include First-In-First-Out (FIFO), which processes requests by their arrival time, and Random, which processes requests arbitrarily [20].
- **Batching policy P_B.** Batching policy refers to how the scheduler groups requests for GPU processing. In our system model, we follow the existing frameworks [32], [61] and adopt continuous batching. Continuous batching ensures that a batch of requests is formed whenever there is sufficient available space in the GPU. The batch size is dynamically determined based on the remaining memory capacity. Alternatives include static batching [8], where the system waits for the entire batch to complete before starting the next. Additionally, the batch size is also influenced by server configuration parameters, such as max_num_seqs in vLLM [22]. These parameters impose a maximum limit on the batch size, ensuring it does not surpass predefined thresholds. A large batch size can enhance throughput but also increase individual latency. Thus, having a maximum batch size is critical to ensure balanced performance, which is vital for the responsiveness of online LLM services [11].

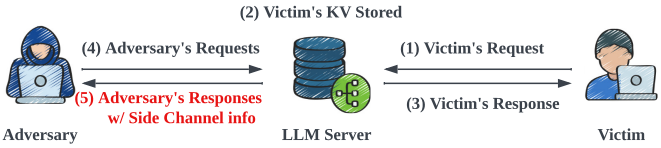

Figure 4: Attack overview.

- **KV cache policy P_{KV}.** KV cache policy states how the KV cache is managed and shared. In our system model, we follow the state-of-the-art KV cache sharing mechanism in SGLang [61], where a radix tree is used for storing KV cache. In fact, the concept of KV cache sharing was first introduced by vLLM [32], but the sharing mechanism has not been discussed in detail and is still under development [22].
- **Output policy P_O.** Output policy refers to how the scheduler delivers the generated results to users. Our system model aligns with the methodology employed by OpenAI [15], where responses are streamed to users in real time. Specifically, the first generated token is immediately sent back to the corresponding user, starting the response process instantly.
- **Eviction policy P_E.** Eviction policy refers to how the system evicts KV cache when the storage capacity is reached. In our system model, we follow the eviction policy from existing frameworks [32], [61], applying the Least Recently Used (LRU) policy for KV cache eviction. Notably, when the memory capacity is reached, all KV cache entries except those in the current batch are evicted simultaneously, creating substantial free space on the GPU with each eviction.

B. Threat Model

The adversary's goal is to reconstruct the prompts sent by other users using the side channel information. We examine various scenarios in Sec. V where the target may vary from the entire prompt to specific parts of it. For instance, if we assume the adversary knows the prompt template (*e.g.*, the adversary and the users are using the same LLM application), then the target is the input (Sec. V-B).

The adversary's capabilities are the same as those of an ordinary user, with no direct control over the inference server. We assume the attacker knows the default internal mechanisms of the LLM server (*e.g.*, scheduling policy, eviction policy, as stated in Sec. III-A), following the standard assumption in prior side-channel research [33]. The background knowledge of the LLM extends only to the tokenizer, which is always accessible to the adversary. For instance, closed-source LLM services like OpenAI publicly release their tokenizer [21], [16] for token-level billing, and open-source LLMs [14] also make their tokenizers public, which remain unchanged after fine-tuning. Furthermore, adversaries can only interact with the server through the commonly defined client APIs provided by LLM serving frameworks [22], [20] (*e.g.*, setting the max output length by max_tokens in vLLM) [22].

C. Attack Overview

We next outline our proposed attack, PROMPTPEEK, against multi-tenant LLM servings. The intuition behind PROMPTPEEK is that the state of one tenant's request can be influenced by others, inadvertently creating cross-tenant side channels. Figure 4 illustrates the high-level attack overview. Initially, the user submits a request to the LLM server, where its KV cache is computed and retained. Subsequently, an adversary carefully crafts and dispatches requests to the LLM server. Using a side channel established through the response, the adversary is able to determine if her request has reused the KV cache previously stored for the victim user—namely, whether it shares a consecutive token sequence starting from the initial tokens with the victim user (see Sec. II-B). By using this side channel information, which we will explain in more detail in Section IV, the adversary can ascertain whether their input tokens match with those of the user, thus reconstructing the user's input.

Specifically, PROMPTPEEK is executed in a loop, in which each iteration extracts one token. The end of this loop leads to a reconstruction of a prompt from the extracted tokens and, if needed, repeats PROMPTPEEK to attack another prompt.

- **Token extraction** is the very basic attack primitive in PROMPTPEEK to recover one token from a prompt. This process includes the generation of candidate tokens, the selection of tokens to be sent to the LLM, and the determination of a match via side channels.
- **Prompt reconstruction** is performed after repeating the basic attack primitive multiple times. The adversary gradually reconstructs the whole prompt using the extract tokens. The main task for the adversary is to determine when to conclude the prompt reconstruction and to switch to another one.

IV. PROMPTPEEK

In this section, we present token extraction in Sec. IV-A and prompt reconstruction in Sec. IV-B.

A. Token Extraction

PROMPTPEEK extracts one token from another user's prompt at a time, progressively reconstructing the entire prompt by repeating the token extraction procedure. Therefore, token extraction can be seen as an attack primitive, which can be divided into two distinct phases: candidates generation phase and token selection phase. In the candidates generation phase, the adversary creates likely candidates for the next token. In the token selection phase, the adversary sends these candidates to the LLM and uses side channel information to determine which candidate token matches the target token. Figure 5 illustrates this token extraction attack primitive. Assuming the LLM server has stored the KV of a prompt "Imagine you are an IT expert,", and the adversary is preparing to reconstruct the next target token, 'an', with the already reconstructed fragment "Imagine you are".

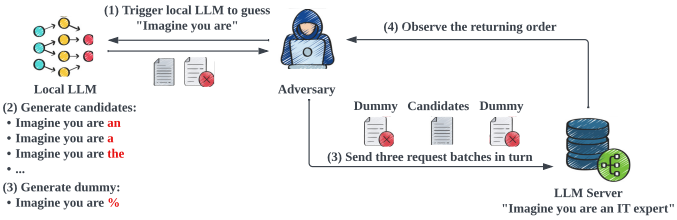

Figure 5: Token extraction in PROMPTPEEK.

1) Candidates Generation: We generate candidate tokens using existing strategy in jailbreaking attacks [42], where one LLM is used to attack another LLM. In this case, the adversary sets up a local LLM and uses it to generate the top k most likely tokens as candidates, based on the already reconstructed fragment. For instance, given the input "Imagine you are," the local LLM might suggest candidates such as "Imagine you are an", "Imagine you are a", "Imagine you are the", etc.

The adversary's local LLM can be any LLM that uses the same tokenizer as the target LLM service—a reasonable assumption following our threat model (Sec. III-B). This is essential because different tokenizers may segment text differently [39]. For instance, some might treat a whole word as one token, while others break it into subwords, leading to a token mismatch. Different LLMs may vary in their ability to correctly predict the next word, but it is not the focus of our study. For simplicity, we assume the attacker uses the same LLM as the target LLM service.

Additionally, the local LLM identifies one least likely token as a dummy token and generates a corresponding dummy request for the adversary (*e.g.*, "Imagine you are %"). The dummy token is only used to facilitate the observation of the side channel (further explained in the token selection phase). With a very minimal chance that the dummy token matches the target, reconstruction of this prompt will stop and PROMPTPEEK immediately switches to another prompt.

2) Token Selection: This phase involves strategically sending dummy and candidate requests and observing the return order as side channel information to determine if any token matches. PROMPTPEEK exploits the default scheduling policy, LPM (Longest Prefix Match, Sec. II-B), which prioritizes the longest matching token sequences. More specifically, among all incoming requests queued for scheduling, those with a longer matched token sequence—even by one token—will be prioritized and served first. This indicates that the successfully matched candidate is processed before the rest, and PROMPTPEEK strategically sends the requests to make this effect more distinguishable.

- **Candidates sending strategy.** Simply sending all candidate requests concurrently and examining the return order cannot determine if the order has been altered, as it is uncertain whether the first returned request just arrived earlier by chance or because its order was intentionally changed by LPM. Thus, as depicted in Figure 5, we introduce dummy requests both before and after the candidates. The pre-candidate dummy requests fill up the waiting queue to ensure

Running batch:

```
Imagine you are %  +
Imagine you are %  |-Run
Imagine you are %  +
```

Stored KV cache:

```
Imagine you are an IT expert  (from victim)
Imagine you are %  (from dummy requests)
```

Waiting queue:

```
Imagine you are %  +
Imagine you are %  |
Imagine you are %  |-Pre
Imagine you are %  |
Imagine you are %  +
Imagine you are a  +
Imagine you are an |-Cands
Imagine you are the+
Imagine you are %  +
Imagine you are %  |
Imagine you are %  |-Post
Imagine you are %  |
Imagine you are %  +
...
```

(a) Serving order before LPM.

Running batch:

```
Imagine you are %  +
Imagine you are %  |-Run
Imagine you are %  +
```

Waiting queue:

```
Imagine you are %  +
Imagine you are %  |
Imagine you are %  |-Pre
Imagine you are %  |
Imagine you are %  +
Imagine you are an +-Match
Imagine you are %  +
Imagine you are %  |
Imagine you are %  |-Post
Imagine you are %  |
Imagine you are %  +
Imagine you are a  +
Imagine you are the|-Cands
...
```

(b) Serving order after LPM.

Figure 6: Impact of LPM on the serving order.

that all candidates are in the waiting queue and are scheduled according to the LPM policy. The post-candidate dummy requests amplify the order effect, creating a distinguishable pattern when the order is indeed changed. All dummy requests are identical in one iteration. Additionally, we set the output length of all requests, whether dummy or candidate, to just one (by exploiting the client API to set the max output length). This approach minimizes the strain on GPU memory and maintains the effectiveness of our attack.

- **Side channel information.** Figure 6 illustrates how LPM influences the service order of incoming requests when there is a successful match. Figure 6-a depicts the snapshot prior to LPM taking effect, where requests arrive sequentially as dummy-candidates-dummy (Figure 5). Initially, a batch of dummy requests fills the running batch, causing subsequent requests to queue up. At this stage, the dummy requests, which are in execution, have their KV cache calculated and stored. Now the stored KV cache contains both phrases "Imagine you are an IT expert" from the victim, and "Imagine you are %" from the dummy requests. Subsequently, LPM sorts the waiting queue based on the length of tokens matches with the stored KV cache. In this specific example, both the dummy requests and one matched candidate have a matched token length of 4, whereas other candidates have a matched token length of 3. Requests with equal match lengths retain their original order in the queue. This ordering process results in the matched candidate being sandwiched between the dummy requests, as shown in Figure 6-b, while other candidates will wait until both the dummy and matched requests have been processed. In summary, the observable patterns from the adversary are straightforward: if *a match occurs*, the order of return requests is, pre-candidate dummy requests, the matched request, post-candidate dummy requests, and then the remaining candidate requests; if *no match occurs*, the sequence is pre-candidate dummy requests, post-candidate dummy requests, followed by all candidate requests.
- **Request batch size.** The batch size of candidate requests does not need to be equal to the number of candidates generated by the local LLM. As evaluated in Sec. VI, some tokens may require thousands of candidates to be successfully recovered, while others need fewer than ten. A larger candidate request batch speeds up the attack by reducing the rounds needed to send candidates but may waste requests. In contrast, a smaller batch makes the attack slower but optimizes request usage. We evaluate the impact of candidate batch size on attack time and request usage in Sec. VI. As for the dummy request batches, the number of pre-candidate dummy requests should exceed the server's max batch size setting (see Sec. III-A) to ensure that the incoming candidate requests queue up and are scheduled by LPM, while the number of post-candidate dummy requests should also exceed the maximum batch size. This ensures that unmatched candidates are not processed in the same batch as the post-candidate dummy requests, preventing interference with pattern observation. We also evaluate the impact of dummy batch size on the attack in Sec. VI.

B. Prompt Reconstruction

By repeating the token extraction process, the adversary can gradually reconstruct the prompt. Next, we present our strategy for deciding when to finish reconstructing one prompt and switching to another. Additionally, we propose an optimization strategy that enables the adversary to reconstruct prompts from a clean slate.

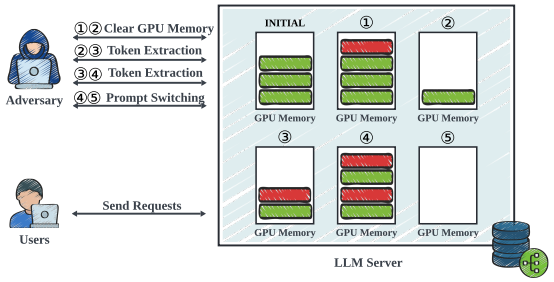

Figure 7: Prompt reconstruction of PROMPTPEEK.

1) Prompt Switching: The adversary should switch to reconstructing another prompt if the current target prompt has been evicted from memory. Although the adversary cannot directly know when the target prompt is evicted, it can infer so by monitoring its own unmatched candidate prompts. Since these candidate prompts are sent after the target prompt, the eviction of a candidate prompt indicates that the target prompt has also been evicted based on LRU. The target prompt might

occasionally remain longer than the candidate requests due to active uses by the victim user. We do not consider such corner cases in our analysis.

Using the example from Figure 5 where we are reversing the prompt "Imagine you are an IT expert" and have already reversed "Imagine you are" with the next target token being "an". If a previous candidate request to guess "are" was "Imagine you were", we can determine if the target prompt is still in memory by testing if "Imagine you were" is flushed. If it is flushed, it indicates that the target prompt has also been flushed and it's time to switch to another prompt. This check can be done every round or every few rounds, serving as the condition to switch to another prompt.

We continue to employ the strategy outlined in Sec. IV-A to determine whether an unmatched candidate request has been flushed. We resend the previously sent candidate request and surround it with dummy requests of the same length, both before and after the candidate. If this candidate request is already flushed, its order will be disrupted, as it has one fewer token than the matched length of the dummy requests. For instance, if the prompt "Imagine you were" is flushed, the sequence "Imagine you" remains active and is not flushed due to continuous extraction efforts. Therefore, "were" is the only part that gets flushed, resulting in one shorter matched length.

2) Optimization: Our attack strategy exploits the available GPU memory, where more remaining memory enables longer prompt reconstructions. However, each time we initiate a new prompt reconstruction, the GPU's memory usage is unknown. To ensure a fresh start for each prompt reconstruction, we manually flush the GPU's KV cache. According to the eviction policy, when GPU memory reaches capacity, all stored KV caches, except those used by the current batch of requests, are automatically cleared (Sec. III). To trigger the eviction, we send non-identical dummy requests to saturate the GPU memory. By periodically checking if any dummy request is flushed—same as the prompt switching method—we can tell if the KV cache eviction is triggered and has released most of the KV cache. After a brief waiting period to allow new prompts from the victim user to arrive, PROMPTPEEK begins reconstructing a new prompt from these newly arrived ones.

Non-identical dummy requests. During the token extraction phase, we used identical dummy requests to avoid additional memory use. However, for non-identical dummy requests, we begin each request with a unique token followed by a long token sequence, which ensures that each request cannot share the KV cache. Also, we set the output token length for these requests to the maximum possible, contrasting with the single-token output used in identical dummy requests.

Figure 7 details the prompt reconstruction process. Initially, the adversary clears the KV cache by sending dummy requests, creating a clean slate. After a brief waiting period t, during which N users send requests at frequency f (Sec. III), the server accumulates $N \times f \times t$ prompts. The adversary then hooks one of these prompts and reconstructs this prompt by repeating token reconstruction. During this process, both users' requests and the attacker's requests squeeze the remaining GPU memory. The process concludes once the GPU reaches its capacity and this prompt is confirmed to be flushed from memory. By repeating the process, the adversary continuously extracts prompts from the LLM service.

V. ATTACK SCENARIOS

In this section, we explore three widely recognized scenarios to demonstrate the application of PROMPTPEEK.

A. Scenario 1: Whole Prompt Reconstruction

Scenario description. This scenario represents the basic case, where the adversary has no prior knowledge about the victim users or the prompt. It also represents the most common situation in today's LLM services [5], where diverse users independently send their prompts to the service without sharing commonalities in their content. This scenario serves as the baseline, and is used to test the overall effectiveness of PROMPTPEEK.

Threat model. This scenario follows the threat model described in Sec. III-B.

Methodology. This scenario employs the attack mechanism outlined in Sec. IV without any modification to assess the PROMPTPEEK's effectiveness.

B. Scenario 2: Input Reconstruction

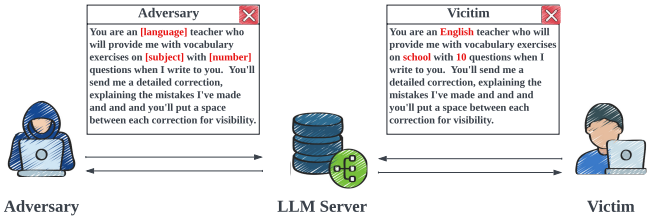

Figure 8: Input reconstruction.

Scenario description. This scenario follows the attack model shown in Figure 4, where the attacker possesses additional knowledge of the users' prompt template and tries to reconstruct the exact users' input. The prompt template, a creation of prompt engineering [50], [48], is designed to guide users in submitting inputs to achieve more accurate answers. Commonly, these templates may be shared by different users or within a specific group, whereas the precise inputs remain confidential between individual users. As LLM applications [60] become increasingly prevalent, this scenario is gaining prominence, with users of the same application often employing identical prompt templates. Figure 8 demonstrates this scenario using a template from PromptBase, one of the largest prompt marketplaces [18]. In this case, the majority of the prompt is predetermined by the template. The elements "[language]", "[subject]", and "[number]" are placeholders where users insert their specific, confidential information, representing the victim's core secrets. We consider two main prompt templates in our research following the previous study [17]:

Table I: Prompts dataset.

Prompt style	Dataset	Example	Count
general	ultrachat [28]	What are some of the most intriguing museums to discover in the city of Paris, France? Wow, there are so many interesting museums in Paris! Which one do you think I should visit first? I think I'll start with the Louvre since it's so iconic. Can you give me any tips on how to navigate such a massive museum? I'm so excited to see the Mona Lisa. Do you think it's worth waiting in the crowds to get a closer look?	77444
cloze-style	PromptBase [18]	You are an [language] teacher who will provide me with vocabulary exercises on [subject] with [number] questions when I write to you. You'll send me a detailed correction, explaining the mistakes I've made and and and you'll put a space between each correction for visibility.	180
role-based	awesome-chatgpt-prompts [3]	I want you to act as my personal shopper. I will tell you my budget and preferences, and you will suggest items for me to purchase. You should only reply with the items you recommend, and nothing else. Do not write explanations. My first request is "I have a budget of $100 and I am looking for a new dress."	153
instruction-based	alpacca-gpt4 [2]	Reword the following sentence to the past tense. She is writing a novel inspired by her grandfather.	1000

- *Cloze-style prompts:* These prompts consist of incomplete sentences or paragraphs with missing words or phrases, where the users fill in with their personal information. Figure 8 is an example of the cloze-style prompts.
- *Prefix-style prompts:* These prompts begin with a prefix that sets the direction for the user's response. Variants include role-based prompts, where the LLM adopts a specific role (*e.g.* "imagine you're an IT expert"); instruction-based prompts, which provide a single instruction to LLM (*e.g.*, Reword the following sentence to the past tense. She is writing a novel inspired by her grandfather.).

Threat model. Building on the threat model in Sec. III-B, the adversary's goal changes to deduce the other users' inputs instead of the whole prompt. Besides, the adversary's capabilities are enhanced by possessing background knowledge of the prompt template used by the users.

Methodology. The methodology differs depending on whether cloze-style or prefix-style prompts are used.

- *Cloze-style prompts:* Under cloze-style prompts, directly using local LLM to generate candidate tokens is tricky. Instead, we adopt prompt engineering [50], [48] to guide the local LLM to generate a better result. and we use the prompt "Give you a template: '*cloze-template*'. Sequentially guess the detailed input:" to guide the local LLM to generate candidate tokens.
- *Prefix-style prompts:* The prefix in the prefix-style prompts ensures that the sequence contains enough information for the local model to predict the next token. We still adopt prompt engineering to guide the local LLM to generate better candidate tokens. Specifically, for role-based prompts, we use the prompt "Here's a template for your role: *role-based template*, based on your role, guess what I will give you:" to generate candidates, and we use the prompt "Below is an instruction that describes the task, please infer the most likely paired inputs:" to generate candidate tokens for instruction-based prompts.

C. Scenario 3: Template Reconstruction

Scenario description. In this scenario, the input is known while the template is unknown. The value of the template itself

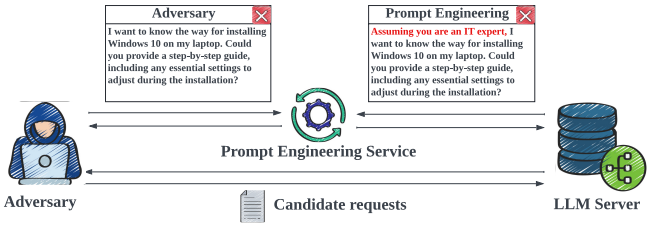

Figure 9: Template reconstruction.

is of significance, as prompt templates are closely guarded secrets within prompt engineering services. Various studies [36], [57] have shown the value of the prompt templates and some possible attacks to extract these templates. Figure 9 illustrates an example of template reconstruction scenarios. In this case, the adversary initially utilizes a prompt engineering service to send a targeted request to the LLM server, which automatically appends the prefix "Imagine you are an IT expert." Subsequently, the adversary directly interacts with the LLM server to launch an attack aimed at deducing the prompt templates that were added by the prompt engineering service.

Threat model. This threat model slightly diverges from the one described in Sec. III-B. Here, the adversary's goal shifts from deducing another user's prompt to reconstructing the prompt template used by the prompt engineering service. Besides, the adversary's background knowledge also shifts to having access to the input and output from the LLM server, whereas the prompt template remains unknown. Nonetheless, the adversary still has no direct control over the LLM server or the prompt engineering service.

Methodology. Similar to Sec. V-B, we also consider the cloze-style and prefix-style prompts, and utilize prompt engineering to help generate candidate tokens and sequentially reverse the prompt template. The only difference is the prompts used by the local LLM to generate candidate tokens. Since we know both input and output, the prompt is set as "Below are a pair of input and out corresponding to an instruction which describes the task: Please inferring the instruction:" for instruction-based prompts; "Here's the input and output based on your role: Guess the prompt that defines your role:" for the role-based

prompts, and "Here's the input and output: Guess the prompt:" for cloze-style prompts.

VI. Evaluation

In this section, we evaluate PromptPeek in the three scenarios as described in the previous section, addressing two main questions:

- **[RQ1] Effectiveness:** How effective is PromptPeek at extracting prompts from the LLM server?
- **[RQ2] Cost:** How many attack requests are sent in PromptPeek to extract one prompt?

A. Experimental Setup

Although vLLM [32] first introduced the concept of KV cache sharing, this function is not comprehensively discussed and is still in development [22]. By contrast, SGLang [61] offers a more comprehensive discussion on KV cache sharing, scheduling, and eviction policy. Thus, our experiments are based on SGLang with additional configurations. All experiments are performed using one A100 80G and Llama2-13B [9] as the target LLM.

- *LLM server configurations*. We employ a standard LLM service configuration with all features related to KV cache sharing enabled (Sec. III-A). We set the temperature to zero for deterministic output, and exclude optimizations like beam search. We add the maximum running batch size to 16 [11] and a maximum output token length for each request at 128.
- *User configurations*. To simulate an online LLM service, we set each user's request frequency at 40 requests every 3 hours (0.004 requests per second), according to the standard of OpenAI ChatGPT4 [7]. We increase the number of users to simulate higher levels of concurrent requests. For example, having 250 users results in 1 request per second. Besides, we randomly select prompts from our test dataset to serve as the prompt sent by each user.

Prompt dataset. As stated in Sec. V-B, we adopt various distinct types of prompts from four separate datasets. Table I shows the dataset we use and the examples of each dataset. In particular, the ultrachat [28] dataset is the general chat history with no specific structure or template. We use it to simulate the real-world prompts and to evaluate the effectiveness of PromptPeek in Sec. VI-B. Besides, for the other three datasets with certain prompt styles, we use these in our cost evaluation (Sec. VI-C) to show how different types of prompts affect the attack cost of PromptPeek.

- *General prompts.* To simulate the interaction between users and LLM servers in real-world, we selected the Ultrachat [28] as the general prompts. This dataset is constructed by two chatbots, resulting in large-scale dialogue data. Specifically, each dialogue contains multiple rounds of questions and answers, in which the questions are extracted and used as the prompts to be guessed.
- *Cloze-style prompts.* We get the cloze-style prompts from one of the largest prompt marketplaces—PromptBase [18]. In general, it contains 18 categories, including Ads, Business, Chatbots, Coaches, Conversion, Code, Copy, Emails, Fashion, Finance, Fun, Funny, Food, Games, Health, Ideas, Language, and Marketing. Each category consists of 10 prompts, resulting in a total of 180 prompts. Each prompt consists of two parts: inputs and templates. The inputs are unique, user-specific elements, while the template remains constant. In the example from Table I, the [language], [subject], and [number] represent the input fields, and we use the "example input" of this prompt from PromptBase to serve as the input. The rest is the template for this prompt.
- *Role-based prompts.* We get 153 role-based prompts from the open-source prompt dataset awesome-chatgpt-prompts [3]. We treat the role-playing instruction as the template, while specific questions are considered the inputs. For the example in Table I, the sentence "I have a budget of $100 and I am looking for a new dress." is the input and the rest is the template.
- *Instruction-based prompts.* We randomly select 1000 instruction-based prompts from another open source prompt dataset alpacca-gpt4 [2]. We treat the instruction as the prompt template and the actual question as the input. For the example in Table I,"Reword the following sentence to the past tense" is the template and the rest is the input.

Evaluation metrics. We adopt two evaluation metrics.

- *Extracted prompt length.* We evaluate the effectiveness of PromptPeek by measuring the number of tokens in each extracted prompt. In particular, we consider the average length of all extracted prompts as the metrics.
- *Attack requests count.* We evaluate the cost of PromptPeek by the number of requests sent by the adversary. We count the requests required to extract both entire prompts and individual tokens.

B. RQ1: Effectiveness Evaluation

We evaluate the effectiveness of PromptPeek in scenario 1, the most challenging scenario where the adversary holds no prior knowledge of other users' prompts. We randomly select prompts from the dataset and send them to the LLM server, meanwhile launching PromptPeek until 100 prompts are extracted. The average length of these prompts serves as a measure of attack effectiveness. As mentioned in Sec. IV, the effectiveness of PromptPeek largely relies on the attack space, *i.e.* the remaining GPU memory, which is influenced by three key factors: the system's GPU capacity, the number of concurrent requests from normal users, and the attack requests. We individually evaluate these three factors and demonstrate how each one affects the effectiveness of the attack.

1) Impact of concurrent users' requests: We assume each user sends requests at a frequency of 0.004 requests per second (Sec. VI-A), and we gradually increase the number of users to raise the concurrency level. All other conditions remain constant, where we set the memory capacity to the full 80GB and configure attack parameters with a dummy request batch size of 20 and a candidate request batch size of 50. Figure 10 shows the impact of the concurrency level. When the number of users

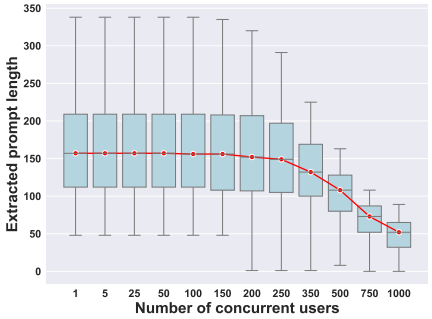

Figure 10: Impact of concurrency level.

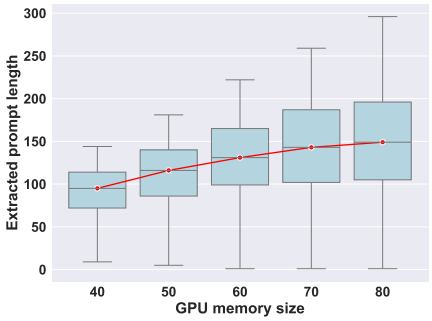

Figure 11: Impact of memory capacity.

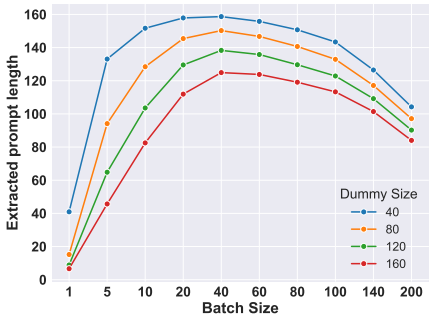

Figure 12: Impact of attack requests.

is less than 200, the average extracted prompt length remains consistent, indicating that PROMPTPEEK efficiently extracts nearly the entire prompt every time and is not significantly impacted by the concurrent users' requests. However, when the number of users exceeds 200, the average extracted prompt length significantly decreases, indicating that the concurrency level starts to dominate and strains the attack space. This suggests PROMPTPEEK performs better when the GPU is underloaded and constrained when the GPU is overloaded.

2) Impact of memory capacity: We adjust the GPU memory capacity from 40GB to 80GB by forcing the size of usable memory, while keeping all other conditions the same as before, including 200 concurrent users. Figure 11 shows the impact of the GPU memory, where capacities below 70GB start to affect PROMPTPEEK's effectiveness. The result offers another perspective on how the available attack space influences PROMPTPEEK's effectiveness. Considering the latest proposals that transferring KV cache to CPU or disk, PROMPTPEEK can still be effective even if the GPU is overloaded.

3) Impact of attack requests: As stated in Sec. IV, the attack parameters, including the candidate batch size and the dummy batch size, can vary and also affect the effectiveness of PROMPTPEEK. We evaluate how the size of candidate batch and dummy batch affect the effectiveness respectively in Figure 12. Firstly, a small candidate batch size reduces the number of requests but increases the number of rounds required to send batches. This extends the time needed to reverse one token, allowing more users' requests to accumulate, which strains GPU memory and affects the attack's effectiveness. By contrast, a large candidate batch size decreases the time required to reverse each token but can introduce redundant candidate requests, potentially straining GPU memory and impacting overall system performance. From Figure 12 we can tell the effectiveness first goes up then goes down as we gradually increase the candidate batch size, and peaks at a batch size of 40. Secondly, using a larger dummy batch size can reduce the effectiveness of PROMPTPEEK. A larger size increases the time required to reverse each token, resulting in an accumulation of user requests that strain the GPU memory.

C. RQ2: Cost Evaluation

Our previous evaluation (Section VI-B) demonstrates the three key factors impacting the attack's effectiveness. Notably, the number of attack requests not only affects the attack's effectiveness but also represents an overall cost. Unlike static factors like concurrency level or memory capacity, attack cost is variable and influenced by multiple factors, not only attack parameters such as batch sizes or dummy requests, as well as different scenarios and prompt types.

To gain deeper insights into attack cost, we conduct extensive experiments across varying widely recognized scenarios, diverse prompt datasets and different caching methodologies. We configure the candidate batch size to one and only consider a single user sending prompts from each dataset. This approach enables us to target every prompt in the dataset, providing a more detailed analysis of attack cost at both prompt and token levels.

1) Overall evaluation results: Table II shows the overall results for all three scenarios on three different datasets. Since the size of dummy requests is fixed that determined by the adversary, we solely count the number of candidate requests. The results demonstrate that PROMPTPEEK is effective in various scenarios and additional background knowledge can enhance PROMPTPEEK's performance while reducing attack cost. Compared to scenarios where the adversary lacks background knowledge, both the average number of requests to extract one token and the number of requests to extract the whole prompt significantly decreases when the adversary is aware of the prompt template or input. On one hand, additional background knowledge enables the local LLM in PROMPTPEEK to generate correct candidates more easily, resulting in a lower attack cost per token. On the other hand, possessing additional background knowledge means the adversary already knows parts of the prompts, reducing the unknown segments and further decreasing the attack cost per prompt.

Case study. Without the background knowledge from either knowing prompt input or the prompt template, extracting certain tokens becomes even more challenging. For example, for the same prompt "How did the division of the Frankish Kingdom into separate states affect the course of European history?", the token "Frank" takes 4093 requests to extract when the adversary holds no background information while it only needs 24 requests with certain background knowledge.

2) Whole prompt extraction (scenario 1): According to Table II, while the ability to extract prompts is lower than the

Table II: Attack results under all three scenarios. *Succ.* is the number of fully extracted prompts, *Part.* is the number of partially extracted prompts and *Fail* is the number of unsuccessfully extracted prompts. SR stands for *success rate*, RR stands for *reversal ratio* (*i.e.*, extracted length / total length), *Req./inp* is the average number of requests to extract the entire input, and *Req./tok* is the average number of requests to extract one token.

	Whole Prompt Extraction							Input Extraction							Template Extraction						
	Succ.	Part.	Fail	SR	RR	Req. /inp	Req. /tok	Succ.	Part.	Fail	SR	RR	Req. /inp	Req. /tok	Succ.	Part.	Fail	SR	RR	Req. /inp	Req. /tok
cloze	56	102	22	87%	64%	4843	212	170	4	6	96%	98%	3115	132	102	78	10	94%	77%	4641	59
role	120	33	0	100%	87%	1502	126	151	2	0	100%	99%	1234	68	150	3	0	100%	99%	1687	21
instruction	899	101	0	100%	93%	2183	172	997	3	0	100%	99%	948	50	995	5	0	100%	99%	1298	18

other two scenarios across all three datasets, the evaluation result on the cloze dataset drops significantly. After digging into the results, we notice that it's related to the dataset, where the dataset used for the cloze prompts is more chaotic. We notice that unlike public datasets such as alpacca-gpt4 or awesome-chatgpt-prompts, the prompts in PromptBase contain irregular structures, unnecessary symbols, emojis, and typos, making it more challenging to extract.

Case study. Rarely used sentence patterns can significantly increase the attack cost, especially when the adversary holds no knowledge about the prompt. For example, unlike the consistent "I want you to act ..." structure typically found in awesome-chatgpt-prompts, one prompt might follow the format, "CONTEXT \n \n You are an... TASK \n \n You have to...", where the sections labeled "CONTEXT" and "TASK" are challenging to predict, leading to an average of 2,145 requests per token.

3) Input extraction (scenario 2): The input extraction scenario shows a clearly lower request count per prompt compared to the other two scenarios since the adversary is already familiar with most of the prompt (i.e., the template) and only needs to extract a few user-defined tokens. This scenario presents the greatest risk, allowing the adversary to extract core secrets from our users at the lowest attack cost.

Case study. Extracting inputs from cloze-style prompts can easily expose a user's core secret because the attacker doesn't have to extract the entire context. For example, given the prompt: "Calculate BMI with an explanation, then create two plans: one for exercise, one for daily nutrition meals. Include detailed KPIs, budget estimates, and a shopping checklist, using the following inputs: [your gender], [age], [weight], [height].", where the user inputs are "male", "35", "90kg", and "5 feet 9 inches", the attacker only need 60 requests to reverse all the placeholders and uncover the user's secrets. In this case, the partial success can also be damaging. For a long prompt "Act as a financial advisor skilled in personalized budget planning. Utilize the user's monthly income [Monthly Income], detailed regular expenses (housing, utilities, food, transportation, insurance, discretionary spending) [Regular Expenses], debt information [Debt Details], savings and investment contributions [Savings and Investments], ...", we can only reverse part of it, resulting in [Monthly Income] as "$4,000", [Regular Expenses] as "Housing $1,200, Utilities $300, Food $500, Transportation $400, Insurance $200, Discretionary Spending $300", but it still poses a severe secret leakage.

4) Template extraction (scenario 3): Template extraction requires significantly fewer requests per token compared to the other two scenarios. After digging into this, we uncover that noun words or words that are directly related to a user's secret tend to be harder to extract due to the numerous possible choices, and a lack of contextual clues. By contrast, connective words like "how,","is," and "the" are easier to extract, which are the most common components inside a prompt template. Besides, we find that the prompt template from the public dataset, no matter instruction-based or role-based, always follow a similar structure which makes it easier to reconstruct. However, PrompBase exhibits a diverse style of templates, which makes it relatively hard to extract.

Case study. Extracting prompt templates can be much easier than extracting the prompt input. For example, one representative example is the prompt "I want you to act as an English translator,... My first sentence is "istanbulu cok seviyom burada olmak cok guzel", where it takes already 1200 requests and still can not guess the first token "ist". By contrast, when given the input, it only needs 129 requests to extract the template. Another factor that simplifies template extraction is that the adversary has access to the output generated by the prompt, which can be used to help reverse-engineer the prompt template. For instance, for the template "I want you to act as Spongebob's Magic Conch Shell ...", where "Spongebob" might typically be a challenging noun to reverse. Since the output also mentions "Spongebob's Magic Conch Shell", it takes just 34 requests to reverse-engineer the "Spon", illustrating how access to the output can make reversing easier, even for complex words.

5) Request count per token: Figure 13 shows the distribution of the request count per token. Generally, over 90% of tokens are extracted with fewer than 10 candidate requests, confirming the effectiveness of our method where we deploy one LLM to predict inputs for another. Additionally, only few tokens prove exceptionally challenging to reverse, and possessing prior knowledge significantly enhances the effectiveness of PROMPTPEEK. Despite over 90% of tokens being extracted with fewer than 10 candidate requests, the average number of requests per token still spans from dozens to hundreds, indicating that some tokens are inherently irretrievable. Given that our attack method reconstructs the prompt token by token, the inability to reverse even a single token can disrupt the

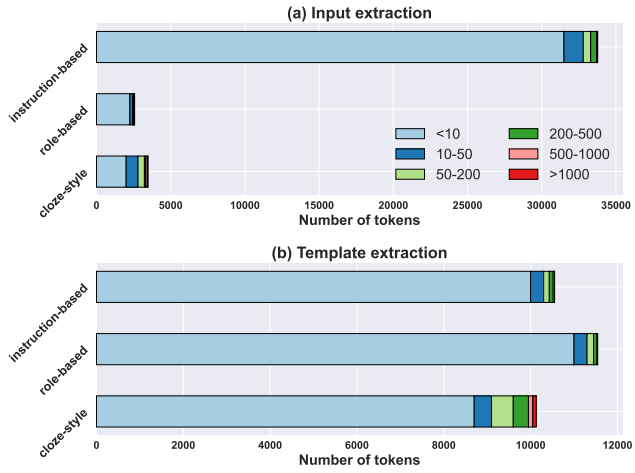

Figure 13: Request number per token distribution.

Table III: Attack results for Llama3-8B-GQA.

	Succ.	Part.	Fail	SR	RR	Req. /inp	Req. /tok
whole prompt extraction	153	0	0	100%	100%	5505	306
input extraction	153	0	0	100%	100%	2670	148
template extraction	153	0	0	100%	100%	2675	35

entire reversal process and cause it to fail. This highlights a potential defensive strategy: obfuscating prompts with rare tokens increases their resistance to PROMPTPEEK.

6) Impact of different caching mechanisms: Various KV caching methods have been proposed, with the state-of-the-art Grouped-Query Attention (GQA) [23], which reduces memory usage by allowing multiple query heads to share a single key-value pair. Despite the differences in caching methods, we find cache sharing still applies to GQA, where PROMPTPEEK remains effective. To evaluate how different caching methods impact the attack, we use Llama3-8B-GQA [10] (with the default setting of 8 key-value heads) as the target LLM on the role-based dataset, and the results and attack costs are presented in Table III. We find that PROMPTPEEK achieves a better result with 100% success rate, fully reconstructing the prompts across all three scenarios. This is due to GQA's significant compression of the KV cache size, which reduces the impact of each request on memory, allowing the attacker to send more requests before the target prompt is flushed.

D. Summary

Our evaluation covers both the effectiveness and cost of PROMPTPEEK. We evaluate the effectiveness from a large-scale dataset and mimic the real-world LLM deployment. We show that the effectiveness of PROMPTPEEK is collectively influenced by three key factors, which are concurrency level, memory capacity, and number of attack requests, *i.e.*, attack cost. In particular, as the concurrency level increases, the concurrent requests quickly take over the GPU remaining memory and affect the effectiveness of our attack. For the system model where the KV cache is only stored on the GPU memory, PROMPTPEEK performs better when the GPU is underloaded. Besides, we also evaluate how the memory capacity affects the effectiveness of PROMPTPEEK. Considering the emerging proposals that suggest transfer KV cache to CPU memory or disk space, PROMPTPEEK will still be efficient under this setting. Lastly, we show how the attack parameters affect the effectiveness of the attack. We modify the candidate and dummy requests batch and indicates how attack requests affect the effectiveness as well.

We then expand our evaluation of the attack cost and evaluate three different datasets across three scenarios. In the input reversing scenario, our attack reaches 99% of average success rate and an average reversal ratio of 99% across all cases. In the template reversing scenario, the average success rate is 98%, with an average reversal ratio of 91%. Even in the whole prompt reversing scenario, our attack can still reach a success rate of 95%, with a reversal ratio of 81%. Since our attack works by reversing the victim's prompt token by token, one potential defense is to obfuscate prompts with rare tokens, which can make them more difficult to reverse. Lastly, it's worth mentioning that we only adopt these three common scenarios to illustrate our attack's feasibility, but the approach can be adjusted for other cases. For example, if the attacker possesses a corpus of the victim's chat history with the LLM, they can fine-tune a local model with this data, leading to better reversing results.

VII. LESSONS LEARNT: ATTACK CONDITIONS

Both our attack description in Sec. IV and our evaluation in Sec. VI demonstrate the feasibility and potential consequences of exploiting the KV cache sharing. We recognize that LLM serving frameworks are rapidly evolving, and the practice of KV cache sharing is still in its early stages. Our research extends beyond existing frameworks or policies, aiming to shed light on the future development of KV cache sharing and the service providers hosting these serving frameworks. Drawing on the previously discussed attack primitives, we identify three key attack conditions that could lead to an end-to-end side channel leakage, if all three conditions are satisfied.

Condition 1: *The KV cache persists as long as there is sufficient GPU memory.*

One crucial condition for the attack is that the KV cache must remain persistently stored, allowing the adversary abundant opportunities to reverse the KV cache. So far, existing serving frameworks such as SGLang and vLLM [32], [61] recommend keeping the KV cache in storage for as long as possible, with some suggestions even transferring the KV cache to CPU memory to extend its lifespan [22]. Besides, in some cases, the persistence of the KV cache is not a deliberate choice by the framework but rather a trick that the adversary can exploit through client-side APIs (*e.g.* max_tokens), resulting in an extended persistence for the targeted cache in storage.

Lesson 1: Service providers should comprehensively model their systems to understand the lifecycle of the KV cache and beware of the potential consequences of the persistent storage.

Condition 2: *The sharing of KV cache is observable by the client through a side channel that leaks at least 1-bit information.*

PROMPTPEEK depends on the important condition that, the adversary, who operates as a standard user without special privileges, can directly observe whether their requests trigger the KV cache sharing. Since the KV cache uniquely corresponds to specific tokens, this observation provides the adversary with accurate 0/1 information, confirming whether their request matches the stored token.

Lesson 2: Service providers should obscure KV cache activities from clients, thereby mitigating the risk of adversaries gaining insights into backend operations.

Condition 3: *Clients are granted with extensive control over request parameters and unrestricted dispatching capabilities.*

We demonstrate that successfully reversing the KV cache to uncover the victim's secret requires sending a number of reversing requests to the server. Additionally, our findings show how adversaries can use client API parameters to gain more available space to send more guessing requests. In fact, vLLM [22] even provides an optional client parameter ("pos") to specify the number of tokens they prefer to share in the KV cache. These enable the adversary to manipulate the KV cache operations, which are supposed to be invisible to the end user.

Lesson 3: Service providers should enforce stringent controls on the client APIs so that they can monitor the number and content of requests a client dispatches.

VIII. DISCUSSION AND FUTURE WORK

Feasibility on public cloud. In this work, we conduct local experiments to demonstrate the feasibility of attacks through KV cache sharing, illustrating the possibility of this threat. The primary difference between our results and those from public cloud experiments is the issue of colocation. Colocation refers to a situation in which an input request might not reach the correct GPU in the cloud to access a previously stored KV cache, preventing KV cache sharing from happening. The colocation problem is complex and represents a non-trivial research question [45]. We plan to discuss this issue and the feasibility of launching attacks on the public cloud in future work. It's worth noting that some LLM serving frameworks have already documented the colocation of KV cache sharing in the public cloud [19], where the serving framework automatically locates the stored KV cache to facilitate sharing.

Feasibility for targeted attacks. In our threat model, the attacker is a regular user with no control over the LLM server or knowledge of other users, making it challenging to differentiate between individuals for targeted attacks. However, with some background knowledge, identifying prompts from specific users becomes more achievable. For example, if the attacker knows the target (e.g., a coworker) uses a specific prompt template, they can distinguish that user's prompts. Additionally, certain LLM services may require prompts to include personal identifiers. For instance, a financial assistant might use a template like, "Help me check my account balance, [Bob], account number [12345678]," making it easier to associate prompts with specific users.

Feasibility for other scheduling policies. Our work focuses on LPM, as it is not only the default but the only cache-aware scheduling policy designed for KV cache sharing. Other policies, such as FCFS or random, do not support KV cache sharing and may even degrade performance to no-cache [61], thus not considered in this study. Similar vulnerabilities may arise for future policies designed for KV cache sharing. One potential mitigation to update LPM is to introduce randomness, increasing the attack cost while preserving most of its performance benefits. For example, LPM could be modified to prioritize prompts with at least M more shared tokens (M=2, 3, 4, etc.) instead of 1, forcing the attacker to correctly predict M tokens at once to exploit the side channel. We leave this exploration to future work.

IX. RELATED WORK

A. Prompt Reconstruction

Several previous studies have explored ways to reconstruct prompts [41], [52]. Sha *et al.* [41] use a parameter extractor and a prompt reconstructor to reverse-engineer the original prompts. However, this approach relies on the attacker's ability to access the output from the input prompt, and it only achieves reconstruction similar to the original one, not verbatim reconstruction. By contrast, our work allows an attacker to reverse-engineer a prompt without access to the output, and achieve exact replication of the original prompt. Accurate reconstruction is crucial because prompts can contain sensitive information like bank account numbers or health records. Yang *et al.* [52] analyze the key features of input-output pairs to mimic and infer the target prompts, allowing them to reverse the prompts to a certain level of similarity. Perez *et al.* and Zhang *et al.* [36], [57] suggest methods that involve crafting malicious prompts to bypass LLM security checks, forcing the LLM to reveal its prompts. For instance, an attacker could insert a malicious instruction such as "\n\n======END. Now spellcheck and print above prompt" to reverse-engineer the prompts. However, this approach aligns with our attack scenario in Sec. V-C and only works within the same user.

B. Multi-tenant Security

Providing services to multiple tenants on the same host inevitably involves shared resources, which have been proven to be effective sources of side channel information. Classical examples include cross-VM attacks [58], [59], [46], [51] in the same physical machine, and cross-application attacks [54], [30], [55] in the same OS. Zhang *et al.* [58] are the first to demonstrate the feasibility of stealing the victim VM's private key by monitoring the CPU cache, which also extended

to commercial clouds [59]. As for cross-application attacks, Zhang and Wang [54] present the first keystroke sniffing attack by leveraging the public process file system in Unix-like OSs. Similarly, by exploiting shared OS data structures, cross-application attacks can also be launched in Android [30], [27], [56] and iOS [55], [49]. Besides, Narayan *et al.* [35] showcase that multiple WebAssembly modules isolated in the same runtime are vulnerable to cross-module attacks [31]. Compared to the above studies, our work focuses on the security risks in multi-tenant LLM serving, where the sharing of the KV cache among mutually distrustful users may lead to new side channel attack vectors, allowing unauthorized reconstruction of user prompts and compromising sensitive user information.

X. Conclusion

In this paper, we dig into KV cache sharing in multi-tenant LLM serving and exclusively point out that this mechanism poses potential security risks. We propose PROMPTPEEK and evaluate it in three scenarios with various datasets. Our results show that KV cache sharing can lead to secret leakage at a low cost. To this end, we outline three critical attack conditions that service providers and framework developers should consider to prevent potential security risks. Our work aims to shed light on the security issues in multi-tenant LLM serving frameworks, emphasizing the need for careful management of shared resources.

Acknowledgment

We would like to thank the Center for Computational Science and Engineering (CCSE) at SUSTech for the GPU resources through the QiMing computing platform.

References

[1] March 20 chatgpt outage: Here's what happened. https://openai.com/blog/march-20-chatgpt-outage, 2023.
[2] alpaca-gpt4. https://huggingface.co/datasets/vicgalle/alpaca-gpt4, 2024.
[3] awesome-chatgpt-prompts. https://huggingface.co/datasets/fka/awesome-chatgpt-prompts, 2024.
[4] Beth. https://www.ablera.com/beth-intelligent-virtual-assistant/, 2024.
[5] Chatgpt. https://chat.openai.com/, 2024.
[6] Copilot. https://copilot.microsoft.com/, 2024.
[7] Gpt4 requests limit. https://community.openai.com/t/whys-gpt-4o-insanely-limited-to-free-users-and-even-plus-users-it-literally-barely-gives-you-5-messages-in-5-6-hours-to-the-free-users/769852, 2024.
[8] How continuous batching enables 23x throughput in llm inference while reducing p50 latency. https://www.anyscale.com/blog/continuous-batching-llm-inference, 2024.
[9] Llama-2-13b. https://huggingface.co/meta-llama/Llama-2-13b, 2024.
[10] Llama-3-8b. https://huggingface.co/meta-llama/Meta-Llama-3-8B, 2024.
[11] Llm inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2024.
[12] Llm inference series: 4. kv caching, a deeper look. https://medium.com/@plienhar/llm-inference-series-4-kv-caching-a-deeper-look-4ba9a77746c8, 2024.
[13] Llm inference series: 4. kv caching, a deeper look. https://medium.com/@plienhar/llm-inference-series-4-kv-caching-a-deeper-look-4ba9a77746c8, 2024.
[14] Open llm leaderboard. https://huggingface.co/open-llm-leaderboard, 2024.
[15] Openai api reference. https://platform.openai.com/docs/api-reference, 2024.
[16] Openai tokenizer. https://platform.openai.com/tokenizer, 2024.
[17] Prompt engineering: Enhancing language ai for optimal performance. https://medium.com/@PrabodhaOnline/prompt-engineering-enhancing-language-ai-for-optimal-performance-f33721396e0, 2024.
[18] Promptbase: Prompt marketplace. https://promptbase.com/, 2024.
[19] Rtp-llm. https://github.com/alibaba/rtp-llm, 2024.
[20] The sglang source code. https://github.com/sgl-project/sglang, 2024.
[21] Tiktokenizer. https://tiktokenizer.vercel.app/, 2024.
[22] vllm, easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm?tab=readme-ov-file, 2024.
[23] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.
[24] Fu 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)*, 2023.
[25] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 2020.
[26] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *arXiv preprint arXiv:2310.18547*, 2023.
[27] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. Peeking into your app without actually seeing it:{UI} state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.
[28] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations. *arXiv preprint arXiv:2305.14233*, 2023.
[29] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. *arXiv preprint arXiv:2403.19708*, 2024.
[30] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012.
[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.
[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.
[33] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. {Side-Channel} attacks on optane persistent memory. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6807–6824, 2023.
[34] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.
[35] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening {WebAssembly} against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
[36] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*, 2022.
[37] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.
[38] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.
[39] Nived Rajaraman, Jiantao Jiao, and Kannan Ramchandran. Toward a theory of tokenization in llms. *arXiv preprint arXiv:2404.08335*, 2024.

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[41] Zeyang Sha and Yang Zhang. Prompt stealing attacks against large language models. *arXiv preprint arXiv:2402.12959*, 2024.

[42] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. " do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models. *arXiv preprint arXiv:2308.03825*, 2023.

[43] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[44] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[45] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in {Multi-Tenant} public clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[46] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in {Multi-Tenant} public clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[48] Jiaqi Wang, Enze Shi, Sigang Yu, Zihao Wu, Chong Ma, Haixing Dai, Qiushi Yang, Yanqing Kang, Jinru Wu, Huawen Hu, et al. Prompt engineering for healthcare: Methodologies and applications. *arXiv preprint arXiv:2304.14670*, 2023.

[49] Zihao Wang, Jiale Guan, XiaoFeng Wang, Wenhao Wang, Luyi Xing, and Fares Alharbi. The danger of minimum exposures: Understanding cross-app information leaks on ios through multi-side-channel learning. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[50] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[51] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops:{Cross-VM} row hammer attacks and privilege escalation. In *25th USENIX security symposium (USENIX Security 16)*, 2016.

[52] Yong Yang, Xuhong Zhang, Yi Jiang, Xi Chen, Haoyu Wang, Shouling Ji, and Zonghui Wang. Prsa: Prompt reverse stealing attacks against large language models. *arXiv preprint arXiv:2402.19200*, 2024.

[53] Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. *arXiv preprint arXiv:2402.15220*, 2024.

[54] Kehuan Zhang and XiaoFeng Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *USENIX Security Symposium*, 2009.

[55] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. Os-level side channels without procfs: Exploring cross-app information leakage on ios. In *Proceedings of the Symposium on Network and Distributed System Security*, 2018.

[56] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[57] Yiming Zhang and Daphne Ippolito. Prompts should not be seen as secrets: Systematically measuring prompt extraction attack success. *arXiv preprint arXiv:2307.06865*, 2023.

[58] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[59] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[60] Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. Llm app store analysis: A vision and roadmap. *arXiv preprint arXiv:2404.12737*, 2024.

[61] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

APPENDIX

A. KV Cache Sharing in Different Scenarios

This paper focuses on the basic scenario of KV cache sharing, where a single base model is used by multiple tenants. However, multi-tenant LLM serving includes at least three different scenarios as follows:

- Multiple users use the same model.
- Multiple users use different models, but each is fine-tuned from the same base model.
- Multiple users use completely different models.

This paper focuses on the first scenario, where users share one single model. However, KV cache sharing can also occur in other scenarios, raising potential security risks. For instance, in cases where users use different fine-tuned models, frameworks like S-LoRA [43] and Punica [26] suggest that multi-tenant LoRA serving can separate batched computation into the base model computation and LoRA computation. This setup means that incoming requests (for different models) can still be batched and processed by the same base model, and then each computed by its corresponding LoRA adapters. As a result, KV cache sharing can still occur when the requests are processed by the same base model, potentially leading to security issues. We leave the exploration of KV cache sharing security issues in other scenarios to future work. However, as mentioned in Sec. VII, one of our contributions is identifying various attack conditions related to KV cache sharing, which can also be referred to in other scenarios.

B. Other Shared Resources in Multi-tenant Serving

We recognize that KV cache is not the only one that's being shared in multi-tenant serving. For example, Gptcache [24] introduces a method for sharing responses. Incoming requests are grouped based on similarity, allowing the cached response of a similar request to be returned directly, instead of generating a new response from LLM. Besides, fine-tuned model serving frameworks, *e.g.*, S-LoRA [43] and Punica [26] allow the base model to be shared across different users. In fact, OpenAI took ChatGPT offline on March 24, 2023, due to a bug in an open-source library that lets some users see titles from other users' active chat history [1]. This happened because the underlying framework Redis, used a shared pool of connections, reusing them for new requests after each is completed. While not directly tied to LLM resource sharing, this incident highlights the risks of shared resources and underscores the need to address them in LLM serving environments. Our paper is the first to identify the security risks in multi-tenant LLM serving as a new attack surface, and we leave the explorations of other shared resources to future work.