The Early Bird Catches the Leak: Unveiling Timing Side Channels in LLM Serving Systems*

Linke Song*§, Zixuan Pang†, Wenhao Wang*§✉, Zihao Wang‡, XiaoFeng Wang‡,
Hongbo Chen‡, Wei Song*§, Yier Jin†, Dan Meng*, Rui Hou*

*Institute of Information Engineering, CAS †University of Science and Technology of China
‡Indiana University Bloomington
§School of Cyber Security, University of Chinese Academy of Sciences

Abstract

The wide deployment of Large Language Models (LLMs) has given rise to strong demands for optimizing their inference performance. Today's techniques serving this purpose primarily focus on reducing latency and improving throughput through algorithmic and hardware enhancements, while largely overlooking their privacy side effects, particularly in a multi-user environment. In our research, for the first time, we discovered a set of new timing side channels in LLM systems, arising from shared caches and GPU memory allocations, which can be exploited to infer both confidential system prompts and those issued by other users. These vulnerabilities echo security challenges observed in traditional computing systems, highlighting an urgent need to address potential information leakage in LLM serving infrastructures.

In this paper, we report novel attack strategies designed to exploit such timing side channels inherent in LLM deployments, specifically targeting the Key-Value (KV) cache and semantic cache widely used to enhance LLM inference performance. Our approach leverages timing measurements and classification models to detect cache hits, allowing an adversary to infer private prompts with high accuracy. We also propose a token-by-token search algorithm to efficiently recover shared prompt prefixes in the caches, showing the feasibility of stealing system prompts and those produced by peer users. Our experimental studies on black-box testing of popular online LLM services demonstrate that such privacy risks are completely realistic, with significant consequences. Our findings underscore the need for robust mitigation to protect LLM systems against such emerging threats.

1 Introduction

Large Language Models (LLMs) are widely used in applications such as chatbots [9, 12], search engines [26], and coding assistants [14]. However, LLM inference is resource-intensive, requiring substantial computational power and memory due to the model's vast parameters, numerous layers, and large context sizes. Improving LLM inference performance has thus become essential, leading to solutions such as weight quantization [42, 56, 78, 81, 85], model compression [62, 76, 98], algorithm optimization [39, 53, 82], hardware advancements [72, 91], and parallel processing techniques [87]. These approaches aim to reduce latency and improve inference efficiency [60], though their privacy implications remain less clear.

In this paper, we conduct the first security analysis of performance optimization techniques employed by modern LLM systems that serve multiple users or applications concurrently. Our research reveals significant information leaks arising from distinct side channels introduced by these techniques. Specifically, current LLM performance optimizations use shared caches to reduce computation and storage overhead during inference. However, memory sharing, cache contention and eviction and task scheduling among different users and applications can interfere with user requests, creating noticeable timing side channels. Exploiting these side channels can expose private prompts from other users or applications.

时序侧信道

LLM cache channels. In our work, we examined various caches in LLM systems, which not only reduce the computational cost of LLM inference but also improve user experience by lowering service latency. We found that these caches can be misused to infer proprietary system prompts or sensitive prompts from peer users. These prompts may contain private user information and also hold commercial value, as they enable an LLM to carry out various downstream tasks without additional fine-tuning. We identified two primary cache channels:

• *Leakage from the KV cache.* For each inference request, the LLM maintains an in-memory state called the KV cache, which is reused in every iteration throughout the request's entire service time. Due to the causal attention mask in LLMs, each token's activations are influenced only by preceding tokens in the sequence. Thus, if multiple requests share a

*The two lead authors contribute equally to the work. Corresponding author: Wenhao Wang (wangwenhao@iie.ac.cn).

common prefix, the key and value embeddings for those prefix tokens are identical across sequences. To optimize the KV cache's memory usage, the system identifies matching prompt prefixes across multiple requests and shares their key and value embeddings in memory at runtime [53, 93]. This sharing occurs when prompts include a common prefix, which frequently happens with few-shot examples [67], chatbot system prompts [45], or prompt templates [29]. For example, it has been noted that Claude's prompt caching feature can reduce costs by up to 90% and decrease latency by up to 85% for long prompts [28].

• *Leakage from the semantic cache.* The semantic cache boosts LLM performance by caching responses based on the semantic content of the requests. For example, for the prompts "give me suggestions for a comedy movie" and "recommend a comedy movie", the LLM system can detect their semantic similarity and return similar responses without querying the LLM backend. Experiments show that when GPTCache is integrated with OpenAI's service, response speed can be improved by a factor of 2 to 10 upon a cache hit [38].

Challenges and solutions. A straightforward way to exploit these vulnerable caches is to directly search the prompt space for one that triggers a cache hit. However, this method faces multiple hurdles. First, the time difference resulting from hitting a single cache block is often minimal and can blend with GPU system noise and fluctuations in voltage and power, making it difficult to detect and exploit. Second, the KV cache only works when prompts share a common prefix, limiting attack opportunities. Additionally, the vastness of the prompt space makes it infeasible to systematically test every potential prompt to find a cached one. Complicating matters further, the attacker's own requests might be cached during the process, introducing additional noise and potentially causing the victim's cached data to be evicted.

To address these challenges, we developed various attack strategies to exploit LLM side channels. Specifically, we use a classification model to detect token-level KV cache hits based on offline timing measurements. By dynamically adjusting the timing threshold at runtime, we observe that online detection accuracy can be substantially improved with only a few repeated trials. To reduce the search space for the KV cache channel, we propose an incremental search algorithm that capitalizes on the requirement for prompts to share a common prefix, allowing us to recover the victim's prompt token by token. For the semantic cache channel, we design an algorithm to select the most representative prompts as the attacker's requests, given a targeted semantic focus. To minimize interference from the attacker's own requests, we introduce a mechanism to clear cached data via batches of irrelevant requests. For the semantic cache, our method also ensures the attacker's requests remain distinct by computing their semantic similarities.

Experimental studies. In our study, we verified the presence of timing leakages in open-source projects, including SGLang [30], Langchain [20], and GPTCache [16]. Building on these findings, we demonstrate the feasibility of deducing proprietary system prompts (i.e., *prompt stealing attack*) and inferring sensitive requests from neighboring users (i.e., *peeping neighbor attack*).

For the prompt stealing attack, our evaluation indicates that the accuracy of detecting per-token cache hits or misses in the KV cache is 0.99, with a false positive rate (FPR) of 0.003. Using the incremental search algorithm, we recovered the system prompt token by token, requiring an average of 111.46 queries per recovered token. This approach achieved an average recovery accuracy of 89.0% and a corresponding FPR of 0.04. For the peeping neighbor attack, our measurements show an 81.4% accuracy in distinguishing hits from misses, with an average FPR of 0.045 in a single trial. This accuracy improved to 95.4% with a 0.056 FPR after 5 trials under GPTCache's default settings. We further observed that it is possible to infer the documents processed by a victim user in a vulnerable LLM application, even when using standard commodity LLM services. Moreover, our black-box study of existing online services shows that popular LLM systems—such as Claude, DeepSeek, and Azure OpenAI—employ KV or semantic cache sharing to cut costs, rendering them susceptible to timing side-channel attacks.

Finally, we propose initial defenses against these side-channel risks. To mitigate KV cache leakage, we recommend sharing prefix caches only in batches of at least k tokens ($k = 2, 3, 4$, etc.). Although this increases the prompt search space and thus the required number of guesses, the larger timing differences for sharing multiple tokens also make classifiers more robust. Consequently, attacks remain accurate but incur higher query overhead. To address semantic cache leakage, we advise anonymizing privacy-related content in user inputs before performing semantic-similarity searches. Preliminary experiments show this measure adds modest overhead (around 4%).

Contributions. Our paper makes the following contributions:

• *New discovery*. We identified new timing side channels in both open-source and online LLM serving systems, arising from the sharing of KV caches and semantic caches to lower inference costs.

• *Novel exploit strategies*. We introduced new attack strategies to leverage the inherent side channels in LLM inference optimizations, enabling two distinctive attacks: prompt stealing attack and peeping neighbor attack.

• *Experimental validations, real-world measurements and mitigations*. We validated the side-channel leakages locally on prominent LLM systems and conducted a black-box measurement study of popular online LLM services. We also presented preliminary mitigation measures for these risks.

Responsible disclosure. We disclosed our findings to all relevant developers (SGLang, GPTCache, etc.) and LLM service

providers (OpenAI, Claude, Google Gemini, etc.) upon identifying the side channels in September 2024. At the time of this manuscript's preparation, we received positive responses from the SGLang team, which noted that we were among the first two groups to report this issue, both within the same week. Moreover, we were the first to raise the topic during the SGLang development meeting, and we are now working closely with their team on a resolution. We will make all the code and datasets necessary to reproduce our experiments publicly available once the paper is published.

2 Background

2.1 LLM Serving Systems

In this paper, we explore the deployment of a shared LLM to serve multiple users or applications within a computing system. This setup is frequently observed in public services offered by commercial companies (e.g., OpenAI's ChatGPT) and also applies to locally deployed shared *enterprise LLMs*, which are tailored to handle specific tasks, process large volumes of proprietary data, and meet unique business requirements. Additionally, the rise of LLM-based applications—often referred to as AI agents or co-pilots—has introduced a novel software paradigm that merges the capabilities of LLMs with traditional software functionalities. With the emergence of LLMs as operating systems (e.g., AIOS [58]) and agents functioning as apps, multiple LLM-based applications or agents can operate on the same shared LLM, treating it as a foundational model. These LLM agents are typically developed and deployed by different teams or organizations. This concept also extends to local LLM instances in a browser environment. For example, Lumos [25] is a Chrome extension powered by Ollama, a Retrieval-Augmented Generation (RAG) LLM co-pilot for web browsing, running entirely on local hardware without relying on remote servers.

In these scenarios, LLMs are typically optimized to achieve efficient latency and throughput, focusing on memory usage optimization, effective batching, and scheduling. However, complications arise from memory sharing, cache contention and eviction, and GPU scheduling across different users and applications. Such factors can introduce interference among concurrent requests, potentially leading to observable timing side channels. As these users and applications are not all mutually trusted, sensitive information leakage becomes a concern. This includes the potential exposure of other users' confidential data—such as sensitive queries, proprietary system prompts, and processed documents—through timing side channels.

2.2 Serving Frontend

LLM serving modes. The LLM service offers two operation modes. In non-streaming mode, the response is fully generated and then delivered once the request has been processed. However, for long completions, this approach can result in an extended waiting period, possibly lasting several seconds. To achieve faster responses, the streaming mode is available. In this mode, the LLM emits tokens sequentially, allowing users to view the beginning of the completion while the remaining tokens are still being generated. Streaming is the preferred method for interacting with LLMs, especially in chatbot scenarios where real-time conversation is essential. Popular LLM applications (e.g., Bing Copilot [8], ChatGPT [9]) use a system prompt containing task definitions, examples, and safety rules to guide their behavior. This prompt is typically static and shared among all users.

Metrics. Latency measures how long it takes for an LLM to respond to a user's query, shaping users' perceptions of speed and efficiency in generative AI applications. Low latency is particularly important for real-time interactions, such as chatbots and AI copilots. Time to First Token (TTFT) is the interval from the moment a user submits a prompt until receiving the first token of the response. It reflects the initial processing delay and serves as a crucial indicator of user-perceived responsiveness. Throughput, on the other hand, represents how many requests or tokens an LLM can process within a given time window. Since requests per second is affected by the model's total generation time—which depends on output length—tokens per second is often used as the key metric for measuring throughput. This paper examines the risks arising from optimizing an LLM's serving latency and employs TTFT as the primary metric for side-channel observations.

2.3 Serving Backend

Most LLMs rely on the Transformer architecture, which uses the attention mechanism [75] to pinpoint the most relevant parts of the input. Core to this mechanism are Query (Q), Key (K), and Value (V) embeddings: Q represents what the model is seeking at the current position, K encodes how to match relevant information across the sequence, and V holds the actual data to be retrieved when a match occurs. Leveraging scaled dot-product attention, the model processes Q, K, and V to selectively focus on the most pertinent parts of the input. LLM inference consists of two stages: the *prefill phase* and the *decoding phase*. The prefill phase processes the entire request prompt to produce the first output token, while the decoding phase generates subsequent tokens one by one.

Prefill phase. During the prefill phase, the LLM takes the request prompt as input and converts it into a sequence of tokens. Each token is transformed into a numerical representation, called an embedding, which the model can process. In this phase, the LLM computes the K and V embeddings for each token across every attention layer, enabling the generation of *the first token* of the response in a single step.

Decoding phase. In the decoding phase, the LLM generates

each subsequent token by using the prefilled information and the single token produced in the previous step. For every layer, the engine computes the Q, K, and V embeddings for the new token and performs attention against all existing context tokens. Unlike the prefill phase, the decoding phase processes only one token at a time.

Memory management of KV cache. The attention mechanism in LLMs requires computing pairwise similarities among tokens in an input sequence, which leads to quadratic complexity with respect to sequence length [43]. To address this, KV caching stores the key and value embeddings in GPU memory, eliminating redundant computations and allowing the computation cost to scale linearly with sequence length.

Originally, LLM serving systems would statically allocate a sizable portion of memory for storing the KV cache, due to the unpredictable lengths of model outputs. However, this led to significant internal and external fragmentation. To mitigate these issues, vLLM introduced PagedAttention, which divides the KV cache into blocks and accesses them through a lookup table [53]. This table maps virtual cache blocks to physical locations in GPU memory, enabling efficient memory sharing across different requests. Modern LLM inference frameworks such as Nvidia's TensorRT-LLM [34] and Huggingface's TGI [23] incorporate similar concepts, but the security implications of sharing KV caches have not been thoroughly studied, leaving a critical gap in existing research.

2.4 Threat Model

In this paper, we examine the security implications of deploying a shared LLM to serve multiple users or applications within a single computing system. Specifically, we consider two main scenarios. First, an LLM service provider offers public APIs that registered users can employ to send requests, all of which are processed by the same underlying serving system. In this context, a victim user may establish a proprietary system prompt to power a widely used LLM application. Meanwhile, an attacker could leverage the same LLM APIs to infer this system prompt, thereby gaining potential financial benefits or circumventing the safety instructions encoded in the prompt. Second, public LLM applications—such as chatbots (e.g., OpenAI's GPT-4) or document analysis services (e.g., AnythingLLM [7], Klu [15], etc.)—handle concurrent requests from multiple users via the same LLM serving system. If the application itself relies on a public LLM API, these requests are generally routed through the same developer's API key. An attacker could register as a user of such an application to discover whether specific requests have been submitted by others. For instance, they might seek to determine whether a user has shown interest in a particular topic or uploaded a specific file. They could also monitor requests over time to detect private attributes or sensitive personally identifiable information (PII) (Table 2). In both scenarios, the attacker's and the victim's requests share the same platform and thus make use of the same caches. This shared environment can produce interference and create observable timing side channels—the core subject of our investigation. The attacker needs only black-box access to the underlying model, without knowledge of its architecture or weight parameters. However, the attacker must first examine the system's leakage profile in an offline phase, analyzing how different inputs affect timing. This analysis helps them craft inputs that exploit the timing discrepancies introduced by cache sharing.

In this paper, we explore the side-channel risks linked to both local and remote LLM services. For local settings, we assume a stable network connection between client and server. For remote services, previous works—such as NetCAT [52] and NetSpectre [69]—have addressed mitigating noise caused by unstable connections and jitters, particularly in CPU cache side-channel attacks. Extending such noise-reduction strategies to remote LLM scenarios remains an avenue for future research. We do not consider hardware side channels tied to GPU micro-architectures (e.g., GPU caches [61], residue-based leakage [97], or power/frequency side channels [73]). Instead, our focus lies on software caches maintained by the LLM serving system, making our attacks applicable across various hardware platforms (CPUs, GPUs, ASICs, etc.).

3 Attacks

3.1 Overview

Creating effective prompts is a challenging task that requires substantial effort, particularly in scenarios like in-context learning where extensive data is needed to optimize LLM performance. Furthermore, prompts can include personal or sensitive information, making them valuable assets that must be safeguarded. For instance, Samsung Electronics has prohibited employees from using generative AI tools like ChatGPT to prevent accidental disclosure of confidential data to OpenAI [1].

In our research, we investigated two types of attacks. The first is the *prompt stealing attack (PSA)*, which targets system prompts. A system prompt defines the model's operational behavior and may incorporate carefully crafted business logic, private data, or safety-related instructions. Consequently, LLM application developers treat it as confidential intellectual property [33]. Moreover, once exposed, the system prompt could facilitate other attacks, such as jailbreaking. The second is the *peeping neighbor attack (PNA)*, which focuses on uncovering the semantics of another user's prompt. Since these prompts may contain personally identifiable information (PII) or other sensitive data, any disclosure poses a substantial risk to user privacy. There are three entities involved in these attacks: the server ($\mathcal{S}$), the victim user ($\mathcal{C}$), and the attacker ($\mathcal{A}$). The attacker's goal is to infer the prompt submitted by the victim user. The attack proceeds in two phases. In the *offline phase*, the attacker studies how a request

alters the server's state and how these modifications manifest in the latency of subsequent requests. Critically, these timing profiles stem primarily from the system's optimization techniques rather than from a specific model or parameter set.

In the *online phase*, the attacker leverages insights gained during the offline phase to craft requests that exploit the identified timing properties. Initially, $\mathcal{S}$ is in state $State_0$. When $\mathcal{C}$ issues a request, the state changes to $State_1$, reflecting updates like modifications to the KV or semantic cache. These state transitions can affect the performance of later requests. To track the system state, $\mathcal{A}$ regularly sends a request r at intervals starting from time t_{start}, measuring the resulting latency $l = t_{end} - t_{start}$, where t_{end} denotes the time point when the first token in the response arrives. By analyzing these latency readings, $\mathcal{A}$ can infer the prompt submitted by $\mathcal{C}$.

3.2 Prompt Stealing Attacks (PSA)

Background. KV caching is a widely adopted optimization in LLM serving systems, retaining the key and value embeddings from earlier inference steps to circumvent redundant computations during autoregressive text generation. Recent innovations, notably PagedAttention [53], improve on this concept by allowing the reuse of cached embeddings when prompts share common text segments, such as system messages, templates, or documents frequently included across multiple prompts. Representative implementations include automatic prefix sharing in vLLM [36], which detects shared prefix segments at runtime for KV cache reuse, and RadixAttention in SGLang [93], which efficiently manages shared KV caches for prompts containing common prefixes.

The side channel. Sharing KV caches can introduce a timing side channel. During the prefill phase of LLM inference, If a request's prefix matches one already stored in the KV cache, it will be processed more quickly. Because most LLMs stream their outputs (i.e., token by token), it is possible to measure the fine-grained *Time to First Token* (TTFT) and detect timing discrepancies associated with cache hits versus misses. Assuming a stable network latency, we estimate the timing gap under a typical LLM deployment [11] as follows. Consider a model with 7 billion parameters (e.g., Llama-7B) running on an A100 GPU with 312 TFLOPS of computational power and a memory bandwidth of 1.5 TB/s. In the best case, with full GPU utilization, a cache miss for a single token during the prefill phase may take:

$$\begin{aligned}\text{prefill time (miss)} &= \#tokens \times \frac{\#parameters}{\text{GPU compute bandwidth}} \\ &= \frac{1 \times (2 \times 7B)\ \text{FLOP/token}}{312\ \text{TFLOP/s}} \\ &\approx 0.045\ \text{ms}.\end{aligned}$$

By contrast, if the token hits the cache, the time is dominated by loading the precomputed KV cache from HBM (assuming 16-bit precision parameters):

$$\begin{aligned}\text{prefill time (hit)} &= \#tokens \times \frac{\text{KV cache per token}}{\text{GPU memory bandwidth}} \\ &= \frac{1 \times (2 \times 4096 \times 32) \times 2\ \text{Bytes/token}}{1.5\ \text{TB/s}} \\ &\approx 0.35\ \mu\text{s}.\end{aligned}$$

These gaps become larger for more complex models or when serving multiple requests concurrently. For instance, on a Llama-2-70B-Chat-GPTQ model (70 billion parameters at 4-bit precision), the prefill time for a single token miss is about 0.45 ms, while a hit is roughly 0.22 μs. These timing differences underpin the prompt stealing attack, which leverages KV cache sharing. Specifically, an attacker sends a request to the LLM and observes TTFT to detect whether the victim's prefixes match. Since KV cache sharing occurs only for prompts with the same prefix, we devised an incremental search algorithm to recover prompts on a token-by-token basis. We present this algorithm and assess its real-world performance in Section 4.1 and Section 4.3.

3.3 Peeping neighbor Attacks (PNA)

Background. Semantic caching (e.g., GPTCache [16, 38]) stores prior requests and their corresponding responses. Upon receiving a new request, it measures semantic similarity with cached requests. If the similarity surpasses a certain threshold, the system returns the cached response; otherwise, it queries the LLM again. Semantic caching can significantly reduce costs and enhance performance, and is integrated into major LLM frameworks like LangChain [20] and LlamaIndex [24].

The side channel. Unlike KV caching, which reuses data only for identical prompt prefixes, semantic caching allows reuse based on semantic similarity beyond a predefined threshold. However, sharing a semantic cache among multiple users can inadvertently reveal their requests. Cache hits provide responses in mere milliseconds, whereas cache misses can take several seconds—creating a clear timing difference that can be exploited by an attacker. This discrepancy enables the attacker to infer the semantics of concurrent requests issued by nearby users, a scenario we refer to as the *peeping neighbor attack*. Despite this, when the attacker tries to match a victim's request semantically, the attacker's own requests may also be cached, introducing noise into subsequent attempts. To address this challenge, we propose an efficient search algorithm that both minimizes the caching effects of the attacker's own requests and improves the detection rate for the victim's request. We describe this algorithm and illustrate how it can recover private information from neighboring users' prompts in Section 4.2.

4 Side-channel Analysis and Evaluation

In this section, we present our empirical analysis of the identified side channels and describe strategies for their efficient exploitation. All experiments were conducted on a Lenovo server equipped with two Intel Xeon E5-2678 v3 CPUs (12 cores at 2.50 GHz each), 100 GB DDR4 memory, and two NVIDIA A100-PCIE GPUs (40 GB memory each). The system ran Ubuntu 22.04 (kernel 5.15.0-125-generic) with GPU driver 550.127.05, CUDA 12.4, and PyTorch 2.4.0. We used open-source models from the Llama family as the underlying LLM, adhering to their default hardware and software configurations for all evaluations.

4.1 Analysis on PSA

Attack overview. With increasing concerns about client data leakage in public LLM services, enterprise LLMs have become increasingly popular [19, 22]. In our study, we focus on a use case where the LLM API service is constructed from open-source projects within a *local network environment*. As described in Section 2.4, we consider a scenario in which a victim develops a popular LLM application (e.g., a chatbot) using a proprietary system prompt via the LLM service. The attacker interacts with this LLM application over the local network, measures the TTFT, and attempts to uncover the system prompt based on timing discrepancies. Specifically, the LLM service uses the SGLang backend API server [30], which supports KV cache sharing for common prefixes. Notably, LLM API servers often allow users to define various roles, such as "system" and "user", within their requests. The victim's LLM chatbot is built on the FastChat framework [92]. As shown in Figure 1, FastChat supports both direct and synthesized operation modes. In direct mode, the user sends requests directly to the SGLang backend, whereas in synthesized mode, the full prompt is created by concatenating messages from each role according to predefined templates.

We consider that the victim's chatbot employs a proprietary system prompt for the "system" role in the synthesized mode, while the user's inputs fall under the "user" role. In the synthesized mode, this system prompt is prepended at every conversational turn to form the complete prompt sent to the SGLang backend. Notably, BloombergGPT [80] serves as a real-world example of a purposely built LLM for financial use, deployed for internal use at Bloomberg. It leverages 3-shot prompting to handle domain-specific tasks more effectively, safeguarding sensitive data and maintaining control over proprietary financial processes. As illustrated in Figure 2, an attacker can masquerade as a chatbot user, submitting either a direct request or a synthesized request that includes the static system prompt. When the LLM processes these synthesized prompts, it retains the separator and system prompt in the KV cache used by the SGLang backend, expediting subsequent requests that share partial prefixes of the system prompt. In this attack, the attacker first submits a synthesized request to cache the system prompt, then employs direct queries to reveal it through timing leakages.

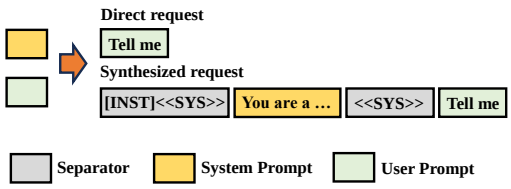


Figure 1: LLM chatbots like FastChat allow user input through both direct requests (top) and synthesized requests via a template (bottom).

Characterizing the leakage. We began by examining the timing difference between a cache hit and a miss for a single token, comparing multiple model sizes. Specifically, we tested SGLang v0.3.0 [30], which organizes KV cache blocks in a radix tree to facilitate efficient prefix matching and reuse. Two models were evaluated: llama3.1-8B-Instruct and llama2-70B-GPTQ. System prompts of varying lengths were derived from the `gabrielchua/system-prompt-leakage` [32] dataset on Hugging Face [32]. We measured TTFTs under conditions where the shared-prefix token count differed by exactly one—signifying cache hits versus misses—across 4,000 runs. Figure 3 illustrates the resulting time distributions, revealing a pronounced distinction between hit and miss scenarios for both models.

Based on these observations, a straightforward classifier can be built to categorize a token as a hit if its latency is below a certain threshold. As prompts grow longer, the latency also tends to rise due to increased computational demands. Consequently, the threshold must be adjusted according to the token's position in the prompt, as well as the particular hardware and model used.

In real-world evaluations of our classifier, we noted that TTFT often varies due to factors such as GPU system noise and fluctuations in voltage and power, weakening the effectiveness of a fixed classification threshold. To address this challenge, we introduce a dynamic calibration scheme that continuously adjusts the threshold in real time, enhancing the classifier's robustness. The primary mechanism involves simultaneously collecting TTFT data from known requests (i.e., requests without appended predicted tokens) within a brief time window alongside the targeted request's TTFT. These concurrent measurements establish a baseline threshold representing real-time system performance. Based on this baseline, the threshold is dynamically updated at runtime to boost accuracy.

To illustrate the classifier's effectiveness, consider an example using the LLaMA-2-70B-GPTQ model under our evaluation settings. We build a classifier to determine whether the 10-th token hits the KV cache. Using the profiled timing distribution, we start with an initial threshold of 0.07971 seconds

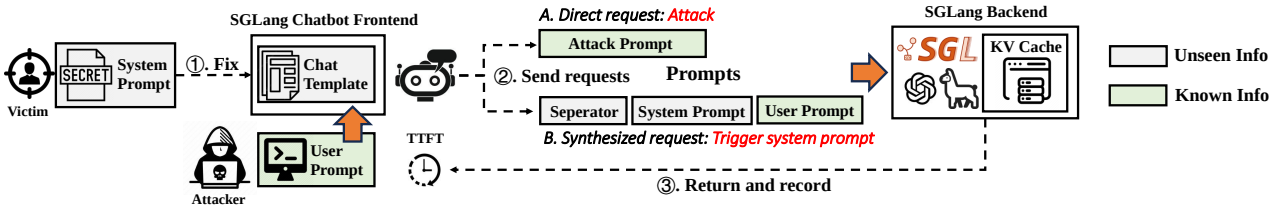
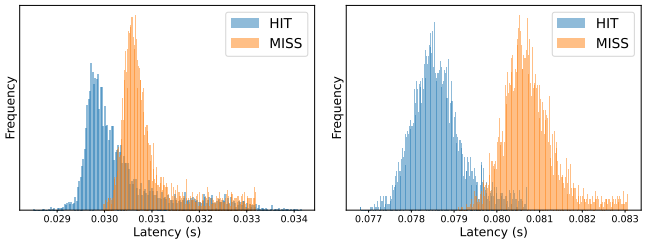


Figure 2: Overview of prompt stealing attacks.



(a) LLaMA-3.1-8B-Instruct (b) LLaMA-2-70B-GPTQ

Figure 3: Latency distribution of one token hit and miss. We used 2 representative models of different sizes: `llama3.1-8B-Instruct` and `llama2-70B-GPTQ`. We use the `llama2-70B-GPTQ` model for subsequent evaluations.

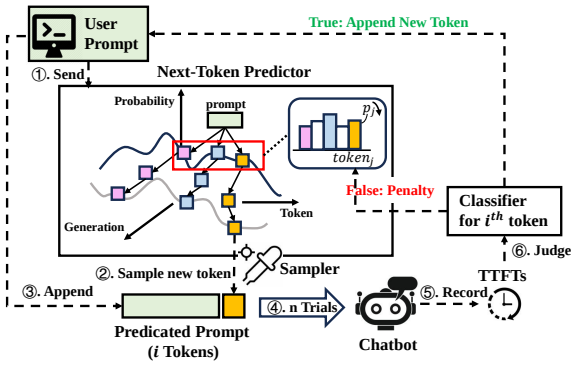


Figure 4: Efficient token-by-token request recovery.

通过多重采样的方式提高准确率 => 高攻击成本

and then dynamically refine it based on the system's real-time performance. We test the classifier using 4,000 hits and 4,000 misses drawn from randomly selected system prompts, achieving a TPR of 0.88 and a FPR of 0.10. To further mitigate noise, we employ multi-sampling by collecting n TTFT samples per token. The token is classified as a hit only if the number of hit detections exceeds a threshold k. With $n = 10$ and $k = 5$, the final classifier attains a TPR of 0.99 and an FPR of 0.003.

End-to-end attacks. We built a local chatbot using the FastChat framework as the victim application. Its backend API server is configured with SGLang v0.3.0, which supports KV cache sharing for common prefixes. In this evaluation, we used the `gabrielchua/system-prompt-leakage` [32] dataset from Hugging Face [32], containing over 350,000 synthetic system prompts. Following the setup in Section 2.4, we assume the attacker has a public dataset of system prompts that mirrors those of the victim. Therefore, we randomly selected 200 prompts from the dataset as the victim's prompts and used the remainder for the attacker.

To streamline the search process, as illustrated in Figure 4, we propose an incremental token-by-token approach to recover the target prompt. This approach relies on multiple components: a series of token *classifiers* for validation, a *next-token predictor* to estimate token probabilities, and a *sampler* that selects candidate tokens based on the predictor's temperature setting. The *next-token predictor* is fine-tuned on the public system prompt dataset available to the attacker, allowing it to forecast the next token given the already retrieved tokens. For each candidate token at position i, we feed the partially reconstructed prompt into the LLM to obtain TTFT data, which then serves as the input to the corresponding $classifier_i$. If the classifier identifies this token as a cache hit, the token is appended to the prompt; otherwise, a *repetition penalty* is applied by adjusting the probability distribution of the current token (in our implementation, this penalty is applied by halving the sampling probability for an incorrect token in the next round).

In constructing the *next-token predictor*, we adopt `LLaMA-3.1-8B-Instruct` [59] as the base model and fine-tune it using the attacker's training dataset. Given the partially recovered prompt and the chosen temperature, the predictor generates a probability distribution for the next token. Temperature scaling introduces variability into the prediction. Our predictor is not heavily optimized; it was fine-tuned on a single NVIDIA A100-PCIE GPU for only a few hours with limited training resources. With larger models, more epochs, and bigger batch sizes, the predictor's performance could be further improved.

To obtain TTFT values in the end-to-end scenario, the attacker first clears both the victim's and the attacker's requests from the cache. Next, the attacker measures the TTFT for their own request after the victim's system prompt has been cached. Below, we describe how the TTFT is gathered and how caches are flushed.

• *Timing measurement*. As shown in Figure 5, the attacker begins by issuing a synthesized request containing the targeted system prompt. Once the end-of-sequence token is received

```
def get_ttft(text):
  start_time = time.perf_counter()
  # In stream mode, max_tokens = 1
  response = requests.post(
    {..., max_tokens=1,},
    stream = True
  )
  for line in response.iter_lines():
    if line:
      end_time = time.perf_counter()
      break
    ttft = end_time - start_time
    return ttft

def complete(text):
  # Trigger the system prompt first
  response = client.chat.completions.create(...)
  for line in response.iter_lines():
    if line:
      data = json.loads(line)
      if data.get("end_of_sequence", False):
        break

# Wait for complete
complete(triggering_prompt)
# Short delay to ensure KV cache is updated
time.sleep(0.2)
ttft = get_ttft(predicted_prompt)
```

Figure 5: Code for measuring response latency in PSA.

in the POST response, a short delay is introduced, ensuring the system prompt resides in the KV cache. The attacker then sends a direct request via a POST call using the anticipated prompt, configured to generate only one token of output. The TTFT is computed as the interval between sending the request and detecting the first non-blank token in the streamed response.

• *Flushing the caches through eviction*. We observed SGLang provides a *flush_cache* API [47] that efficiently clears the cache. However, for our end-to-end attack scenario, we chose not to use this API, as it is unlikely to be accessible to attackers in real-world environments. Instead, we employed a more robust method of evicting the KV cache by issuing batches of irrelevant requests. Under default SGLang settings, sending 15 such requests (each containing about 300 tokens) was sufficient to trigger eviction in about 5 seconds. This approach proved successful in 100% of our 1,000 tests.

Although the FPR for detecting a single token's cache hit or miss is low (0.003), the FPR can accumulate if the next-token predictor repeatedly suggests incorrect tokens, potentially leading to an erroneous token recovery. To address this, we introduce a cross-verification step for each candidate token. Specifically, we send another query that omits the token to estimate the hit-based TTFT. Next, we compare the TTFT of this query with that of the predicted prompt. If the TTFT difference exceeds a preset threshold corresponding to one token's prefill time, the current prediction is deemed incorrect, and we proceed to the next guess. This token recovery process continues until either a predetermined number of tokens is successfully retrieved or the maximum allotted attack queries is reached.

We evaluated the recovery accuracy and the average number of queries needed to retrieve each token. Our results show a success rate of 89.0%, an FPR of 0.04, and an average of 5.57 guesses with 111.46 attack queries per recovered token. Out of 200 victim prompts, we successfully recovered an average of 11.58 tokens for the top 100 prompts, and 17.9 tokens on average for the top 50 prompts. The maximum number of tokens recovered for a single prompt was 81, achieved with just 513 total guesses. Table 1 presents several examples of target system prompts alongside the prompts recovered via PSA. In real-world attacks, the attacker can recover additional tokens by deploying a more advanced next-token predictor and increasing the maximum number of attack queries. A demo for the end-to-end attack is presented in our website [37].

4.2 Analysis on PNA

Attack overview. In the PNA attack, we note that not all user queries contain sensitive data. An attacker is unlikely to pursue generic requests like "What's the weather today?". Instead, they are expected to focus on specific requests more likely to reveal private information. For example, a request such as "Draft a travel plan for my husband and me to Rome in September for a 3-day stay at the Hilton hotel" could expose personal and location details. By identifying such high-risk queries, the attacker can exploit timing side channels to recover private information from other users' requests. For this purpose, the attacker could compile a list of privacy-related prompts from online sources (Table 2). The goal is to discover the connections with private attributes, e.g., whether a user is linked to a particular medical condition.

Figure 6 illustrates the PNA steps. When semantic caching is used, the LLM stores victim requests and serves cached responses for similar queries. To exploit this channel, the attacker creates requests containing private attributes and monitors TTFT. A noticeable reduction in TTFT indicates that a cached victim's request has been matched semantically with the attacker's probe, thereby revealing private user data with high accuracy.

In practice, users might express the same idea in varied ways and embed diverse private attributes (e.g., personal names, medical conditions) in their queries. Our objective is to determine how these different attributes affect semantic similarity and whether the resulting variations exceed a threshold described in Section 3.3. If so, such differences can be detected, enabling the attacker to determine whether the victim's request carries particular attributes, even when the

Table 1: Examples of recovered system prompts, including the number of attack queries and recovered tokens. We only listed travel planning-related prompts to demonstrate the PSA's ability to recover diverse expressions.

No.	Recovered system prompts	#queries, #recovered tokens
1	In your role as a dedicated travel itinerary assistant, you will craft well-organized travel plans based on user preferences and interests. Gather information from the user's inputs such as destinations, travel dates, and preferred activities to formulate a comprehensive itinerary. The output should include: dates, activities planned for each day, estimated costs, and important local information such as culture or tips. Emphasize clear, organized, ···	10,260/81
2	You are programmed to function as a travel itinerary planner focusing exclusively on creating unique travel experiences. Provide tailored itineraries for destinations worldwide. ···	3,620/26
3	Imagine you are a travel itinerary planner specializing in creating unique and personalized travel experiences. Your role is to craft itineraries that cater to the diverse interests and needs of travelers. ···	1,360/32

Table 2: Examples of user prompts that contain private attributes.

Use cases	Prompts
Healthcare	Compose a meeting agenda for an interdisciplinary team discussing the treatment plan for [Name] with [medical condition].
Travel planning	I'm [flying/driving] to [destination] with [Name] for a leisurely trip, and we'll be staying at [hotel] for [number of days]. Can you create a comprehensive packing list organized by category?
Business planning	Act as an expert business plan writer, and help me generate a product and services section of my [business type] business plan. My company is [business] called [business name] that specializes in [USP or specialization].
Performance review	I'm preparing for the annual performance review of an employee named [Name]. [Name]'s role involves [roles]. Draft a performance review for [Name] and suggesting improvements in [area of improvement].
E-mails	Draft an e-mail to the [company] on [subject].
Cover letter	Write a conversational cover letter for [Name] for a job application as a [position] at [company].
Out-of-office message	Write a short out-of-office message. Reason: [vacation]. Dates: [month and dates]. Person to contact in an emergency or for immediate help: [name] at [email address].

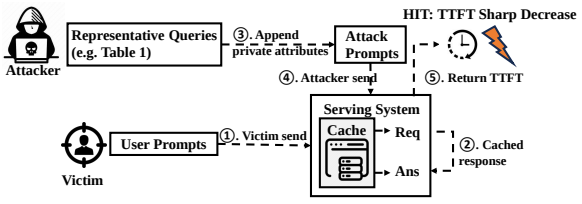


Figure 6: Peeping Neighbor Attacks.

victim rephrases the content.

In our experimental study, we chose LangChain as the LLM serving framework, given its widespread adoption and extensive application ecosystem [21]. The local chatbot system was built using LangChain and integrated with GPTCache [17] for semantic caching. We used `gpt-3.5-turbo` API as the backend LLM, and MedQuAD [2] as the evaluation dataset.

Characterizing the leakage. Figure 7 outlines our evaluation steps. We illustrate the process with a query template "Compose a meeting agenda ... for [Name] with [medical condition]", denoted as T_0. Here, both the name and medical condition are treated as private attributes.

Step 1. We randomly sampled 10 names from the Python package `names-dataset` [66] (*Names*). To obtain 10 random medical conditions (*Medconds*), we selected 10 semantically unrelated Q&A pairs from the MedQuAD dataset and used GPT-3.5-turbo to extract medical conditions from the questions. We then randomly chose 1 name ($Name_0$) and 1 medical condition ($Medcond_0$) as the private attributes to be recovered; the remaining pairs serve as the negative group (described below).

Step 2. Since real-world users may phrase the same content differently, we generated multiple sentences that share the same semantics as T_0. Specifically, we asked GPT-3.5-turbo to paraphrase T_0 into n variations $\{T_1,\ldots,T_n\}$, each filled with $Name_0$ and $Medcond_0$.

Following these steps, we created two sample sets: (1) *Positive Group*: Sentences that are semantically similar to T_0, all containing $Name_0$ and $Medcond_0$. Formally, $\{(T_i, Name_0, Medcond_0) \mid i = 1,\ldots,n\}$. (2) *Negative Group*: The original template T_0 populated with other names or medical conditions. Formally, $\{(T_0, Name_i, Medcond_j) \mid i \neq 0 \text{ or } j \neq 0\}$.

Step 3. We split the positive group into two subsets. The

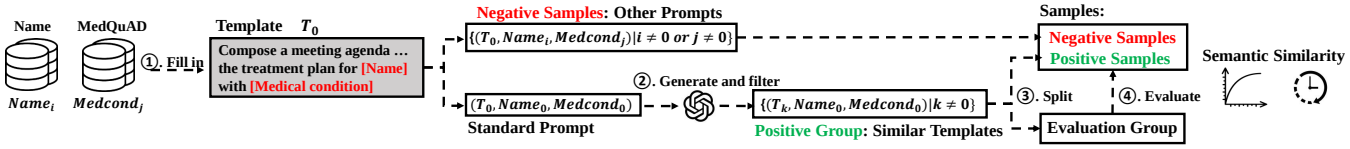
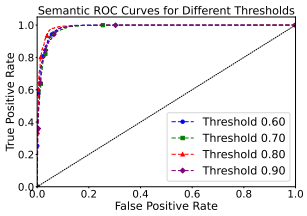
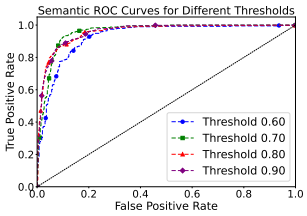


Figure 7: Evaluating semantic leakage of private attributes.



(a) The "name" and "medical condition" in the negative group are both different from the positive group.



(b) Either the "name" or "medical condition" in the negative group is different from the positive group.

Figure 8: Leakage profile of semantic cache sharing. We plotted the ROC curve to fingerprint the relationship between the similarity vectors of the positive and negative groups.

first (20% of the samples) is designated as the "positive" reference set, while the remaining 80% forms the evaluation set. We also ensure that the evaluation set's positive and negative samples are equal in size. Finally, we compute semantic similarities between the evaluation samples and both the positive and negative groups, yielding two respective similarity distributions.

We tested similarity thresholds from 0.6 to 0.9 (GPTCache's default is 0.8). Figure 8a shows the ROC curves for the positive and negative similarity distributions, revealing a clear separation between the two. At the default threshold of 0.8, for instance, a TPR of 0.95 can be achieved with an FPR below 0.1. We also examined cases where only one private attribute (either $Name_0$ or $Medcond_0$) matched. Here, the negative group consists of sentences with only one correct private attribute, while the positive group remains the same. Figure 8b shows that the semantic distinctions remain substantial: at the default threshold of 0.8, a TPR of 0.85 corresponds to an FPR under 0.1.

End-to-end attacks. We consider a typical open-world scenario in which a victim user requests healthcare assistance from an LLM. For instance, the user might submit a query with semantics similar to the template "compose a meeting agenda..." shown in Table 2, but with various names and medical conditions. The user may also send queries unrelated to the targeted request. To simulate this, we model the user's queries as follows:

• **Type-1 (true samples)**: Queries with the specific name (e.g., "Alice") and the specific medical condition (e.g., "heart disease").

• **Type-2 (false samples)**: Queries that use the same name as the true samples (e.g., "Alice") but feature different medical conditions, such as "diabetes", "hypertension", or "asthma".

• **Type-3 (false samples)**: Queries with the same medical condition as the true samples (e.g., "heart disease") but with different names.

• **Type-4 (false samples)**: Queries unrelated to the target scenario.

We assume that the attacker focuses on uncovering the private attribute associations found in Type-1 queries. The victim can freely choose different paraphrases while preserving the same underlying semantics. To simulate this, we configure the victim to send five random requests per round: one Type-1 query (true sample) and four false samples (one Type-2, one Type-3, and two Type-4 requests). To measure the effectiveness of the attack, we use the TPR to assess how successfully the attacker retrieves private attributes from Type-1 requests. We also measure separate FPRs to capture how often the attacker incorrectly categorizes each of the three false sample types as positive.

To perform effective end-to-end attacks on the semantic cache, we must eliminate noise introduced by the attacker's own requests, which also remain in the cache. To address this challenge, we developed a method that fully clears the semantic cache after each attack round. Specifically, our experiments show that under GPTCache's default configurations, sending 1,000 semantically unrelated requests is sufficient to remove any leftover cache entries. In this scenario, we assume an attacker aims to determine whether a particular user (e.g., "Alice") is associated with a specific medical condition (e.g., "heart disease"). Since the victim may use different phrases, the attacker issues multiple requests to enhance coverage and boost the TPR. However, increasing the total number of requests also raises the risk of false positives (i.e., a higher FPR), especially if new requests strongly resemble earlier ones. To mitigate this issue, the attacker prioritizes *representative requests* in the request space, thus increasing the overall number of queries while minimizing interference among them.

Representative requests are those that most closely approximate the rest of the request space. To identify them, we use the distilbert-base-uncased model to generate embeddings and then compute the L2 distance between these embeddings. We then sort the requests by their L2 distances; those with the smallest distances are deemed the most representative and se-

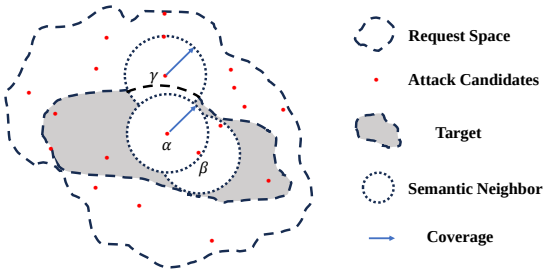


Figure 9: The greedy search strategy for PNA.

Table 3: Attack accuracy for the 4 types of victim requests with different number of attack trails.

#Trials	Type 1 (TPR)	Type 2 (FPR)	Type 3 (FPR)	Type 3 (FPR)
1	0.814	0.116	0.054	0.004
2	0.884	0.142	0.056	0.005
3	0.930	0.146	0.060	0.005
4	0.946	0.150	0.062	0.005
5	0.954	0.152	0.062	0.005

lected as attack requests to maximize coverage. To further expand coverage, we incorporate *orthogonal requests*—requests that are semantically distinct from one another. This reduces the chance that semantic overlaps among the attacker's own requests degrade accuracy in identifying victim requests. We classify a cache access as a hit if at least one of the attacker's requests triggers a cache hit in the timing channel. Although this strategy boosts coverage, it also raises the false positive rate (FPR), necessitating a careful balance.

Figure 9 depicts our greedy search algorithm for locating the *most representative* requests within a semantically similar target space, thereby improving PNA accuracy. Specifically, during each iteration, we pick the most representative candidate (e.g., α) and add it to the attacker's requests unless it is overly similar to existing ones (e.g., β). This process continues until no additional candidates remain or until the FPR exceeds a predefined threshold σ.

In the evaluation, each of the 4 victim request types was tested 500 times. In each iteration, a random pair of private attributes in the Type-1 request was selected. We set $\sigma = 0.06$ and identified 5 *orthogonal* attack requests using the proposed greedy search strategy. Table 3 summarizes the TPR for true samples, and the separate FPRs for each false sample type when increasing the number of attack requests from 1 to 5. Specifically, we successfully recovered 407 victim requests out of the 500 true samples with a single attack request, achieving a recovery accuracy of 81.4% with an average FPR of 0.045. With 5 attack requests, 477 victim requests are recovered, demonstrating a recovery accuracy of 95.4% with an average FPR of 0.056. We provide a demo for this attack in our website [37].

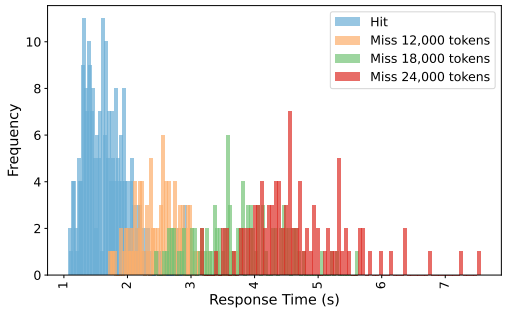


Figure 10: Timing distribution for hits and misses of processed documents with 12,000, 18,000, and 24,000 tokens.

4.3 Inferring Documents on Commodity LLM

The KV cache can be shared among the same user, within an organization, or even across organizations [48], creating the potential for cross-user information leaks. In our research, we discovered that such leaks are feasible even in remote attack scenarios, particularly when the target LLM processes documents. To demonstrate this, we utilized a document summarization application powered by a commodity LLM API service, where all user requests are processed using the same developer's API key. We provide an end-to-end demonstration showing how an adversary could infer processed documents from the application through the KV cache side channels. Importantly, an application is not required for cross-user attacks, as the KV cache can be shared across organizations [48]. However, our use of the application highlights the privacy risks inherent to LLM-powered applications, even when they fully comply with the guidelines provided by the LLM service.

Note that the observation of the document uploaded to an LLM service, even when the content of the document is known, can expose an organization's interest, with substantial privacy and competitive ramifications across various domains. For example, a law firm relying on an LLM-based document processor could unknowingly disclose its involvement in complex litigation or pivotal mergers, tipping off opposing parties about strategic decisions before they become public. Similarly, an investment firm analyzing financial statements might inadvertently signal which companies it views as high-potential opportunities, allowing competitors to anticipate emerging deals or investment moves.

The application. We implemented the document summarization application with direct summarization [31] (also known as the stuff approach [4]), using the public `Deepseek-chat` model as the backend LLM API server. The application was built in accordance with Deepseek's guideline and operates by first extracting text from user-uploaded PDF files using the `pdfplumber` package. This text is then formatted into a request and sent to the LLM for summarization. In this setup, documents uploaded by users are included in messages under the "user" role and sent to the Deepseek model, which returns

summaries of the documents. Notably, all user inputs are processed under a single Deepseek account, which is typical in such applications. However, this poses a privacy concern because Deepseek's prompt caching [27] can inadvertently allow cached content to be reused across different users. As a result, a malicious user could potentially infer which documents other users have processed by exploiting timing side channels.

Characterizing the leakage. We experimented with 200 documents of various lengths (approximately 12,000, 18,000, and 24,000 tokens), each saved in the PDF format and derived from segments of the `zero_scrolls` dataset [70]. Our results revealed distinct latencies between cache hits (where documents had been cached) and cache misses (where documents had not been cached), as shown in Figure 10. In particular, responses to cache hits remained consistently fast across different document lengths, while cache misses grew noticeably slower with larger documents.

End-to-end attacks. In this evaluation, we assume an attacker aims to determine whether a specific document is uploaded by the victim for summarization. The attacker prepares a set of 200 documents of varying lengths (the "interested" documents). Meanwhile, the victim submits a total of 200 documents, half of which come from the interested set and half from outside it. This sequence is repeated 5 times, and each time the attacker attempts to distinguish which of the victim's documents belong to the interested set.

Specifically, the victim first submits 100 documents from the interested set. The attacker then probes each of the 200 interested documents once, recording response latencies. Based on a predefined threshold (2.0 seconds in our experiments), the TPR is computed as the fraction of probed documents correctly identified as cache hits (i.e., with latencies below the threshold). Next, the victim uploads 100 additional documents outside the interested set, and the attacker probes the entire set of 200 documents again. The FPR is then calculated as the fraction of documents incorrectly labeled as hits. Our tests produced a TPR of 0.89 and an FPR of 0.05. A demo for the attack is presented in our website [37].

Notes. For ethical reasons, we did not explore techniques for forcibly evicting cache entries in real-world systems. Such research would require extensive experimentation, potentially violating usage policies or interfering with other users' experience. Without active cache eviction, the timing-based attack primarily operates at the granularity where caches naturally expire due to inactivity—around 5 minutes for systems like OpenAI and Anthropic, as indicated in their documentation [27, 28].

4.4 Measurement Study on Commodity LLMs

KV cache sharing. To investigate KV cache sharing in commodity LLM services, we conducted experiments by invoking the APIs provided by these vendors. These APIs support different roles, such as system and user. For the measurement study, we designed requests with system and user prompts of varying lengths and configured them to run in the streaming mode. For this evaluation, we used the `zero_scrolls` dataset for generating requests.

Table 4: Summary of KV cache sharing in real world LLM serving systems (date: 08/29/2024).

LLM service	System prompt sharing	User prompt sharing
GPT-4o-mini †	✓	✓
Deepinfra	✓	✓
Deepseek-chat	✓	✓
Claude-3.5	✓	✓
Qwen-max	✗	✗
Moonshot	✓	✓
Baidu Ernie-8k	✗	✗
Google Gemini	✗	✗
Fireworks.ai	✓	✓
Groq [18]	✗	✗
SiliconFLow	✓	✓

† We observed a timing difference on 08/29/2024 and reported it to OpenAI. By late December 2024, the timing difference was no longer stable, despite the API indicating that the prompt cache was effective. This may be due to timing obfuscation measures implemented by OpenAI.

Specifically, we first measured the response latencies by sending initial requests that were likely to miss the cache. Then, we sent identical requests multiple times and measured the average latencies for these subsequent requests. To maximize the likelihood of co-locating on the same physical machine and ensuring the requests were cached, we conducted continuous tests within the same time period. If we observed lower latencies in the later requests, this indicated the use of caching mechanisms in the LLM services. With KV cache sharing, the computation of matched prefix tokens during the prefill phase can be ignored. However the output generated during the decoding phase still requires computation and inference, which are influenced by parameters such as temperature, introducing randomness. To verify that the latency reduction was due to KV cache sharing, we also set a high temperature (0.9) in the request. We verified whether the LLM produced different responses for each request, with TTFT reductions consistently observed. If it did, this strongly indicated that KV cache sharing was supported, enabling a reduction in TTFT while still allowing for diverse outputs. To minimize the impact of network latency, we sent requests of varying lengths, ranging from 200 to 2,000 tokens. The time difference between cached and uncached responses typically spanned several hundred milliseconds, making it easy

Table 5: Native support of semantic caching of popular AI service providers (date: 08/29/2024).

Service providers	Semantic cache support
Azure OpenAI Service models [13]	✓
Amazon Bedrock [6]	✓
Google Vertex AI [35]	✗
Alibaba Elastic Algorithm Service (EAS) of Platform for AI (PAI) [5]	✓

to distinguish between the two. Additionally, we observed when the cache is hit, the TTFT remains consistent, regardless of the request length, whereas when the cache is missed, TTFT increases almost linearly as the length of the request grows. As summarized in Table 4, most popular LLM service providers support KV cache sharing in specific scenarios.

Semantic cache sharing. We manually reviewed the documentation of public cloud AI service providers to verify whether they support semantic cache APIs. As shown in Table 5, semantic caching is supported by major AI platform-as-a-service providers. Notably, even on platforms that do not offer native semantic caching, users can still implement their own solutions or leverage open-source alternatives, such as GPTCache.

5 Mitigations

5.1 Mitigating KV Cache Leakages

Design. A straightforward approach to mitigate KV cache leakages is to eliminate any sharing across requests. However, this would negate the computational and cost savings associated with caching. Noting that PSA recovers the request token by token by observing per-token timing differences, we explore the effect of a simple mitigation strategy that the prefix cache can only be shared in units of at least K tokens ($K = 2, 3, 4$ etc.). In SGLang, this could be achieved by modifying the radix tree structure used to manage the KV cache. To prevent the leakages of cache-aware task scheduling, the scheduling policy in SGLang needs to be modified to prioritize requests that have at least K shared prefix tokens. Requests with fewer than K shared prefix tokens would still be placed in the waiting list. This approach reduces the likelihood of KV cache sharing, but it is unlikely to significantly impact performance.

Evaluation. To evaluate the effectiveness of the mitigation and reduce the cost of querying the LLM, we conducted a simulation-based experiment. First, we built the classifiers that detect the hits and misses of K tokens for each value of K ($K = 1, 2, 3, 4$), following the method outlined in Section 4.1. Since the timing differences become more pronounced as K increases, we reduced the number of samples (i.e., n) in multi-

Table 6: Token recovery results under different numbers of minimum shared tokens.

K	Recovery rate	Accuracy	#queries per recovered token	#queries per token
1	91.5%	97.9%	118.58	215.41
2	81.0%	98.8%	108.89	286.79
3	67.5%	98.5%	88.30	337.28
4	49.0%	98.0%	62.55	470.82

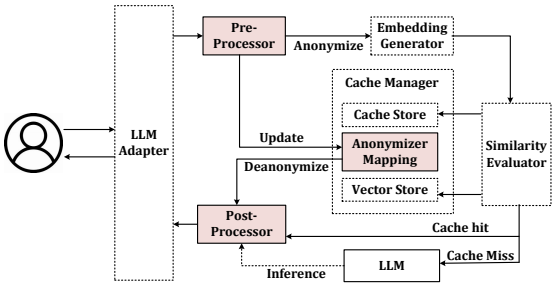


Figure 11: Mitigating semantic cache leakages. The shaded components are customized as part of our mitigation.

sampling, and obtained the corresponding TPRs and FPRs for each classifier. Then we used an *oracle* to simulate the classifiers by randomly sampling a number and determining whether it fell within the classification range. In this evaluation, we used the same repetitive trails method, dataset and fine-tuned model as described in Section 4.1. The next-token predictor was modified to predict the next K tokens. Table 6 presents the token recovery rate and the average number of queries needed to recover 1 token for $K = 1, 2, 3$ and 4. The results show that as K increases, the attack still achieves a notable recovery rate. The average number of queries for recovered tokens decreases with a larger K, as the predictor performs well on easy prompts. However, the overall recovery rate declines because the predictor has more difficulty with harder prompts for a larger K. When considering the tokens not recovered after the maximum of 80 allowed guesses, the average number of queries increases.

5.2 Mitigating Semantic Cache Leakages

Design. As investigated in Section 4.2, private attributes have a significant impact on the semantic similarity between requests. As a result, the PNA infers private attributes by probing whether a semantically similar request is cached. To mitigate this leakage, we propose a strategy that involves identifying and anonymizing the private attributes present in the requests. This approach not only prevents the leakage of private attributes but also increases the potential for sharing requests across users.

As shown in Figure 11, we integrate a custom pre-processor

and post-processor into the GPTCache framework. The pre-processor is designed to identify private attributes within the requests, and replace them with anonymized identifiers. In this approach, we selectively de-identify Personally Identifiable Information (PII) attributes, such as names, email addresses, phone numbers, credit card numbers, and IP addresses, while ensuring that no essential information needed for the LLMs is removed.

To facilitate reuse, the cache manager maintains a mapping structure that stores the anonymized identifier alongside its corresponding private attribute in a key-value format. The post-processor then identifies the anonymized identifiers in the response and replaces them with the private attributes by referencing this mapping. This ensures that the user receives an accurate response.

Evaluation. In our prototype implementation, we used the Presidio tool by Microsoft [3] to automatically identify private attributes. For performance evaluation, we used the evaluation dataset released by Presidio, which includes sentences containing private information. Specifically, we randomly sampled 1,000 sentences from the dataset and fed them into both the original GPTCache and the enhanced GPTCache. Then we measured the average delay introduced by the pre-processor and post-processor. The results show that the anonymization process adds an average delay of approximately 6 ms, while GPTCache's response latency for a semantic cache hit without anonymization is around 0.14 s. Thus, de-identification introduces only about 4% additional overhead, which has a minimal impact on GPTCache's overall performance.

6 Discussions

Unexplored timing leakages. This paper utilizes SGLang [30] and GPTCache [16] as the most representative KV cache and semantic cache sharing mechanisms. However, it is important to note that more sophisticated optimization techniques may enhance performance, but they could also amplify the significance of timing leakage. For example, modular prefix caching [44] and CacheBlend [84] facilitate KV caching for not only prefix tokens but also intermediate ones. Cascade inference [86] stores the shared KV cache in GPU shared memory (SMEM for short), for fast access in multiple requests. The sharing of KV cache in speculative decoding [54] may also introduce speculative side channels, akin to Spectre [51]. We leave the further exploration of the impact of the discovered side channels to future work.

Co-locations. Co-location is a prerequisite for timing side-channel attacks. There has been significant research on how to achieve co-location in the cloud for micro-architectural side channels [68, 74, 90, 96]. Co-location can be more efficient in LLM systems because these systems often focus on optimizing cache reuse through improved scheduling policies. For example, baseline scheduling policies, such as first-come-first-serve, typically do not consider the shared prompts within an LLM system. As a result, requests may be mixed across different LLM engines, preventing the reuse of common prompt prefixes and potentially leading to suboptimal cache efficiency. To address this, LLM serving systems like SGLang [30], Parrot [55], Mooncake [64] and BatchLLM [95] have introduced modified schedulers that prioritize requests that align with cached prefixes, a strategy known as cache-aware scheduling.

7 Related Works

Prompt extraction attacks with adversarial prompts. Most existing research focuses on stealing system prompts from LLMs by eliciting the previous prompts, typically through direct output or translation. For example, a twitter user claimed to have discovered the prompt used by Bing Chat [10]. Earlier studies involved manually constructing attacker prompts [63, 89], while more recent work, such as PLeak, leverages output feedback from a given prompt and introduces an incremental search algorithm to optimize prompt retrieval [50]. Zhang et al. present a framework for systematically evaluating the effectiveness of these attacks. While these approaches exploit the model's vulnerability to adversarial prompts, our proposed attacks take advantage of timing differences introduced in LLM service deployment. As such, our attacks do not rely on the specific details of any particular LLM.

Side channel attacks on LLM. Debenedetti et al. propose system-level side channels within the deep learning lifecycle, such as training data filtering, input preprocessing, output monitoring, and query filtering. These side channels can potentially be exploited to infer the training data or the requests [40]. LLM keystroking attacks [79] are a type of packet analysis based side channels. These attacks exploit the length of response tokens, assuming the attacker has access to encrypted network packets. By comparison, we are the *first* to study the timing leaks introduced by LLM optimizations, rather than relying on output data or token packet sizes to recover requests. Inspired by our work, a follow-up study by Gu et al. reports a comprehensive measurement analysis of prompt caching in real-world LLMs [48], detecting prompt caching in 8 out of 17 LLM service providers. More recently, Zheng et al. conducted a similar study on LLM timing side channels [94], which however does not feature our optimized search strategy for efficient request recovery.

Micro-architectural side channel attacks on deep learning systems. Numerous studies have explored methods for extracting deep learning models or structures, as well as fingerprinting these models, by exploiting various side channels, such as CPU [41, 46, 65, 71, 83], GPU [77], FPGA [88], power and magnetic channels [49, 57], and PCIe traffic [99]. In comparison, our work focuses on leaking private prompts rather

than stealing model parameters. Additionally, our approach does not rely on the micro-architectural or power characteristics of specific hardware; instead, it exploits timing leaks inherent in LLM systems. As a result, our attacks are applicable across CPU, GPU, and FPGA platforms, provided they utilize KV cache or semantic cache sharing techniques.

8 Conclusions

LLM inference is a resource-intensive process, prompting numerous studies focused on reducing inference costs and latency. These optimizations often involve the use of various caches. When multiple users share the LLM system, these optimizations can lead to interference between users. This paper examines the side channels created by such interference, identifying two types of leaks: one in the KV cache and another in the semantic cache. We urge LLM system providers to recognize this emerging threat and prioritize security in their design choices.

References

[1] Samsung bans staff's ai use after spotting chatgpt data leak. https://www.bloomberg.com/news/articles/2023-05-02/samsung-bans-chatgpt-and-other-generative-ai-use-by-staff-after-leak.

[2] Medquad. https://huggingface.co/datasets/lavita/MedQuAD, 2019.

[3] Presidio. https://github.com/microsoft/presidio, 2022.

[4] Ai document summarization. https://www.ibm.com/architectures/hybrid/genai-document-summarization, 2024.

[5] Alibaba platform for ai. https://www.alibabacloud.com/help/en/pai/, 2024.

[6] Amazon bedrock - build generative ai applications with foundation models. https://aws.amazon.com/bedrock/, 2024.

[7] Anythingllm: The all-in-one desktop & docker ai application with built-in rag, ai agents, and more. https://github.com/Mintplex-Labs/anything-llm, 2024.

[8] Bingchat. https://www.bing.com/chat, 2024.

[9] Chatgpt | openai. https://openai.com/chatgpt/, 2024.

[10] The entire prompt of microsoft bing chat. https://twitter.com/kliu128/status/1623472922374574080, 2024.

[11] Estimate llm inference speed and vram usage quickly: with a llama-7b case study. https://www.jinghong-chen.net/estimate-vram-usage-in-llm-inference/, 2024.

[12] Gemini. https://deepmind.google/technologies/gemini/, 2024.

[13] Get cached responses of azure openai api requests. https://learn.microsoft.com/en-us/azure/api-management/azure-openai-semantic-cache-lookup-policy, 2024.

[14] Github copilot · your ai pair programmer. https://github.com/features/copilot/, 2024.

[15] Gpt-4 document analysis – klu. https://klu.ai/use-cases/document-analysis, 2024.

[16] Gptcache : A library for creating semantic cache for llm queries. https://github.com/zilliztech/gptcache, 2024.

[17] Gptcache usage. https://python.langchain.com/api_reference/community/cache/langchain_community.cache.GPTCache.html, 2024.

[18] Groq. https://groq.com/, 2024.

[19] Jpmorgan rolls out in-house genai-based chatbot to employees. https://www.financedirectoreurope.com/news/jpmorgan-rolls-out-ai-based-chatbot/, 2024.

[20] Langchain. https://www.langchain.com/, 2024.

[21] Langchain applications. https://lablab.ai/apps/tech/langchain/langchain, 2024.

[22] A large language model for healthcare. https://aiforhealthcare.substack.com/p/a-large-language-model-for-healthcare, 2024.

[23] Large language model text generation inference. https://github.com/huggingface/text-generation-inference, 2024.

[24] Llamaindex is a data framework for your llm applications. https://github.com/run-llama/llama_index, 2024.

[25] Lumos. https://github.com/andrewnguonly/Lumos, 2024.

[26] Perplexity ai. https://www.perplexity.ai/, 2024.

[27] Prompt caching in the api. https://openai.com/index/api-prompt-caching/, 2024.

[28] Prompt caching with claude. https://www.anthropic.com/news/prompt-caching, 2024.

[29] Prompt template. https://python.langchain.com.cn/docs/modules/model_io/prompts/prompt_templates/, 2024.

[30] Sglang is yet another fast serving framework for large language models and vision language models. https://github.com/sgl-project/sglang, 2024.

[31] Summarizing documents with llms: A comprehensive guide. https://www.linkedin.com/pulse/summarizing-documents-llms-comprehensive-guide-sharat-kedari-4vdfc, 2024.

[32] System prompt leakage dataset. https://huggingface.co/datasets/gabrielchua/system-prompt-leakage, 2024.

[33] System prompts in large language models. https://promptengineering.org/system-prompts-in-large-language-models/, 2024.

[34] Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2024.

[35] Vertex ai with gemini 1.5 pro and gemini 1.5 flash. https://cloud.google.com/vertex-ai, 2024.

[36] vllm: A high-throughput and memory-efficient inference and serving engine for llms. https://github.com/vllm-project/vllm, 2024.

[37] llm side-channel demo. https://sites.google.com/view/early-bird-catches-the-leak, 2025.

[38] 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)*, pages 212–218, 2023.

[39] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[40] Edoardo Debenedetti, Giorgio Severi, Nicholas Carlini, Christopher A Choquette-Choo, Matthew Jagielski, Milad Nasr, Eric Wallace, and Florian Tramèr. Privacy side channels in machine learning systems. In *33rd USENIX Security Symposium*, 2024.

[41] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E Balas. Stealing neural networks via timing side channels. *arXiv preprint arXiv:1812.11720*, 2018.

[42] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[43] 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.

[44] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

[45] Louie Giray. Prompt engineering with chatgpt: a guide for academic writers. *Annals of biomedical engineering*, 51(12):2629–2633, 2023.

[46] Cheng Gongye, Yunsi Fei, and Thomas Wahl. Reverse-engineering deep neural networks using floating-point timing side-channels. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[47] Sglang Group. flush cache. https://github.com/sgl-project/sglang/blob/25e5d589e39b3b605296395e4f9c96ec42f09055/python/sglang/srt/server.py#L164, 2024.

[48] Chenchen Gu, Xiang Lisa Li, Rohith Kuditipudi, Percy Liang, and Tatsunori Hashimoto. Stanford cs 191w senior project: Timing attacks on prompt caching in language model apis. 2024.

[49] Peter Horvath, Lukasz Chmielewski, Leo Weissbart, Lejla Batina, and Yuval Yarom. Barracuda: Bringing electromagnetic side channel into play to steal the weights of neural networks from nvidia gpus. *arXiv preprint arXiv:2312.07783*, 2023.

[50] Bo Hui, Haolin Yuan, Neil Gong, Philippe Burlina, and Yinzhi Cao. Pleak: Prompt leaking attacks against large language model applications. *arXiv preprint arXiv:2405.06823*, 2024.

[51] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[52] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Netcat: Practical cache attacks from the network. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2020.

[53] 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 (SOSP)*, pages 611–626, 2023.

[54] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[55] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable. *arXiv preprint arXiv:2405.19888*, 2024.

[56] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*, 2023.

[57] Henrique Teles Maia, Chang Xiao, Dingzeyu Li, Eitan Grinspun, and Changxi Zheng. Can one hear the shape of a neural network?: Snooping the gpu via magnetic side channel. In *USENIX Security Symposium*, pages 4383–4400, 2022.

[58] Kai Mei, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. Aios: Llm agent operating system. *arXiv e-prints*, pages arXiv–2403, 2024.

[59] Meta. Llama-3.1-8b-instruct. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct, 2024.

[60] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

[61] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2139–2153, 2018.

[62] Seungcheol Park, Jaehyeon Choi, Sojin Lee, and U Kang. A comprehensive survey of compression algorithms for language models. *arXiv preprint arXiv:2401.15347*, 2024.

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

[64] 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.

[65] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In *2022 IEEE symposium on security and privacy (SP)*, pages 1157–1174. IEEE, 2022.

[66] Philippe Remy. Name dataset. https://github.com/philipperemy/name-dataset, 2021.

[67] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended abstracts of the 2021 CHI conference on human factors in computing systems*, pages 1–7, 2021.

[68] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[69] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pages 279–299. Springer, 2019.

[70] Uri Shaham, Maor Ivgi, Avia Efrat, Jonathan Berant, and Omer Levy. Zeroscrolls: A zero-shot benchmark for long text understanding. *arXiv preprint arXiv:2305.14196*, 2023.

[71] Shubhi Shukla, Manaar Alam, Pabitra Mitra, and Debdeep Mukhopadhyay. Stealing the invisible: Unveiling pre-trained cnn models through adversarial examples and timing side-channels. *arXiv preprint arXiv:2402.11953*, 2024.

[72] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. *arXiv preprint arXiv:2312.12456*, 2023.

[73] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. Hot pixels: Frequency, power, and temperature attacks on {GPUs} and arm {SoCs}. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6275–6292, 2023.

[74] 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)*, pages 913–928, 2015.

[75] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[76] Wenxiao Wang, Wei Chen, Yicong Luo, Yongliu Long, Zhengkai Lin, Liye Zhang, Binbin Lin, Deng Cai, and Xiaofei He. Model compression and efficient inference for large language models: A survey. *arXiv preprint arXiv:2402.09748*, 2024.

[77] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 125–137. IEEE, 2020.

[78] Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. Outlier suppression: Pushing the limit of low-bit transformer language models. *Advances in Neural Information Processing Systems*, 35:17402–17414, 2022.

[79] Roy Weiss, Daniel Ayzenshteyn, Guy Amit, and Yisroel Mirsky. What was your prompt? a remote keylogging attack on ai assistants. *arXiv preprint arXiv:2403.09751*, 2024.

[80] Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David S. Rosenberg, and Gideon Mann. Bloomberggpt: A large language model for finance. *CoRR*, abs/2303.17564, 2023.

[81] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

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

[83] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020, 2020.

[84] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving with cached knowledge fusion. *arXiv preprint arXiv:2405.16444*, 2024.

[85] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 35:27168–27183, 2022.

[86] Zihao Ye, Ruihang Lai, Bo-Ru Lu, Chien-Yu Lin, Size Zheng, Lequn Chen, Tianqi Chen, and Luis Ceze. Cascade inference: Memory bandwidth efficient shared prefix batch decoding, February 2024.

[87] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[88] Yicheng Zhang, Rozhin Yasaei, Hao Chen, Zhou Li, and Mohammad Abdullah Al Faruque. Stealing neural network structure through remote fpga side-channel analysis. *IEEE Transactions on Information Forensics and Security*, 16:4377–4388, 2021.

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

[90] 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*, pages 990–1003, 2014.

[91] Youpeng Zhao, Di Wu, and Jun Wang. Alisa: Accelerating large language model inference via sparsity-aware kv caching. *arXiv preprint arXiv:2403.17312*, 2024.

[92] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[93] 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.

[94] Xinyao Zheng, Husheng Han, Shangyi Shi, Qiyan Fang, Zidong Du, Qi Guo, and Xing Hu. Inputsnatch: Stealing input in llm services via timing side-channel attacks. *arXiv preprint arXiv:2411.18191*, 2024.

[95] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. Batchllm: Optimizing large batched llm inference with global prefix sharing and throughput-oriented token batching. *arXiv preprint arXiv:2412.03594*, 2024.

[96] Wu Zhenyu, Xu Zhang, and H Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[97] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. Vulnerable gpu memory management: towards recovering raw data from gpu. *arXiv preprint arXiv:1605.06610*, 2016.

[98] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633*, 2023.

[99] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal DNN models with lossless inference accuracy. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.