ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

I Know What You Said: Unveiling Hardware Cache Side-Channels in Local Large Language Model Inference

Zibo Gao[1,2], Junjie Hu[1,2], Feng Guo[1,2], Yixin Zhang[1,2,✉], Yinglong Han[1,2], Siyuan Liu[1,2], Haiyang Li[1,2], and Zhiqiang Lv[1,2,✉]

[1]Institute of Information Engineering, Chinese Academy of Sciences.

[2]School of Cyber Security, University of Chinese Academy of Sciences.

Email: {gaozibo,hujunjie,guofeng,zhangyixin,hanyinglong,liusiyuan,lihaiyang,lvzhiqiang}@iie.ac.cn

arXiv:2505.06738v3 [cs.CR] 15 Jun 2025

Abstract

Large Language Models (LLMs) that can be deployed locally have recently gained popularity for privacy-sensitive tasks, with companies such as Meta, Google, and Intel playing significant roles in their development. However, the security of local LLMs through the lens of hardware cache side-channels remains unexplored. In this paper, we unveil novel side-channel vulnerabilities in local LLM inference: token value and token position leakage, which can expose both the victim's input and output text, thereby compromising user privacy. Specifically, we found that adversaries can infer the token values from the cache access patterns of the token embedding operation, and deduce the token positions from the timing of autoregressive decoding phases. To demonstrate the potential of these leaks, we design a novel eavesdropping attack framework targeting both open-source and proprietary LLM inference systems. The attack framework does not directly interact with the victim's LLM and can be executed without privilege.

We evaluate the attack on a range of practical local LLM deployments (e.g., Llama, Falcon, and Gemma), and the results show that our attack achieves promising accuracy. The restored output and input text have an average edit distance of 5.2% and 17.3% to the ground truth, respectively. Furthermore, the reconstructed texts achieve average cosine similarity scores of 98.7% (input) and 98.0% (output).

1 Introduction

Large Language Models (LLMs), such as OpenAI's ChatGPT [14], Meta's Llama [8], Google's Gemma [4], and Mistral AI's Mistral [13], have enabled a broad spectrum of applications ranging from chatbots to personal agents. Their ability to follow human instructions and make decisions has garnered substantial attention from the public, reshaping the digital landscape.

✉ Corresponding authors.

However, users may unintentionally disclose private data while interacting with LLMs. For instance, Samsung Electronics exposed its confidential data to a cloud-based LLM service in 2023, resulting in a ban on employee use of generative AI tools [23]. To mitigate the risk of sharing sensitive data with third parties, locally deployed LLMs have garnered increasing attention for their suitability in privacy-critical tasks [57, 72, 79, 81]. This paradigm is supported by corporations such as Meta, Google, and Microsoft. Local LLMs are suited for handling sensitive tasks such as editing confidential emails, seeking advice on personal matters, and assisting with financial analysis [57, 80].

Unfortunately, we find a new security threat in which LLMs' sensitive input and output text during inference can be leaked through hardware cache side channels, which has not been previously reported, to the best of our knowledge. Specifically, we shed light on several fundamental characteristics of LLM inference that cause the side-channel leakage: (i) LLMs depend on *token embedding* [34, 42, 52, 68, 76], which is essential for converting text into semantic representations that the model can process. However, this embedding operation creates secret-dependent data access, exposing the input's token (akin to words) values through cache access patterns. (ii) LLM inference generally follows the *autoregressive* paradigm, where each output token is recursively fed into the model and passed through the token embedding, enabling the side-channel leakage for both the input and output tokens. Moreover, the autoregressive generation is inherently sequential and unfolds over multiple time steps, implying that the timing of embedding operations correlates with the position of the output token. Exploiting these characteristics, adversaries can launch a spy application co-located with the LLM to probe cache access timing and form cache traces, then map the cache traces to the victim's input and output text.

Challenges. We need to overcome several unique challenges to realize the attack in practical scenarios. First, the cache side-channel exhibits a relatively low signal-to-noise ratio (SNR) due to system activities during LLM inference. This noise can result in missing or randomly valued words in the re-

sulting text.Second, in practical systems, the time order of the token embedding operation for the model input is interleaved or overlapped in the time axis of the cache trace, resulting in a shuffled order of the mapped input tokens. The root cause lies in the batching of input tokens. Their batched processing through parallel computing results in closely clustered execution times and an unpredictable sequence of operations.

Solution. To address the first challenge, we propose a novel text reconstruction algorithm that fuses both the *timing signal* and *token list* mapped from the cache trace. Specifically, we utilize Power Spectral Density (PSD) in signal processing to analyze the trace's timing signal (time series of the cache hit events). We find that the timing signal during the LLM decode phases exhibits strong periodicity, whereas false positive noise shows randomness with an evenly distributed spectrum. Leveraging these findings, we synthesize a dataset to train LLMs to capture the timing patterns and reconstruct the clean text. During dataset synthesis, we sample tokens from publicly available textual corpora and assign each token a periodic timestamp to simulate the timing signal. Then, we remove a random subset of the tokens to simulate false negative noise, and insert random tokens at uniformly distributed time points to simulate false positive noise. We then fine-tune LLMs on the synthesized dataset to reduce noise and reconstruct the victim mode's output text from the nosy cache trace.

To address the second challenge, we exploit the contextual dependence between model input and output text. Specifically, we fuse the token list mapped from the cache trace and the reconstructed output text as a whole context, then fine-tune LLMs on a randomly synthesized dataset to restore the original response from the context.

After overcoming the above challenges, we present the first LLM-targeted hardware cache side-channel eavesdropping framework. The attack does not directly interact with the victim model. Instead, it only passively (i.e., being stealthy) observes the behavior of the hardware cache shared with the LLM inference process and does not rely on profiling the victim.

Evaluations. We demonstrate the feasibility of the eavesdropping attack on a range of popular LLMs (e.g., Meta Llama, Google Gemma, and Microsoft Phi) deployed on a variety of popular LLM frameworks, such as HuggingFace transformers, Intel IPEX-LLM, open-source llama.cpp. Our empirical experiments show that the attack can effectively restore the full text of the model output and reconstruct the model input with approximate semantics. When attacking llama.cpp, our method achieved an average Levenshtein similarity (1 - Edit Distance) of 94.8% and 82.7% for the restored output and input text, respectively. Furthermore, our method reached an average cosine similarity score of 98.7% (output) and 98.0% (input), demonstrating significant information leakage.

Contributions. In a nutshell, the contributions of this paper include:

- *Novel Side-Channel Attack.* We introduce a new side-channel attack against LLM inference, which leaks both the model input and output text via hardware cache side-channels.
- *New End-to-end Attack Framework.* We present the first hardware cache side-channel attack framework targeted both model input and output of various currently deployed LLMs, without profiling or directly interacting with the victim model.
- *Efficient Dataset Synthesis Strategy.* We innovate in automatically synthesizing datasets and train LLMs to capture the timing patterns of the cache trace and reconstruct the model output text from the noisy cache trace.
- *New Input Reconstruction Strategy.* We fine-tune LLMs to reconstruct the original order of shuffled input tokens via the contextual dependence between model input and output text.
- *Empirical Study.* We conduct experiments in real-world scenarios to evaluate the threat of the proposed attack strategies. Additionally, we discuss mitigation of the attack and security recommendations for LLM inference framework implementations.

2 Background

2.1 Large Language Model Inference

An LLM is typically a generative model that estimates the probability distribution of the next token (akin to words) conditioned on the given model input and context. Based on the token distribution, LLMs sample new tokens to generate model output.

Token and Tokenization. A token is the smallest meaningful unit of natural language, which can be a word or a part of the word. Tokens are generated by a process named tokenization, which segments a text into a sequence of individual units denoted by $[t_1, t_2, ..., t_N]$, where for each $1 \leq i \leq N, t_i \in V$, and V is the vocabulary. For example, a text tokenized by a Byte Pair Encoding (BPE) tokenizer is shown as follows:

$$['\text{The}', '\text{quick}', '\text{fox}', '\text{jumps}', '\text{over}', '\text{the}', '\text{lazzy}', '\text{log}']$$

Similarly, the above token sequence can be mapped back to the original text through de-tokenizers, which implement the inverse function of the tokenizers.

We note that tokenizers and de-tokenizers used by different LLMs generally follow similar principles of segmenting the text into meaningful units. Moreover, most LLM vendors do not consider the tokenizers to be secrets. Even proprietary vendors like OpenAI have made their tokenizers public [58].

Model Input. The input to the LLM is a token sequence, denoted as $I = [t_1, t_2, ..., t_N]$, where N is the input length. The

input commonly contains system and user prompts, which can be human instructions or other requests that the user wants the LLM to respond to.

Model Output. In response to the input, the LLM generates an output text, also denoted as $O = [t_{N+1}, t_{N+2}, ...t_{N+M}]$ where M is the output length.

Model Inference. LLM inference refers to the process where the model accepts the given input and context and generates responses, wherein the context consists of previously generated text. Generally, LLM inference follows the autoregressive paradigm that can be divided into two phases.

In *prefill phase*, LLM accepts all the input tokens in this phase. It estimates the distribution $P(t_{N+1}|t_1, t_2, ..., t_N)$ through neural networks. This distribution decides the first new token $t_{N+1} \sim P(t_{N+1}|t_1, t_2, ..., t_N)$.

In *decode phases*, LLMs output the subsequent tokens *autoregressively* one at a time until encountering an end-of-sentence (EOS) token. The i-th token in the output is obtained by decoding the distribution $t_{N+i} \sim P(t_{N+i}|t_1, t_2, ..., t_{N+i-1})$. Then, the sampled token t_{N+i} is fed into the model again to compute $P(t_{N+i+1}|t_1, t_2, ..., t_{N+i})$ and produce the next token t_{N+i+1}. Tokens must be into the numerical representation that the neural networks can process; therefore, LLMs fundamentally require the *token embedding* operation to retrieve the embedding vectors for each token.

Local Inference. Traditionally, LLMs were deployed in expensive high-end hardware in data centers [66]. With the evolution of LLM quantization [33], pruning [55], and optimizations of operators [31, 50], local LLM inference has become feasible on more accessible hardware such as consumer-grade PCs. For example, Intel has released a white paper on CPU-based LLM inferences [71]. Furthermore, there is a growing trend toward AI PCs [17] that deploy LLMs in low-cost personal devices.

Unlike cloud LLMs, local LLMs avoid sending private data to third parties on the Internet, therefore reducing the attack surfaces associated with remote services. Hence, local LLMs are preferred by privacy-critical tasks [57, 72, 79, 81].

2.2 Cache Side-Channel Attacks

Modern processors extensively use caches to mitigate the high latency and inadequate bandwidth of off-chip memories (e.g., DRAMs). When the processor loads or stores memory locations, it first attempts to access the data from the caches. If the desired data are present in the caches (on local or remote cores), it can be directly accessed on the chip with minimal latency. We refer to this case as *cache hit*. Conversely, if *cache miss* occurs, the processors will fetch the data from the off-chip memories, which incurs latency of an order of magnitude higher than that of a cache hit. Moreover, cache coherence protocols allow the sharing of remote cache contents across process cores and sockets within a machine. Once attackers share memory and machine with the victim, they can probe the memory's load latency to infer whether the victim program accesses the data unit, resulting in cross-core and cross-socket side-channel attacks such as flush+reload [78], flush+flush [39], and invalidate+transfer [48]. The cross-core attacks apply to both same-core (with shared local caches) and different-core (with coherent caches) co-location scenarios.

Secret-Dependent Data Access. Adversaries may infer confidential data of a program if secret values influence its memory access. For example, the Rijndael Advanced Encryption Standard (AES) proposal employs several pre-computed lookup tables, namely T-box, to optimize performance [30]. However, without proper protections, the table-lookup implementation exposes secret-dependent data access. By inferring which data unit the victim program visits, adversaries can ultimately recover the secret key [44, 59].

3 Attack Overview

In this section, we provide an overview of the eavesdropping attack. We first illustrate the threat model. Then, we present a high-level description of the vulnerability and the workflow of attack phases.

3.1 Threat Model

As shown in Figure 1, we investigate the feasibility of the eavesdropping attack in real-world scenarios where a victim is using a locally deployed LLM. In this scenario, the operating system isolates the adversary and victim processes. We assume that the unprivileged adversary cannot compromise the victim's LLM inference system.

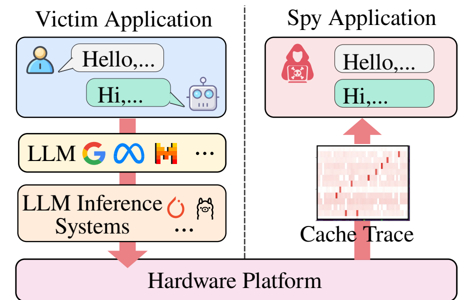


Figure 1: The threat model of our eavesdropping attack. Up and down arrows denote information leakage paths. Dashed line in the middle represents OS process isolation.

Adversary. We assume that a spy application made by the adversary can be installed and executed on the victim machine. Adversaries can create a benign application within popular categories and implant malicious code. Since the spy application does not tamper with the software libraries used by LLM inference, and only involves legitimate system operations, it can be safely published to the public (e.g., Microsoft Store). Once launched, the spy application will stealthily eavesdrop

on the LLM in the background but will not interact with the victim LLM. The attack does not require any special privileges, and the spy's virtual address space is isolated from the victim process by the OS. Consequently, the victim will be unaware of the attack.

Adversary's Capability. The adversary cannot trigger requests to the victim LLM and also cannot tamper with the LLM software. However, aligned with the standard flush+reload attack model, we make the following assumptions: (i) The adversary can execute malicious code on the victim machine. (ii) The malicious code can open the model file in read-only mode and call `mmap`. (iii) If `mmap` is unavailable, the adversary can leverage page deduplication to access shared memory. (iv) The adversary can execute the cache line flushing instruction `clflush`.

Additionally, the adversary does *not* need to know the model architecture and source code or binary of the LLM inference framework, but he/she has access to the publicly available information to compute the address offset of embedding table elements in the model files. The rules for deriving the offset are not secret for mainstream LLMs [2, 15, 16].

Victim. The victim interacts with a locally deployed LLM for various purposes, such as seeking personal advice and writing sensitive emails. We assume that the token embedding operation in the LLM inference is offloaded to CPU while other layers of the models can be sent to any device. This assumption is realistic because it is the default behavior of the mainstream locally-deployed LLM inference frameworks, mainly due to the optimum cost-effectiveness of CPU operations in local deployment scenarios supported by Intel [71], Meta [9], Numenta [67], and the community [18, 37].

3.2 The Vulnerability

By studying how LLM inference generally works, we reveal novel side-channel leakage sources that unveil token *value* and *position*, enabling reconstruction of the original text.

Token Value Leakage. LLMs depend on token embedding to convert token sequences into semantic representations that the model can process. The token embedding is essentially a linear projection that maps one-hot-encoded token lists into its dense representation. Suppose that the token list is encoded as the one-hot matrix $\mathbf{x}$, where $\mathbf{x}_i = [0 \cdots 1_{(t_i)} \cdots 0]^T$ and t_i is the i-th token index. Then, the token embedding is formulated as $\mathbf{E} = \mathbf{W}\mathbf{x}$.

Due to the sparsity of one-hot encoding, it is common to simplify the matrix multiplication into the table lookup $E_i = \mathbf{W}[t_i]$, i.e., retrieving the t_i-th row of $\mathbf{W}$. Since only the requested row vectors are loaded and cached on the chip, adversaries can infer the token index t_i by monitoring which row of embedding table $\mathbf{W}$ has been recently accessed by the victim.

Moreover, due to the autoregressive characteristic, all the newly produced tokens are *fed back* into the model, where each new token is sent to the token embedding operation. Therefore, the token embedding operation leaks both the victim's input and output tokens.

Without any privilege, the spy process cannot directly monitor the memory access of the victim process. Fortunately, cache side-channel attacks can be exploited to indirectly infer the victim's memory access. Once a cache hit of the j-th row of $\mathbf{W}$ is observed, it can be inferred that the embedding operation has processed the token with the index of j. Moreover, the token embedding operation is typically offloaded into the CPU, due to the optimum cost-effectiveness of CPU computing. In contrast, other layers of the model can be offloaded to GPUs. Hence, the adversary can mount cache side-channel attacks on CPU only to cover scenarios where the victim uses GPU to accelerate the model inference.

Token Order Leakage. The above leakage unveils values of input and output tokens, but their *positions* in the original sequence remains unknown. We seek another leakage source in the temporal domain to recover the token order. Specifically, LLM inference generally follows the autoregressive paradigm, which consists of a prefill phase and a series of decode phases. These phases are inherently serialized. The decode phases will not begin until the prefill is complete. Similarly, the $i+1$-th decode phase will wait for the completion of the i-th token. Therefore, the time points of these distinct phases are distinguishable in the time dimension, leaking: (1) the boundary between input and output tokens, and (2) the order of output tokens.

However, it is non-trivial to implement the attack by exploiting the above leakages, given the following unique challenges:

Challenge 1 (C1). *Noise in the side channels introduces errors.* We deduce the token values by monitoring the cache access of each row of $\mathbf{W}$. However, the cache side-channel presents noise, i.e., false positives and false negatives, which leads to randomly-valued and missing tokens.

Challenge 2 (C2). *From the side-channel observer perspective, the order of input tokens is scrambled.* During model inference, input tokens of the LLM are batched and computed in parallel. Therefore, the token embedding operation for each input token is randomly interleaved or overlapping in the time axis, prohibiting adversaries from restoring the token positions via the timing of token embedding operations.

3.3 Attack Workflow

In this work, we address **C1** by fine-tuning LLM_A on a synthetic dataset to capture the cross-modality features, namely timing signal and token text, with the aim of reconstructing the original text from the noisy cache trace. We tackle **C2** by fine-tuning LLM_B on the synthetic dataset to exploit the contextual dependence between model output and input, and restore the position of input tokens. The workflow of our attack consists of the following phases, shown in Figure 2.

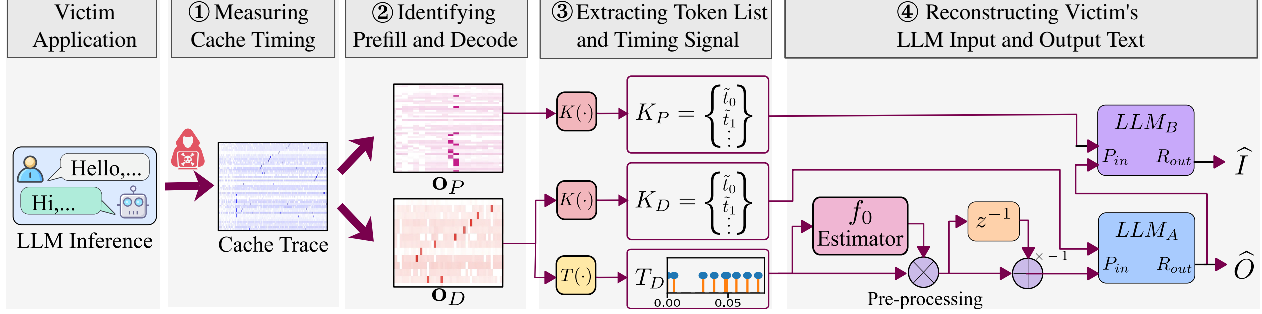


Figure 2: Workflow of our eavesdropping attack.

1. Execute the spy process that co-locates with the victim and collects the cache trace $\mathbf{o}$ during LLM inference.

2. Identify cache trace segments $\mathbf{o}_P$ and $\mathbf{o}_D$ that correlate with the prefill and decode phases of the victim.

3. Map the cache trace to the ordered token list K_D and the timing signal T_D for the decode phase of the victim. Additionally, derive the unordered token list K_P for the prefill phases.

4. Reconstruct the victim's model output by fusing K_D and T_D and infer the text via $\widehat{O} = LLM_A(K_D, T_D)$.

5. Reconstruct the victim's model input by fusing $\widehat{O}$ and K_P, and deduce the text by $\widehat{I} = LLM_B(\widehat{O}, K_P)$.

4 Attack Implementation

In this section, we elaborate on the implementation of the attack phases presented in the workflow.

4.1 Measuring Cache Timing

To exploit the leakage source described in §3.2, the adversary should infer the victim's cache accesses to $|V|$ rows in the embedding table $\mathbf{W}$. These accesses can be deduced from the cache trace that is captured by shared-memory-based cache side-channel attacks. We use this kind of attack because it offers spatial granularity of cache line. This fine spatial granularity enables the differentiation of distinct rows in $\mathbf{W}$. In this work, we use flush+reload [78], but we expect that the attack can also be implemented via other shared-memory cache attacks, such as flush+flush [39] and evict+reload [40].

Before mounting the attack, the adversary should compute a total of $|V|$ target addresses using publicly available information about the model file format [2, 15, 16]. Suppose that the start address of the shared memory region is p_1, and the address offset of $\mathbf{W}$ is p_2. For the i-th row of $\mathbf{W}$ $(0 < i < |V|)$, if the $\mathbf{W}$ is in row-major order, the target address A_i can be selected in the range $p_1 + p_2 + iDb \leq A_i < p_1 + p_2 + (i+1)Db$, where D is the dimension of the embedding vector, and b is the size of the vector element. Similarly, if the $\mathbf{W}$ is in column major order, we select the address $A_i \in \{p | p = p_1 + p_2 + (i + j|V|)b, 0 \leq j < D\}$. For each i, only one deterministic address within the ranges or sets is chosen. However, we retain the ranges and sets and will resolve A_i in the next phase.

Evading Hardware Prefetchers. Interestingly, we reveal that the standard flush+reload implementation failed to maintain the required precision on Intel's recent microarchitecture, namely Raptor Lake. After implementing state-of-the-art adversarial strategies, the standard flush+reload implementation [77] achieves near-zero precision in probing $|V| = 32768$ addresses (the vocabulary size of Llama2), as the average false positive count reaches 5,612,394 per second, indicating high noise from prefetcher activities.

The root cause of this failure is that Intel has employed Array-of-Pointers (AoP) prefetchers in hardware [27], which unfortunately degrades the flush+reload attacks. Specifically, the attack program depends on a pointer array to store the shuffled target addresses, which is required to overcome streaming and spatial prefetchers [77]. When the program visits the array, the AoP prefetcher automatically dereferences the memory pointers in the pointer array, leading to a cache hit for almost every target address.

To overcome the AoP prefetchers, we calculate the minimum value of all target addresses as the offset, subtract this offset from each address, and store the subtracted results in an array. When each target address is required in the attack, the original address is restored by adding the offset to the corresponding value in the array. This ensures that the array no longer contains valid pointers, thereby preventing prefetch triggering by the AoP.

For other types of prefetcher, we follow previous works, i.e., we choose the target addresses A_i such that no adjacent pages are probed [40], and randomize the order of probing sequence [64].

Allocate Shared Memory. To achieve the shared memory relied on the cache attacks, we employ two orthogonal strategies: (i) We note that the memory content of $\mathbf{W}$ is loaded from

model files on the disk. Currently deployed LLM frameworks typically adopt the zero-copy loading technique to optimize the loading performance, eliminating extra data moving via virtual memory mapping and demand paging. Inspired by this observation, the spy process can share memory with the victim by simply calling `mmap` on the model file. Once the victim process maps to the same file, the adversary will automatically share the physical frames of the file with the victim due to the page cache. In this case, page duplication is unnecessary. (ii) In rare cases where the victim does not support zero-copy model loading, we utilize page deduplication [21,62] provided by OS to obtain the shared memory.

A detailed pseudocode for the trace acquisition process using standard flush+reload with conventional multithreaded partitioning is provided in the Appendix E.

Example of Cache Trace. The obtained cache trace $\mathbf{o}$ is a $L \times |V|$ matrix, where each row denotes the memory accessing latency of $|V|$ target addresses at L time points, which is shown in Figure 3. Additionally, the physical timestamp of each time point is captured by `rdtsc` instruction and recorded in the vector $\mathbf{t}$ to facilitate subsequent attack phases.

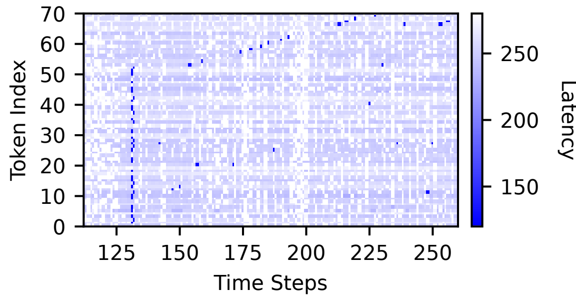


Figure 3: An example of cache trace. A deeper color indicates a higher probability that the token embedding operation accesses the token. For clarity, tokens are sequentially re-indexed.

4.2 Identifying Prefill and Decode Phase

According to our threat model, the spy process does not need to interact with the victim process, i.e., it neither controls the start of the victim process, nor conducts inter-process communication with the victim. Like preceding studies, this setting belongs to asynchronous attack [44]. Therefore, the adversary needs to identify the start time of LLM inference phases, particularly the boundary of prefill and decode phases.

We design a pattern-matching algorithm to find the cache trace segment $\mathbf{o}_P$ and $\mathbf{o}_D$ that correlates with prefill and decode phases, respectively. Our algorithm exploits the behavior of autoregressive generation. Particularly, prefill phase batches input tokens as a whole and compute token embedding in parallel. Therefore, the prefill phase causes abnormally *dense* cache hit events that cluster around a short period (e.g., around time step 130 in Figure 3). Meanwhile, decode phases produce output tokens *serially*, correlating with cache hit events much *sparser* than the prefill phase in the time axis (e.g., one cache hit per time point).

We match $\mathbf{o}_P$ and $\mathbf{o}_D$ in the $\mathbf{o}$ via time interval between each two consecutive cache hit events, i.e., $\mathbf{t}_i - \mathbf{t}_{i-1}, 0 < i < L$. The start point of $\mathbf{o}_P$ is matched if no less than K consecutive events after the point have time intervals that are all less than a threshold α_1. Meanwhile, the start point of $\mathbf{o}_D$ is matched if at least one subsequent event has a time interval greater than α_1. The parameter K is the minimum allowed number of input tokens ($K = 4$ in our implementation), and the threshold α_1 is no greater than the minimum time of one model forward propagation ($\alpha_1 = 10^{-3}$ in our settings).

4.3 Mapping Token Lists and Timing Signal

Having obtained the $\mathbf{o}_P$ and $\mathbf{o}_D$, we now map the cache trace to the token lists and timing signal for further text reconstruction.

As discussed in §3.2, the cache hit of the target address A_i indicates that the token embedding operation has accessed the i-th row of $\mathbf{W}$, and hence implies that the token i has appeared in the model input or output. We define a cache hit event as the time points where the memory access latency is lower than a predefined threshold α_2, following previous cache attacks [78]. The threshold can be obtained by micro-benchmarks for caches [20].

Token Lists. The mapping of token lists consists of two steps. First, we compute $K(\mathbf{o}_D) = [k_1, k_2, \cdots]$, $k_i = \{\mathcal{T}^{-1}(j) | \mathbf{o}_{Dij} < \alpha_2, 0 \leq j < |V|\}$, where α_2 is the cache hit threshold, $\mathcal{T}^{-1}$ is the de-tokenizer that converts the token index into the token text. Second, we remove all the empty sets in the $K(\mathbf{o}_D)$ to derive the resulting token list K_D. Similarly, we can derive the unordered token list $K_P = [\mathcal{T}^{-1}(j) | \mathbf{o}_{Pij} < \alpha_2, 0 \leq i < L, 0 \leq j < |V|]$, where L is the length of the cache trace.

Timing Signal. We extract the timing signal by deriving cache hit events' timestamps corresponding to each token. Formally, $T_D = [\mathbf{t}_i | K_i(\mathbf{o}_D) \notin \emptyset, 0 \leq i < L]$.

4.4 Reconstructing Model Output

As mentioned in **C1** (§3.2), the cache side-channel contains noise, specifically false positive and false negative noise, which induces errors in the token lists and complicates the attack. A *false positive noise* indicates an observed cache hit event that is not attributed to the victim's token embedding operation. According to our token list mapping approach, false positives induce randomly inserted tokens in the resulting K_D or K_P since the victim has not processed the token at all. Meanwhile, a *false negative noise* implies that the cache hit is not successfully observed. Therefore, false negative noise causes missing tokens in the results.

To identify the noise, we scrutinize the timing signal of the cache trace. Intuitively, the autoregressive token generation can be approximated as *periodic* events, since the forward propagation of LLMs is data-flow-oriented and uses regular pipelines, which differs from control-flow-oriented programs. To validate the periodicity of the autoregressive generation, we study the timing signal in the frequency domain using Fourier transformation. Mainly, we utilize the Power Spectral Density (PSD), which is a measure used in signal processing to describe how the power (or "strength") of a signal is distributed across different frequencies. If the signal exhibits periodicity, the PSD will show a peak.

We model the *timing signal* of cache trace $\mathbf{o}_D$ as a Dirac impulse train $T'_D(t) = \sum_{k=1}^{|T_D|} \delta(t - T_{Dk})$. Then, we can derive the Fourier transform of the timing signal using the sifting property of Dirac functions:

$$F(\omega; T_D) = \int_{-\infty}^{\infty} T'_D(t) e^{-j\omega t} dt = \sum_{k=1}^{|T_D|} e^{-j\omega T_{Dk}}$$

which transforms the timing signal into its frequency domain, and further enables the estimation of PSD. Figure 4 shows the PSD estimation of timing signal corresponding to the decode phases of LLM inference.

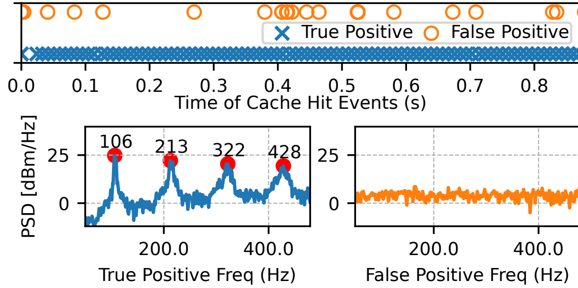


Figure 4: PSD of the timing signal derived from the cache trace $\mathbf{o}_D$. The signal is windowed by the Hann function.

The top of Figure 4 presents the timing signal collected from the llama.cpp that runs Phi-3.5-mini model on an Intel 13900K platform with NVIDIA RTX3060 GPUs. The average time per output token (TPOT) is about 10 microseconds. Thereby, we expect a peak at the frequency of 100Hz in the PSD.

The bottom of Figure 4 shows the PSD of the cache traces respective to the true positives and false positives, from the left to the right, respectively. We clearly observe the expected peak at the base frequency of about $f_0 = 100$Hz and harmonic frequencies at $kf_0, k = 2, 3, \ldots$ in the PSD of the true positive trace. Therefore, we argue that the true positives of the investigated decoding phases exhibit strong periodicity. Meanwhile, in the PSD of false positive trace, the frequency components are relatively evenly distributed across the entire frequency range, implying that false positives are close to white noise.

The PSD offers an effective method for extracting periodic components from the timing signals that contain noise. To further investigate the remaining noise, we now remove the periodic components by computing the normalized first-order difference of the timing signal, as follows:

$$\hat{T}_{D_k} = f_0(T_D) \cdot (T_{Dk} - T_{Dk-1}) \tag{1}$$

Leveraging the PSD, we normalize the first-order difference by multiplying it with the extracted fundamental frequency (f_0). This normalization renders the results that were previously susceptible to hardware-specific TPOT relatively hardware invariant. The base frequency is extracted from the PSD using Sawtooth Waveform Inspired Pitch Estimator Prime (SWIPE') algorithm [25]. The resulting signal is shown in Figure 5.

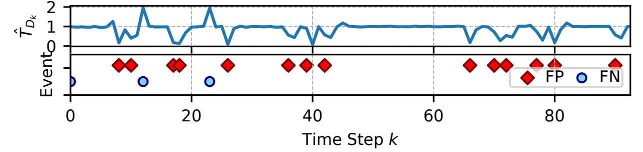


Figure 5: An example of the pre-processed timing signal. False positive (FP) noise likely aligns with the valley. False negative (FN) noise likely aligns with the peak.

We can find that false positive noise correlates with the peaks in the waveform of $\hat{T}_D$. The underlying reason is that true positives are periodic, i.e., are highly likely to occur at regular time points near $kT + \varphi$ (where φ represents the initial phase). Assuming that we remove the i-th true positive to create a false positive, then a gap would appear between $t_1 = (i-1)T$ and $t_2 = (i+1)T$, and therefore the first-order difference would rise to $t_2 - t_1 = 2T$. If such a blank is detected, we can eliminate the corresponding false positive by first predicting its missing token using the context of the preceding and subsequent tokens, then inserting the predicted token to K_D. The token prediction aligns with fill-in-the-blank NLP tasks, which *interpolates* the token sequence.

Similarly, we can find that most false negatives correlate with the valleys of $\hat{T}_D$, because true positives are highly likely to occur at the periodic time points, while false positives are relatively uniformly distributed at the time axis. A false positive occurring between $t_1 = kT + \varphi$ and $t_2 = (k+1)T + \varphi$ will result in a first-order difference $t_2 - t_1 < T$. If a false positive is detected, it should be removed from the K_D.

The analysis implies that, depending on the waveform of $\hat{T}_D$, adversaries must choose one of three actions at each time point: fill a blank, remove a false positive, or leave the token unchanged. We found that this problem can be uniformly formulated as a sequence-to-sequence task. Given the timing signal $\hat{T}_D$ and the token list K_D, the resulting text is:

$$\widehat{O} \sim P(O | \hat{T}_{D_1}, K_{D1}, \hat{T}_{D_2}, K_{D2}, \cdots)$$

where P is implemented by the LLM_A fine-tuned to capture the timing patterns of $\hat{T}_D$ and remove noise.

Training Set Synthesis. We need to synthesize data for the training model P because the adversary cannot access the actual victim models or systems, thereby not being able to profile them to collect the training data. Fortunately, the timing patterns revealed by the PSD enable the automatic synthesis of training samples.

Algorithm 1: Dataset Synthesis

Input: The probability of noise p, the standard deviation of duration σ, the corpus dataset C, and the size of LLM vocabulary $|V|$.
Output: The synthesized training dataset T
// U denotes uniform distribution, $\mathcal{N}$ is normal distribution. $s(\cdot)$ is a sample.
1 **for** $c \in C$ **do**
2   **Init** $L \leftarrow \phi$ // Generated cache trace
3   **Init** $ctime \leftarrow 0$ // Current timestamp
4   **Init** $mtime \leftarrow |c|+1$ // Maximum length
5   **for** $token \in c$ **do**
6   **if** $s(U[0,1]) \geq p$ **then**
7        $L \leftarrow L \cup \{(ctime, token)\}$
8   **end**
9   **if** $s(U[0,1]) < p$ **then**
10        $L \leftarrow L \cup \{(s(U[0,mtime]), s(U[0,|V|-1]))\}$
11   **end**
12      $ctime \leftarrow ctime + s(\mathcal{N}(1,\sigma^2))$ // Simulate the periodicity of decode phases
13   **end**
14    $T \leftarrow T \cup \{(L[0], L[1], c)\}$ // Get a (T_D, K_D, O)
15 **end**

The training set synthesis consists of three steps. First, we collect LLM input and output text pairs (I,O) to form the textual corpus. We query general-purpose LLMs with various prompts from public datasets (like UltraChat) as input I to obtain the model output O. Second, we leverage the periodicity revealed by the PSD (Figure 4) to generate the simulated timing signal T_D. As shown in Algorithm 1, we generate the timestamps of token generation by accumulating Gaussian noise that simulates the minor fluctuations of TPOT. Meanwhile, we randomly add false positives and false negatives with probability p to simulate the noise, and generate K_D Finally, after generating the training samples (T_D, K_D, O), we encode the samples into LLM training pairs (P,R) that consist of prompts and ground-truth responses. We encode the numerical timing signal T_D as textual prompts, similar to the previous work on time series [41, 73]. For example, we encode the $T_D = \{1.3, 1.2\}$ and $K_D = \{'Hello', 'World'\}$ into the following textual prompt:

{(1.3,'Hello'),(1.2,'World')} → '1 3:Hello<s>1 2:World<s>'

LLM Fine-tuning. Having obtained the synthesized training set, we fine-tune the base LLM (e.g., Llama) to obtain the LLM_A. We utilize LoRA to fine-tune the LLM, which allows us to retain most of the knowledge from the pre-trained model while mitigating overfitting. Finally, during the attack, the adversary can obtain the reconstructed model output of the victim via $\widehat{O} = LLM_A(T_D, K_D)$, as shown in Figure 2.

4.5 Reconstructing Model Input

Unlike the output text, reconstructing the input text is more challenging. As mentioned in **C2** (§3.2), the parallel computing of token embedding operation results in *unordered* input tokens in the side-channel attacking results.

The model of unordered tokens aligns with the bag-of-words language model. A bag-of-words model discards the sequential order and assumes that tokens are independent of one another. Without the token order, the bag-of-words model can still reveal latent semantic structures, and is widely applied in the topic models. For instance, Latent Dirichlet Allocation (LDA) automatically identifies topic or theme words from a collection of documents, aiding in understanding the main content of textual data [26].

However, the loss of token order can often impact the linguistic structure and precise semantics of sentences. For instance, the sentences "A chased B" and "B chased A" have the same token set but different semantics. We expect to reconstruct the full text of the model input instead of a series of discrete words.

Fortunately, the LLM input generally correlates with its output in the same context. For example, LLMs tend to repeat the question before proceeding [70]. We formulate the problem of reconstructing model input as a sequence-to-sequence model that restores the positions of each input token. The model is as $\widehat{I} \sim P(I|K_P, \widehat{O})$, where K_P is the unordered set of input tokens, and $\widehat{O}$ is the reconstructed model output, P is implemented by the LLM_B fine-tuned on a synthetic training set.

We encode the token list K_P as a text sequence for the LLM prompt delimited by <s> token. Then, we insert the reconstructed model output $\widehat{O}$ at the beginning of the encoded K_P, providing the context for the model to infer the token order.

Training Set Synthesis. Similar to the strategy of dataset synthesis in §4.4, we randomly shuffle token positions of the prompt I in the corpus dataset to obtain K'_P, which simulates the characteristic of cache trace obtained from the prefill phase. Then, we feed the prompt into a general LLM as its input to obtain the corresponding output text O', and add the training sample (K'_P, O', I) to the resulting dataset, where I is the ground-truth of the input reconstruction.

Fine-tune LLM. We fine-tune the pre-trained LLM to obtain the LLM_B, in the similar process described in §4.4. Finally, during the attack, the adversary can obtain the recon-

structed model input of the victim via $\widehat{I} = LLM_B(K_P, \widehat{O})$, as shown in Figure 2.

5 Experiment Setup

In this section, we introduce the experiment setup of the evaluations for our eavesdropping attack framework.

Environment. We tested the attack on the machine equipped with an Intel(R) 13-Gen 13900K 5.8GHz CPU (with the latest BIOS microcode: 0x129), an NVIDIA 3060 GPU (with 12GiB GPU memory), and 32GiB of dual-channel DDR4-3200 memories on a Maxsun(R) H610ITX baseboard. In §6.5, CPUs (except for the 13900K) were tested on a Gigabyte(R) H610M K DDR4 motherboard equipped with an NVIDIA 3060 GPU and 16GiB of single-channel memory. These machines runs Ubuntu 22.04 (with Linux 5.18.0-38-generic kernel) OS, NVIDIA GPU driver v525.89.02, and CUDA v12.0.

Dataset Construction. We synthesize the dataset for fine-tuning LLM_A and LLM_B via processes described in §4.4 and §4.5. Specifically, we sample prompt text I from UltraChat [35], NQ-Open [49] (belonging to Natural Question), SIQA [61], SQuAD2 [60], and ChatGPT-Roles [1] datasets for our purpose. The synthesized dataset was randomly partitioned into training, validation, and test sets at ratios of 60%, 20%, and 20%, respectively. Details of dataset synthesis settings and data cleaning can be found in Appendix A.

Model Fine-tuning. The fine-tuning is conducted on an AutoDL cloud server with NVIDIA H800 computing card, 300GB CPU memory, and AMD EPYC 9K84 CPU. Hyperparameters of the fine-tuning can be found in Appendix C.

Metrics. We evaluate the fidelity of the reconstructed model input and output using metrics of different granularities: In *character-level*, we used Levenshtein Similarity (LS = 1 - Edit Distance). In *token-level*, we used ROUGE, a measure of n-gram similarity between the reconstructed text and the ground-truth. Specifically, we used R1 and RL metrics, which evaluate F1-score via n-grams and the longest sub-sequence. In *semantic-level*, we employed cosine similarity (ϕ) of sentence embedding to evaluate how effectively the semantics of the text have been reconstructed. We employed the state-of-the-art embedding model AngleIE-Llama-7B [51] to compute the sentence embedding of both the ground-truth text and the reconstructed text, then compute their normalized dot product to obtain the cosine similarity $-1 \leq \phi \leq 1$, where $\phi = 1$ indicates a perfect match.

To objectively determine which value of ϕ indicates a successful attack where the semantics of the text have been leaked, we conducted a survey study inspired by previous works [70]. Using test results, we uniformly sampled 50 sentences across the range $\phi = [0.5, 1.0]$, and recruited 100 random participants on Prolific to vote on whether the attack results accurately capture the privacy contents of the ground-truth sentence. Figure 6 plots the relationship between the human-evaluated privacy exposure and the cosine similarity. This linear relationship indicates that a majority of participants believe privacy information is accurately captured when ϕ is above 0.77. Thus, we define the attack success rate (ASR) as the proportion of testing samples where $\phi > 0.77$.

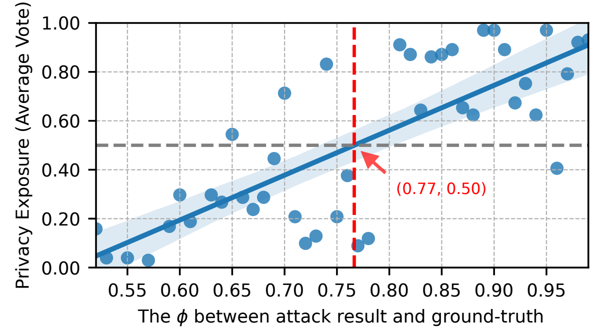


Figure 6: The relationship between human-evaluated privacy exposure and cosine similarity, with a Pearson Correlation of 0.771.

6 Evaluation

In this section, we evaluate the threat of our LLM eavesdropping attack and aim to answer the following Research Questions (RQs):

- RQ1 [Attack Performance] What is the attack performance against different types of victim LLMs in practical systems?
- RQ2 [Parameter Analysis] How do different hyperparameters affect the attack performance?
- RQ3 [Ablation Study] What is the contribution of key components to the attack performance?
- RQ4 [Framework Evaluation] Which LLM frameworks are affected by the side-channel vulnerability?
- RQ5 [Hardware Evaluation] What is the applicability of the attack to other machines?
- RQ6 [Attack Examples] How can concrete examples be used to understand the privacy leakage?

6.1 RQ1: Attack Performance in Practical Systems

In the evaluation of attack performance, we execute the victim and co-located spy process on the victim machine. The victim process runs a real-world LLM framework (llama.cpp in our settings). During experiment, we send testing prompts to the victim process, and wait for the model output, acting

Table 1: Attack performance of each victim LLM, using GPT-4o-mini-2024-07-18 as the base model of LLM_A and LLM_B. N_I and N_O denotes the average number of input and output tokens, respectively. An underline represents the minimum value of each metric, while the maximum value is represented by **bold** text.

Victim LLM	Dataset	Output Reconstruction						Input Reconstruction					
		N_O	R1 (%)	RL (%)	LS (%)	ϕ (%)	ASR (%)	N_I	R1 (%)	RL (%)	LS (%)	ϕ (%)	ASR (%)
Google Gemma2-9B [4]	UltraChat	243	98.2	98.2	97.0	99.6	99.8	20	93.5	90.2	87.4	99.2	**100.0**
	NQ-Open	79	95.9	95.9	94.3	98.7	99.3	13	94.6	93.0	91.3	99.0	**100.0**
	SIQA	193	96.4	96.4	94.2	98.8	99.1	31	86.6	79.2	74.8	96.9	**100.0**
	SQuAD2	55	91.5	91.5	89.8	98.2	**100.0**	183	57.1	47.7	34.4	94.9	**100.0**
	ChatGPT-Roles	222	98.7	98.7	98.0	99.6	**100.0**	48	85.4	79.7	70.6	99.1	**100.0**
Meta Llama-3.1-8B [8]	UltraChat	253	99.0	99.0	98.9	99.2	99.3	19	94.5	91.9	89.5	99.2	**100.0**
	NQ-Open	162	97.4	97.4	96.9	98.1	98.0	12	**94.8**	**93.4**	**91.4**	99.0	**100.0**
	SIQA	64	98.1	98.1	97.6	98.9	99.1	30	86.1	78.5	73.6	96.6	99.7
	SQuAD2	20	90.1	90.1	90.4	96.7	96.4	180	55.8	46.4	33.2	94.3	**100.0**
	ChatGPT-Roles	215	**99.5**	**99.5**	**99.6**	**99.8**	**100.0**	48	86.3	80.7	72.3	99.0	**100.0**
TII Falcon3-10B [3]	UltraChat	175	98.4	98.4	97.3	99.6	99.6	20	**94.8**	92.1	90.2	**99.3**	**100.0**
	NQ-Open	109	98.2	98.1	97.7	99.7	99.9	13	94.3	92.6	91.1	99.0	**100.0**
	SIQA	140	98.9	98.9	97.9	99.7	**100.0**	31	86.2	78.6	75.5	96.7	**100.0**
	SQuAD2	62	90.6	90.6	93.2	98.0	96.4	185	54.6	44.9	33.5	93.8	**100.0**
	ChatGPT-Roles	67	98.9	98.8	99.3	99.6	**100.0**	48	86.8	82.3	73.9	99.0	**100.0**
Mistral-7B [13]	UltraChat	256	94.6	94.6	91.6	98.2	98.7	20	91.6	87.7	84.4	98.7	**100.0**
	NQ-Open	120	95.1	95.1	94.6	97.1	96.8	12	89.1	84.0	80.8	97.3	99.8
	SIQA	65	98.7	98.7	98.2	99.4	99.7	32	85.9	77.6	73.7	96.2	**100.0**
	SQuAD2	57	91.4	91.4	90.1	96.9	98.2	204	51.3	43.2	32.4	92.7	98.2
	ChatGPT-Roles	243	94.6	94.6	91.6	98.9	**100.0**	54	83.2	78.4	69.7	97.9	**100.0**
Microsoft Phi-3.5-mini-3B [12]	UltraChat	263	93.5	93.5	88.9	99.0	**100.0**	21	90.5	87.2	84.3	98.2	99.6
	NQ-Open	194	93.9	93.9	90.9	98.7	99.3	12	88.0	82.9	79.8	97.0	99.8
	SIQA	253	92.7	92.7	87.7	98.5	99.4	33	85.2	78.5	75.4	96.5	99.7
	SQuAD2	137	93.5	93.5	90.6	97.6	98.2	209	51.0	42.4	32.2	92.1	96.4
	ChatGPT-Roles	263	94.6	94.6	92.1	98.8	**100.0**	57	80.6	75.0	65.7	97.6	**100.0**
Average		165	96.3	96.3	94.8	98.7	99.1	24	89.9	85.8	82.7	98.0	99.9

as a regular user. Meanwhile, the spy process performs the eavesdropping attack. The setting parameters of llama.cpp are kept by default, with a batch size of 256 for the prefill. Table 1 shows the resulting performance of attacking different victim models, using GPT-4o-mini as the base model of LLM_A and LLM_B.

[RQ1-1] Reconstruction Accuracy. We first analyze the performance of output reconstruction. Table 1 shows that the ROUGE-1 and ROUGE-L values of the output reconstruction are all higher than 90.1%, and the Levenshtein similarity is no less than 87.7%, which means that our method can restore high-fidelity target output text at both the character and token levels. We can observe that cosine similarity ϕ for all 5 types of victim models is higher than 96.7%, implying that the semantics of text has been leaked. The lowest ASR is 96.4%, indicating that we can successfully infer most of the privacy contents of the tested samples.

We then describe the performance of input reconstruction. As presented in Table 1, the average ROUGE-1 and Levenshtein similarity reach 89.9% and 82.7%, indicating reasonable token-level and character-level reconstruction accuracy. Meanwhile, our method achieves high cosine similarity (the worst is 92.1%) to the ground-truth text. We can observe that the average ASR achieves 99.9%, meaning the topics of 99.9% of tested sentences were leaked.

Nevertheless, we observed a negative correlation between the number of input tokens and the attack performance. In the worst case (Phi-3.5-mini on SQuAD2), the ROUGE-1 and Levenshtein similarity metrics drop to 51.0% and 32.2%. This can be attributed to the reconstruction challenge, which resolves the results in a search space that increases factorially, i.e., on the order of $O(N_I!)$. Nonetheless, the worst cosine similarity reaches 92.1%, since our model can restore the semantics of original sentences using different textual expressions and can still leak the sentences' topic or semantics.

[RQ1-2] Different Prefill Bach Sizes. We investigate the impact of varying prefill batch sizes (b), as shown in Figure 7. As b goes up exponentially, the evaluation metrics settle on stable values with a range of only 4%. However, when b is small, a slight decline in overall accuracy is noted. This

outcome may be attributed to a shift in the training data distribution. Specifically, the synthetic training data involves a random permutation of input tokens, which aligns well with real-world scenarios that usually use larger b values. In contrast, smaller b values necessitate execution through multiple sequential batches when the number of input tokens exceeds b. In such cases, the extracted input tokens T_P display increased sequentiality, deviating from the training data distribution. Fortunately, empirical results suggest that this deviation does not lead to significant performance degradation.

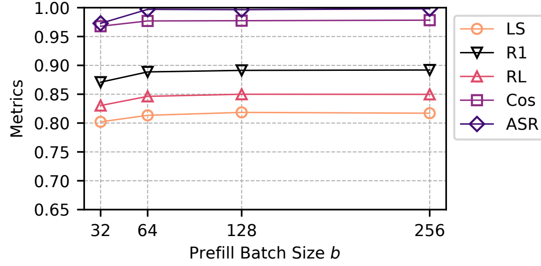


Figure 7: The correlation between the prefill batch size b and the attack performance. We use gpt-4o-mini-2024-07-18 as the base model. The tested victims are Gemma2-9B and Phi-3.5-mini-3B.

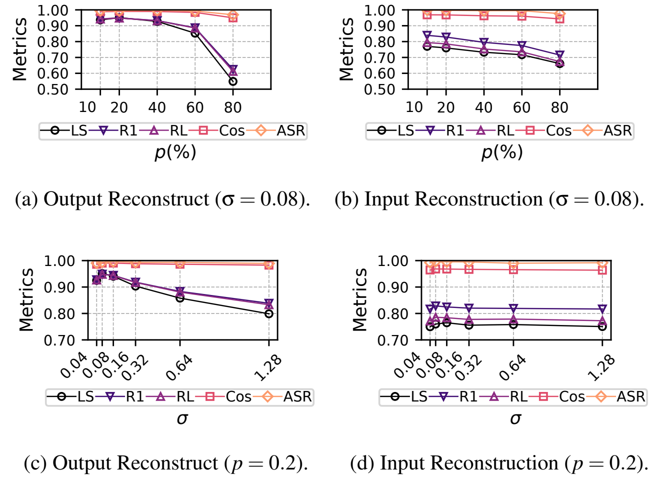
6.2 RQ2: Parameter Analysis

In this research question, we evaluate the impact of hyperparameters in our method.

We have introduced two hyperparameters: p and σ involved in the dataset synthesis (Algorithm 1) for the output reconstruction. To evaluate how these hyperparameters affect attack performance, we conducted a series of training sessions on Llama3.1-8B-Instruct using a set of different hyperparameters. All the evaluations in the following experiments used Gemma2-9B and Phi-3.5-mini as victim models.

[RQ2-1] Different Values of p. We first analyze the sensitivity of the hyperparameter p with σ set to a fixed value (0.08 in our settings). We synthesized several training sets using different σ. For each training set, we fine-tuned a pair of LLM_A and LLM_B, then evaluated these models on the validation set.

Figure 8b shows the evaluation results. As p increases exponentially, we observe that the overall accuracy remains consistently high when p is set in a wide range (10% to 40%). When p reaches larger values (>50%), the overall performance begins to decline. This is likely because more than half of the simulated cache hit events are false positives, introducing additional noise during training and requiring more data for the model to converge. The findings suggest that in practical scenarios, the parameter p can be safely set within the range of 10% to 40%.



(a) Output Reconstruct ($\sigma = 0.08$). (b) Input Reconstruction ($\sigma = 0.08$).

(c) Output Reconstruct ($p = 0.2$). (d) Input Reconstruction ($p = 0.2$).

Figure 8: Correlation between attack performance and the parameter p and σ. We use Llama3.1-8B-Instruct as the base model of LLM_A and LLM_B.

[RQ2-2] Different Values of σ. We then analyze the hyperparameter σ. For our purpose, p is fixed to 0.1, and we fine-tuned Llama3.1 on a set of σ values. Figure 8d shows the testing result on the validation set.

We can observe that the output reconstruction performance peaks at $s = 0.08$ and then declines. This occurs because σ controls the standard deviation of the decode phase period, normalized to $[0,1]$ in Equation 1. A smaller σ implies greater period certainty; $\sigma = 0$ enforces strict periodicity, while $\sigma > 0.5$ weakens periodicity and hinders the model's ability to capture desired patterns. Also, the performance of reconstructing the input is not very sensitive to σ because the output reconstruction only serves as an extra reference context.

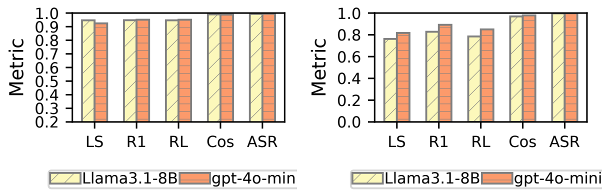
[RQ2-3] Different Base Models. To study how model selection impacts the attack performance, we finetuned and compared two base models: open-source Llama-3.1-8B and close-source GPT-4o-mini.

Figure 9 shows that our LLM_A and LLM_B fine-tuned on two different base models exhibit relatively consistent attack performance. We find that specific model architectures decouple the text reconstruction performance.

6.3 RQ3: Ablation Study

To demonstrate the effectiveness of each component, we conduct the ablation study. We removed each critical attack phase and re-evaluated the attack performance, to demonstrate the contribution of the proposed components.

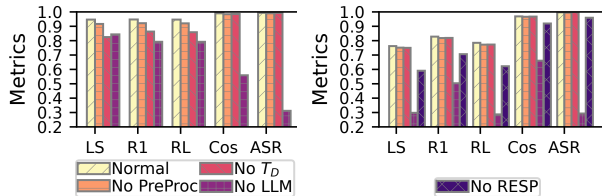
[RQ3-1] Ablation Study of Output Reconstruction. As shown in Figure 10a, after only removing the pre-processing phase (Equation 1) of timing signal T_D, the Levenshtein sim-



(a) Output reconstruction (LLM_A). (b) Input reconstruction (LLM_B).

Figure 9: Comparative evaluation of different base models.

ilarity dropped -3.0% and -1.0% for the output and input reconstruction, respectively. After removing timing signal T_D (do not include T_D in the encoded prompts) in the output reconstruction, the Levenshtein similarity dropped -12.7% (output) and -1.2% (input). These results show that fusing the token list with the timing signal leads to better performance. Additionally, we completely removed the LLM_A (assembling the resulting text from the K_D only) and observed a drop of -10.3% in the average Levenshtein similarity. These results demonstrate the performance gain of our LLM-based text reconstruction model.



(a) Output reconstruction (LLM_A). (b) Input reconstruction (LLM_B).

Figure 10: Ablation study using Llama3.1-8B-instruct as the base model of LLM_A and LLM_B.

[RQ3-2] Ablation Study of Input Reconstruction. To validate the contribution of the reconstructed output in reconstructing the order of K_P, we removed the reconstructed output from the prompt of LLM_B, and observed that the Levenshtein similarity dropped -17.1%. This result confirms the validity of utilizing the reconstruct output to assist in the input reconstruction. Finally, we completely removed the LLM_B, and assembled the extracted K_P as the reconstructing result. We clearly observed that Levenshtein similarity dropped -46.4%, implying the significant performance gain brought by LLM_B. Additional results pertaining to the pure SCA are provided in the Appendix D.

To demonstrate the performance gain of the side-channel attack (SCA) data, we excluded the SCA data and used only the generated output to reconstruct the input. As illustrated in Table 2, we observed that the Levenshtein similarity of the pure output-based attack dropped -21%, while the cosine similarity dropped -5.4%. These results show that the SCA data indeed enhances the performance, particularly at the character and token levels. This improvement is likely due to the SCA's ability to provide concrete tokens that narrow the search space for the target text.

Table 2: Ablation study on the SCA data.

Configuration	Input Recovery Performance (%)				
	R1	RL	LS	ϕ	ASR
LLM_B (SCA + Output)	82.8	78.5	76.2	96.9	99.7
LLM_B (Output)	54.8	51.9	55.2	91.5	95.8

6.4 RQ4: Framework Evaluation

In this research question, we validate the attack in practical scenarios, examining its viability across various LLM frameworks and computing hardware. All victims' settings were maintained at their default values to align with the most probable real-world scenario.

Microbenchmark. To quickly examine whether a target framework or application is vulnerable to the side-channel, we uniformly sampled 20 samples in all the test sets to construct a microbenchmark for performance evaluation.

Results. Table 3 summarizes the results across different LLM inference frameworks. We successfully attacked all 10 frameworks on CPU, 9 of them on GPU. We failed on one GPU target called Transformers because it conducted the embedding lookup on the GPU, which cannot be probed by CPU cache side channels.

Table 3: Evaluation on various LLM inference frameworks and different computing device. Statistics for Github starts are current until January 21, 2025.

Framework	Github Stars	CPU		GPU	
		ϕ_O	ϕ_I	ϕ_O	ϕ_I
LM Studio [54]	N/A†	96.6	97.0	97.4	97.3
HuggingFace Transformers [5]	138k	98.0	74.5	N/A	N/A
Ollama [19]	108k	92.0	95.7	99.7	96.1
llama.cpp [37]	71k	99.5	95.2	99.2	97.8
GPT4All [57]	71k	97.6	95.6	98.9	94.3
LocalAI [10]	28k	99.1	97.6	99.0	96.4
Microsoft BitNet [11]	12k	96.1	76.0	98.3	74.5
PowerInfer [66]	8k	98.0	96.5	98.5	96.2
Intel IPEX-LLM [6]	7k	88.6	93.8	96.6	96.1
koboldcpp [7]	6k	97.6	94.9	99.1	95.5

† LM Studio is close-source.

6.5 RQ5: Hardware Evaluation

We evaluated the same attack across different hardware machines, utilizing the microbenchmark dataset (§6.4) and targeting llama.cpp. Table 4 shows that cosine similarity is at least 96.5%, and the ASR remains 100%. These results are consistent across different hardware configurations, demonstrating the broad applicability of our attack methodology.

Table 4: Evaluation on different hardware machines.

CPU	Output Recovery		Input Recovery	
	ϕ_I (%)	ASR(%)	ϕ_O(%)	ASR(%)
Intel 14900K	99.2	100.0	96.5	100.0
Intel 13900K	99.2	100.0	97.8	100.0
Intel 12700KF	99.3	100.0	96.7	100.0

6.6 RQ6: Attack Examples

In this research question, we present concrete examples to better understand the privacy leakage. Figure 11 illustrates both success and failure samples of prompts. It is intriguing to see that the model perfectly recovered unique n-grams (such as "freddy krueger" and "e5") that were not present in the synthetic training set. These examples highlight the potential to steal Personally Identifiable Information (PII). Additionally, we observed that the model can maintain semantic similarity while replacing synonyms or varying grammar structures; thus, R1 and LS can drop while ϕ remains high in the succeeded samples. For more detailed examples of the results from LLM_A and LLM_B, please refer to Appendix F.

Attacks on Prompts

ϕ : 100% R1: 100% LS: 100% (Succeeded)
who played freddy krueger in the 2010 nightmare on elm street?
who played freddy krueger in the 2010 nightmare on elm street?

ϕ : 98% R1: 96% LS: 87% (Succeeded)
How can I manage my weight and **avoid gaining** excess body fat?
How can I manage my weight and **avoid** excess body fat?

ϕ : 87% R1: 78% LS: 28% (Succeeded)
what rank is an e5 in the air force?
an e5 in the air force is **what rank**?

ϕ : 76% R1: 57% LS: 31% (Failed)
context: Remy enhanced their understanding of the scientific subjects. question:What will happen to Remy?
What will happen to Remy **when they have** enhanced their understanding of scientific subjects?

ϕ : 75% R1: 93% LS: 70% (Failed)
What are the **best times** of year to visit the Grand Canyon for outdoor activities?
Zoeken What **two times** of year are best to visit the Grand Canyon for outdoor activities?

ϕ : 64% R1: 47% LS: 56% (Failed)
who is the national ffa president and where is he from?
headerwhere is fa president and who is he?

Figure 11: Examples of reconstructed prompts.

7 Discussion

7.1 Countermeasures

We provide recommendations on mitigating the proposed attacks during the deployment and implementation of local LLM inference.

Disable Zero-Copy Loading. One feasible defense is to remove the zero-copy model loading that relies on `mmap` in the LLM inference framework. This method can eliminate the shared-memory-based cache attacks such as flush+reload and flush+flush that use `mmap`. However, removing the zero-copy will significantly harm performance in both temporal and spatial. According to our evaluation results, turning off the zero-copy results in 17% performance drops in loading latency in llama.cpp. Moreover, the non-zero-copy version creates nearly 32% extra memory overhead compared to the zero-copy version when launching two or more LLM instances for multiple users simultaneously. Additionally, this defense cannot eliminate shared memory created by page duplication in OS [21, 62].

Deploy Role-Based Access Control. A better mitigation is role-based access control (RBAC), which limits memory page sharing within a safe scope. The RBAC guarantees that only designated programs are permitted to share the memory frames of the model file. We can authenticate programs per session (through kernel hooks via eBPF) to control memory sharing access. Unfortunately, such access control is still lacking in today's local LLM inference frameworks.

Use Hardware-based Mitigation. Intel has developed the Cache Allocation Technology (CAT) for the Xeon sever-grade CPUs, which allows software to control the LLC partition. Based on the CAT, we can isolate the LLM inference frameworks from shared cache resources, eliminating the attacks on LLC (such as flush+reload and prime+probe). However, CAT for LLC is typically unavailable for consumer-grade productions [43], on which local LLMs are mainly deployed.

7.2 Limitation and Future Work

Measurement Challenges. The input reconstruction has relatively low accuracy in character-level and token-level metrics, which is limited by the temporal resolution of the cache attack and the parallel execution characteristic of the prefill phase. However, the attack can still restore the input with high cosine

similarity (94.8% on average) across long text (nearly 6000 characters), resulting in an ASR of 99.1%, i.e., the attack can successfully reveal the semantics of 99.1% sentences under the human-evaluated cosine similarity threshold. This high semantic similarity indicates the potential to directly extract privacy information.

Using Other CPU Side Channels. The attack relies on shared-memory-based cache attacks for distinguishing between different rows in the embedding table. Future work could explore other CPU cache attacks, such as those without shared memory, but needs to address the following *challenges*: (i) Conflict-based attacks (e.g., Prime+Probe) typically offer set-level spatial resolution. Given that the embedding table size is much larger than the way size, each cache set will be mapped to multiple embedding rows, resulting in a search space consisting of many grammatically correct sentences. To mitigate this challenge, future work could leverage LLMs to predict the sentence using inter-sentence context [70]. (ii) Without shared memory, Address Space Layout Randomization (ASLR) and discontinuous page frame allocation can obscure the address mapping between cache sets and embedding rows. To address this challenge, future work may require extra runtime profiling to learn the address mappings.

Attacking LLM Inference that Uses GPU-side Embedding. Discrete GPUs have dedicated caches that are not coherent with the CPU's caches, making traditional CPU cache attacks infeasible for probing the discrete GPU's memory. However, future work may explore state-of-the-art GPU cache attacks, such as Invalidate+Compare [82], to monitor the memory access patterns of the token embedding. Note that Invalidate+Compare is based on set conflicts and thus encounters the previously described *challenge (i)*. Despite this, Invalidate+Compare provides the additional per-set contention intensity, which could potentially reduce the search space. Moreover, NVIDIA drivers allocate GPU page frames with less randomness, favoring physical contiguity and using consistent starting addresses [82]. This deterministic behavior is beneficial for learning the address mapping between cache sets and embedding table rows.

8 Related Work

LLM Prompts or Responses Leakage. Existing studies on breaching the confidence of LLM prompts and responses fundamentally fall under *software-level* vulnerabilities, which can be divided into two categories. One category requires access to the target model that is shared with the victim [29, 32, 47, 65, 75, 83]. In this attack vector, adversaries must trigger requests to the model service and observe its behavior (such as response text [32, 47] or timing [65, 83]) to infer the confidential information. However, one main limitation of these types of attacks is the need for access to the victim model, which can render the malicious requests visible to the victim services. Instead, this paper is the first to reveal both the prompts and responses via the hardware cache side-channel without directly interacting with the victim models.

Another category avoids accessing the victim model, but generally depends on intercepting network traffic or exploiting software vulnerabilities. The keylogging attack on remote AI assistants has demonstrated the passive acquisition of model responses via intercepting encrypted network traffic [70]. This work exploits the packet-length side-channel to infer the response of online AI assistants. Nevertheless, it relies on tapping the network communication between remote LLMs and clients, which has a threat model different from that of local LLMs.

Hardware Side-channel Attacks on Deep Learning. Several studies have demonstrated side-channel information leakage in deep-learning hardware, which aims to extract model structures or parameters [22, 36, 45, 46, 74], to infer classification labels [53, 56, 63, 69], or to infer hardware design of DNN accelerator [38]. However, they fundamentally target *discriminative* Deep Neural Networks (DNNs). Instead, LLMs are generally *generative* and follow the unique autoregressive paradigm. We also found that the cache access patterns of token embedding operation leak token values, and the timing of autoregressive generation leaks token positions. These leakages have still not been explored by previous hardware side-channel works. To the best of our knowledge, this work is the first hardware cache side-channel eavesdropping attack on full text of LLM input and output.

9 Conclusion

This paper presents a novel side-channel attack to steal the model input and output text of local LLMs by leveraging the fundamental characteristics of LLM inference, including the timing of autoregressive generation and the cache access patterns of token embedding computation. To demonstrate the feasibility, we design an eavesdropping attack framework that utilizes a new cross-modality de-noising algorithm to reconstruct the model output from the noisy cache trace. Moreover, we fine-tune the pre-trained LLM to capture the context dependence between model input and output and reconstruct the model input from the shuffled tokens. Finally, to overcome the lack of training data, we propose a new dataset synthesis process to obtain the training set without profiling the victim. Our empirical evaluations across a range of mainstream LLMs demonstrated that the attack can restore high-fidelity text of model output (with an average Levenshtein similarity of 94.8%), and reconstruct the model input with highly similar semantics (with an average cosine similarity of 98.0%). Our results reveal critical vulnerabilities in widely used local LLM inference frameworks (e.g., llama.cpp), highlighting the pressing necessity of improving security measures to defend against such risks.

10 Acknowledgement

We would like to thank the organizers, anonymous reviewers, and shepherd for their constructive comments and helpful feedback.

11 Ethics Considerations

All datasets utilized in our experiments are publicly available and do not contain any harmful content. Our experiments were conducted exclusively within rigorously controlled environments, ensuring no disruption to other public systems.

In adherence to best practices for vulnerability disclosure, we have reported the identified issues to all relevant software developers and are actively collaborating with them to mitigate the vulnerabilities.

12 Open Science

We have made our research artifact publicly available on Zenodo for permanent retrieval: https://doi.org/10.5281/zenodo.15610475. The artifact contains:

- **Complete Datasets**. Including the corpus dataset, the synthesized datasets, and the collected cache traces.
- **Full Source Code**. Including the implementation of the proposed attack and the evaluation scripts or tools for experiment replication.
- **Models**. For the Llama-3.1-8B-Instruct model, full adapter weights and configuration files are provided. For the fine-tuned OpenAI's GPT-4o-mini, we are bound by proprietary restrictions that prevent sharing the model weights. As an alternative, we release the original JSONL training data files, scripts, and documents necessary to reproduce this model.
- **Evaluation Data**. We provide the original experiment results and anonymous survey results.
- **Documentations**. A comprehensive README file is available, which guides other researchers to reproduce and build upon our work.

References

[1] Chatgpt-roles. https://huggingface.co/datasets/WynterJones/chatgpt-roles.

[2] Efficient PyTorch: Tensor memory format matters. https://pytorch.org/blog/tensor-memory-format-matters/.

[3] Falcon3. https://falconllm.tii.ae/falcon3/index.html.

[4] Gemma. https://ai.google.dev/gemma.

[5] HuggingFace Transformers: State-of-the-art machine learning for jax, pytorch and tensorflow. https://github.com/huggingface/transformers.

[6] Intel IPEX-LLM: Accelerate local llm inference and finetuning. https://github.com/intel-analytics/ipex-llm.

[7] Koboldcpp: Run gguf models easily with a koboldai ui. one file. zero install. https://github.com/intel-analytics/ipex-llm.

[8] Llama. https://www.llama.com/?ref=hacktheprompt.

[9] Llama everywhere. https://www.llama.com/docs/llama-everywhere.

[10] LocalAI: The free, open source alternative to openai, claude and others. https://github.com/mudler/LocalAI.

[11] Microsoft BitNet: Official inference framework for 1-bit llms. https://github.com/microsoft/BitNet.

[12] Microsoft phi-3.5 mini. https://ai.azure.com/explore/models/Phi-3.5-mini-instruct/version/6/registry/azureml.

[13] Mistral. https://mistral.ai.

[14] Openai chatgpt. https://chatgpt.com/.

[15] PEP 3154 - pickle protocol version 4. https://peps.python.org/pep-3154/.

[16] Safetensors. https://huggingface.co/docs/safetensors/index.

[17] The new era of the ai pc: What does the ai pc mean for users, and how does it work for them? Technical report, Intel, 2023.

[18] Hugging face transformers - cpu inference. https://huggingface.co/docs/transformers/perf_infer_cpu, 2024.

[19] Ollama: Get up and running with large language models locally. https://github.com/ollama/ollama, 2024.

[20] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 34–46, 2020.

[21] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28. Citeseer, 2009.

[22] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 515–532, Santa Clara, CA, August 2019. USENIX Association.

[23] bloomberg. 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, 2023.

[24] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, and et al. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[25] Arturo Camacho and John G. Harris. A sawtooth waveform inspired pitch estimator for speech and music. *The Journal of the Acoustical Society of America*, 124(3):1638–1652, 09 2008.

[26] Uttam Chauhan and Apurva Shah. Topic modeling using latent dirichlet allocation: A survey. *ACM Comput. Surv.*, 54(7), September 2021.

[27] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. GoFetch: Breaking Constant-Time cryptographic implementations using data Memory-Dependent prefetchers. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1117–1134, Philadelphia, PA, August 2024. USENIX Association.

[28] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, and et al. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2023.

[29] Junjie Chu, Zeyang Sha, Michael Backes, and Yang Zhang. Reconstruct your previous conversations! comprehensively investigating privacy leakage risks in conversations with gpt models, 2024.

[30] Joan Daemen. Aes proposal: Rijndael, 1999.

[31] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[32] 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 (USENIX Security 24)*, pages 6861–6848, Philadelphia, PA, August 2024. USENIX Association.

[33] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 7750–7774. PMLR, 2023.

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[35] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations. *arXiv preprint arXiv:2305.14233*, 2023.

[36] Yansong Gao, Huming Qiu, Zhi Zhang, Binghui Wang, Hua Ma, Alsharif Abuadbba, Minhui Xue, Anmin Fu, and Surya Nepal. Deeptheft: Stealing dnn model architectures through power side channel. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3311–3326, 2024.

[37] Georgi Gerganov. ggerganov/llama.cpp: Port of Facebook's LLaMA model in C/C++. `https://github.com/ggerganov/llama.cpp`, 2024.

[38] Cheng Gongye, Yukui Luo, Xiaolin Xu, and Yunsi Fei. Side-channel-assisted reverse-engineering of encrypted dnn hardware accelerator ip and attack surface exploration. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4678–4695, 2024.

[39] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2016.

[40] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[41] Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew G Wilson. Large language models are zero-shot time series forecasters. In *Advances in Neural Information Processing Systems*, volume 36, pages 19622–19635. Curran Associates, Inc., 2023.

[42] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024.

[43] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11):0–40, 2011.

[44] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[45] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 385–399, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

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

[48] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. ASIA CCS '16, page 353–364, New York, NY, USA, 2016. Association for Computing Machinery.

[49] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

[50] 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 '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Xianming Li and Jing Li. AoE: Angle-optimized embeddings for semantic textual similarity. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1825–1839, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[52] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, and et al. Jamba: A hybrid transformer-mamba language model, 2024.

[53] Jialin Liu, Houman Homayoun, Chongzhou Fang, Ning Miao, and Han Wang. Side channel-assisted inference attacks on machine learning-based ecg classification. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[54] Lms. LM Studio: Discover, download, and run local llms. https://lmstudio.ai/, 2024.

[55] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[56] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell Tessier. Remote power side-channel attacks on bnn accelerators in fpgas. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1639–1644, 2021.

[57] Nomic. GPT4All: A free-to-use, locally running, privacy-aware chatbot. no GPU or internet required. https://gpt4all.io/index.html?ref=localhost, 2024.

[58] OpenAI. Learn about language model tokenization. https://platform.openai.com/tokenizer.

[59] Dag Arne Osvik, A. Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. *The Cryptographer's Track at RSA Conference*, pages 1–20, 2006. 1482 cites:.

[60] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting*

of the Association for Computational Linguistics (Volume 2: Short Papers), Melbourne, Australia, July 2018. Association for Computational Linguistics.

[61] Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. Social IQa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China, November 2019. Association for Computational Linguistics.

[62] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, page 15–26, New York, NY, USA, 2012. Association for Computing Machinery.

[63] Shubhi Shukla, Manaar Alam, Sarani Bhattacharya, Pabitra Mitra, and Debdeep Mukhopadhyay. "whispering mlaas" exploiting timing channels to compromise user privacy in deep neural networks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):587–613, 2023.

[64] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, Santa Clara, CA, August 2019. USENIX Association.

[65] Linke Song, Zixuan Pang, Wenhao Wang, Zihao Wang, XiaoFeng Wang, Hongbo Chen, Wei Song, Yier Jin, Dan Meng, and Rui Hou. The early bird catches the leak: Unveiling timing side channels in llm serving systems, 2024.

[66] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU. *CoRR*, abs/2312.12456, 2023.

[67] Lawrence Spracklen and Subutai Ahmad. Supercharged ai inference on modern cpus. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, pages 1–21, 2023.

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[69] Han Wang, Syed Mahbub Hafiz, Kartik Patwari, Chen-Nee Chuah, Zubair Shafiq, and Houman Homayoun. Stealthy inference attack on dnn via cache-based side-channel attacks. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1515–1520, 2022.

[70] Roy Weiss, Daniel Ayzenshteyn, and Yisroel Mirsky. What was your prompt? a remote keylogging attack on AI assistants. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3367–3384, Philadelphia, PA, August 2024. USENIX Association.

[71] Lim Xiang Yang, Lim Chen Han, and Foong Chun Sheong. Optimizing and running LLaMA2 on Intel CPU. Technical report, Intel Inc., 2023.

[72] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Fast on-device llm inference with npus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 445–462, New York, NY, USA, 2025. Association for Computing Machinery.

[73] Hao Xue and Flora D. Salim. Promptcast: A new prompt-based learning paradigm for time series forecasting. *IEEE Transactions on Knowledge and Data Engineering*, 36(11):6851–6864, 2024.

[74] 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. USENIX Association, August 2020.

[75] Yong Yang, Changjiang Li, Yi Jiang, Xi Chen, Haoyu Wang, Xuhong Zhang, Zonghui Wang, and Shouling Ji. Prsa: Prompt stealing attacks against large language models, 2024.

[76] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. *XLNet: generalized autoregressive pretraining for language understanding*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[77] Yuval Yarom. Mastik: A micro-architectural sidechannel toolkit. https://cs.adelaide.edu.au/~yval/Mastik/.

[78] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 719–732. USENIX Association, 2014.

[79] Yizhen Yuan, Rui Kong, Yuanchun Li, and Yunxin Liu. WiP: An On-device LLM-based approach to query privacy protection. In *Proceedings of the Workshop on*

Edge and Mobile Foundation Models, EdgeFM 2024, Minato-ku, Tokyo, Japan, June 3-7, 2024, pages 7–9. ACM, 2024.

[80] Juntao Zeng, Bo Chen, Yuandan Deng, Weiqin Chen, Yanlin Mao, and Jiawei Li. Fine-tuning of financial large language model and application at edge device. In *Proceedings of the 3rd International Conference on Computer, Artificial Intelligence and Control Engineering*, CAICE '24, page 42–47, New York, NY, USA, 2024. Association for Computing Machinery.

[81] Shiquan Zhang, Ying Ma, Le Fang, Hong Jia, Simon D'Alfonso, and Vassilis Kostakos. Enabling on-device llms personalization with smartphone sensing. In *Companion of the 2024 on ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '24, page 186–190, New York, NY, USA, 2024. Association for Computing Machinery.

[82] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. Invalidate+Compare: A Timer-Free GPU cache attack primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2101–2118, Philadelphia, PA, August 2024. USENIX Association.

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

A Dataset Construction Process

We used all the 2,727 testing prompts of UltraChat, and 4,425 samples in the NQ-Open, 254 prompts (12,394 tokens) in the ChatGPT-Roles, 1682 prompts in the SIQA, and 277 prompts (19,029 tokens) in the SQuAD2. The number of total tokens is 212535.

To obtain ground-truth LLM output corresponding to different input prompts, We send each prompt to the the general-purpose LLM (named Llama3.1-8B-Instruct for our purpose) as its input in the separated context (independent with its preceding prompts).

B Dataset Cleaning

To prevent data contamination of the training dataset from the testing samples, we dropped all the contaminated samples from the training set. Similar to GPT-3 and PaLM [24, 28], we consider a training sample as contaminated if either its user prompts has more than 70% 8-grams overlapping with the user prompt of testing set, or it shares at least the first 8 words with one of the testing samples.

C Hyperparameters of Fine-tuning

We fine-tune OpenAi GPT-4o-mini-2024-07-18 in 3 epochs, each with a batch size of 5. We set the learning rate multiplier as 1.8.

We fine-tune Llama-3.1-8B-Instruct in 3 epochs for LLM_A and 2 epochs for LLM_B, each with a batch size of 2. We set the learning rate as 0.0002 for LLM_B and 0.00008 for LLM_A.

D Additional Results

Impact of Embedded Vector Quantization. We evaluated the attack on other embedding vector quantization types beyond F16. The experiment setup was aligned with §6.4. The attack targets the llama.cpp that uses GPU acceleration. For Q8_0, we skip L1 cache hits to omit extra prefetching.

Table 5 shows that the quantization has a minor impact on the cosine similarity.

Table 5: Impact of embedded vector quantization.

Quantization	Output Reconstruction		Input Reconstruction	
	ϕ (%)	ASR (%)	ϕ (%)	ASR (%)
F16	99.2	100.0	97.8	100.0
Q8_0	98.6	100.0	96.6	100.0
BF16	99.3	100.0	96.5	100.0
F32	98.7	100.0	96.2	100.0

Other Operating Systems. We evaluated the attack on other operating systems, leveraging the same model and datasets in §6.4 and fixing the victim as llama.cpp that uses GPU acceleration. Table 6 shows that the attack is applicable to a range of operating systems.

Table 6: Evaluating the attack against other systems.

Operation System	Output Recovery		Input Recovery	
	ϕ (%)	ASR (%)	ϕ (%)	ASR (%)
Ubuntu 22.04	99.2	100.0	97.8	100.0
Windows 11	99.4	100.0	95.4	100.0
Debian 12 (in Docker)	97.0	100.0	96.3	100.0

Input Reconstruction via Pure SCA. Experiments show that the proposed attack significantly outperforms the pure SCA, as shown in Table 7. The setups were aligned with §6.3.

Table 7: Ablation study on the SCA data.

Configuration	Input Recovery Performance (%)				
	R1	RL	LS	ϕ	ASR
LLM_B (SCA + Output)	82.8	78.5	76.2	96.9	99.7
Pure SCA	50.4	28.7	29.8	66.1	29.3

E Detailed Implementation of Flush+Reload

In this section, we detail the implementation for obtaining cache traces using the pseudocode presented in Listing 1.

Listing 1: C++ pseudo code of the flush+reload attack

```
1  probe(llm_model_file) {
2    //1. Achieve the shared memory (detailed
         error handling logics were omitted here
         )
3    fd = open(llm_model_file, O_RDONLY);
4    fstat(fd, &st_buf); // get file size
5    image_ptr = mmap(NULL, st_buf.st_size,
        PROT_READ, MAP_PRIVATE, fd, 0);
6
7    //2. Get the params of embedding table
8    [emb_ptr, emb_stride, num_vocabs] =
         parse_model_file(image_ptr);
9
10   //3. Generate the target address ranges
11   sorted_seg = {};
12   for(v = 0; v < num_vocab; v++) {
13     pstart = emb_ptr + v * emb_stride[1];
14     pend = pstart + emb_stride[1] - 1;
15     sorted_seg = sorted_seg ∪ {struct
           AddressRange(v, pstart, pend)}
16   }
17
18   //4. Choose target addresses that satisfy
         the constraints to overcome prefetchers
19   visited = {};
20   maxpage = page_offset(emb_ptr + emb_stride
         [1] * num_vocab + emb_stride[1]-1) +
         PAGE_SIZE;
21   for (cp = page_offset(emb_ptr))
22     cp <= maxpage; cp += PAGE_SIZE) {
23     p_center = cp + PAGE_SIZE/2;
24     // find target address from ranges
25     binary_search target in sorted_seg,
           maximizing(target.start) s.t. target.
           start <= p_center
26     if(target is found && target->vocab not
           in visited) {
27       visited += {target->vocab};
28       target_addr[target->vocab] = p_center;
29     }
30   }
31
32   //5. Avoid storing valid pointers in the
         array, to overcome AoP prefetchers
33   for (size_t v = 0; v< num_vocab; ++v)
34     ptr_offset[v] = (target_addr[v] - emb_ptr
           )
35
36   //6. Collect the cache trace
37   cache_trace = {};
38   num_partitions = 15;
39   for (t = 0; t < num_partitions; ++t) {
40     // Conventional settings for the flush+
           reload attack
41     clone(thread);
42     pthread_setaffinity_np(pthread_self(),
           mask(t));
43     sched_setscheduler(sched_get_priority_max
           (SCHED_OTHER));
44     clflush_all_the_target_addr();
45
46     m = ceil(num_vocabs / num_partitions)
47     n = (t == num_partitions - 1) ?
           num_vocabs % m : m;
48     while(running) {
49       for (v = 0; v < n; ++v) {
50         // Pseudo random permutation
51         s = t * m + (v * 167 + 13) % n; //
               Two random coprime integers
52         register p = ptr_offset[s];
53         asm {
54           // restore the pointer
55           add emb_ptr, p
56           // reload
57           mfence
58           rdtsc
59           lfence
60           mov %eax,%esi
61           mov %edx,%edi
62           mov (p),%ax
63           mfence
64           rdtsc
65           clflush (p)
66         }
67         timepoint = UINT64(%eax,%edx);
68         cache_trace[timepoint][s] = timepoint
               - UINT64(%esi,%edi);
69       }
70     }
71   }
72   return cache_trace;
73 }
```

F Attack Examples and Failure Modes

Additional concrete examples, along with an analysis of failure modes of LLM_A and LLM_B, were made publicly available for permanent retrieval in the permanlink: DOI 10.5281/zenodo.15646979.