

Leaky Language Models: Stealing Architecture and Inference Optimizations via Per-Token Timing

Sadegh Majidi
Purdue University
West Lafayette, IN, USA
mmajidiy@purdue.edu

Niloofer Miresghallah
Carnegie Mellon University
Pittsburgh, PA, USA
niloofer@cmu.edu

Kazem Taram
Purdue University
West Lafayette, IN, USA
kazem@purdue.edu

Abstract

This work presents LeakyLMs, a set of attacks that leak proprietary model, architecture, and deployment information from production language models. LeakyLMs is the first to demonstrate that key model and deployment details can be inferred using only token generation timing, even when interacting through remote APIs. LeakyLMs introduces two core attacks. The first attack targets inference optimizations and deployment strategies. For example, our attack detects whether a provider uses speculative decoding, a widely deployed inference-time optimization, and further identifies the context length of the draft model used in the pipeline. Our measurements show that Google Gemini Flash 2.5 uses speculative decoding with a draft context window of approximately 128K tokens. The second attack recovers key architectural properties, including the number of transformer layers, hidden dimension size, and number of attention heads. To achieve this, LeakyLMs builds a detailed and accurate model of token-generation timing on modern NVIDIA GPUs, characterizing how latency scales with model configuration and hardware parameters. The attack then performs a search over the architecture space using this timing model. In experiments with Llama models, the near-correct architectural configuration appears in the top-10 guesses more than 90% of the time.

Keywords

Timing Side-Channels, Large Language Models

1 Introduction

There is an ongoing global race across industry and governments to achieve superior AI performance and dominate the rapidly expanding AI market [19, 61]. In this competitive landscape, model architecture and deployment strategies are valuable assets that provide competitive advantage. As a result, providers do not disclose the details of their most advanced models or their serving infrastructure, keeping them highly confidential [42]. For instance, although OpenAI recently released an open-source model (gpt-oss) [44], it scores significantly below their flagship model, GPT-5, on intelligence benchmarks [4], and the architecture, training scale, and deployment details of GPT-5 remain largely undisclosed.

At the same time, providers heavily optimize the token-generation pipeline for low-latency streaming, with minimal tolerance for delay, to ensure high QoS and a seamless user experience [39]. This exposes fine-grained, controlled, and accurate per-token generation timings of the model to external observers. These per-token timings depend on the architectural parameters (e.g., depth, hidden dimension size, and attention heads), inference-time optimizations

(e.g., speculative decoding), and deployment details (e.g., batching strategy, prompt caching, and hardware), many of which are proprietary and not publicly disclosed.

Therefore, careful observation of differential token timing, that is, comparing the latency between consecutive tokens, which cancels shared noise such as network delay, allows an adversary to extract meaningful signals. When combined with publicly known information, for example, that these systems are transformer-based or employ specific inference optimization techniques, this fine-grained timing data enables attackers to infer the otherwise undisclosed architectural and deployment details.

This paper presents the first attacks of this kind. We demonstrate, for the first time, that it is possible to extract sensitive model and deployment information from production LLMs solely by observing token-generation timing. In particular, we introduce two concrete attacks. The first targets deployment details, revealing whether a provider employs speculative decoding [38], a common inference-time optimization, and uncovering specific parameters of the underlying draft model used in this optimization. Our second attack targets the internal model parameters such as the number of decoder layers, attention heads, and the hidden dimension size.

Just as hardware optimizations such as caching or speculative execution create timing channels, inference-time optimizations can introduce similar vulnerabilities [25]. Speculative decoding, for example, is an inference-time optimization that uses a smaller and faster draft model to produce multiple candidate tokens, and then invokes the main model to verify them. If the draft and main model disagree, the main model falls back to normal generation and produces the next token; if they agree, multiple tokens are accepted at once, improving throughput.

Our first attack leverages this behavior by crafting prompts that control how useful the draft model is. The draft model typically has a smaller context window, so when the correct prediction depends on distant context, the draft model will disagree with the main model. By varying prompt conditions to induce or avoid such disagreements, we produce measurable changes in token-generation timing that reveal the presence of speculative decoding, as well as the context length used by the draft model.

Our second attack targets model architectural parameters such as the number of decoder layers, hidden dimension size, and number of attention heads. The attack is based on the intuition that token-generation timing forms a time series with predictable relationships to these architectural parameters. For instance, the hidden dimension size has a roughly quadratic impact on attention computation time, while the number of attention heads influences the latency linearly across tokens.

We first construct an accurate model of the timing behavior of transformer architectures to capture how architectural parameters affect per-token generation latency. We begin with an analytical examination of the transformer computation graph to derive the theoretical asymptotic relationships between runtime and key parameters. We then empirically refine this model by collecting runtime measurements across a range of configurations and fitting a linear regression model that combines the theoretical terms with empirically estimated constants, giving us a precise, explainable, and generalizable model of timing behavior. We then use this model as an oracle to perform a grid search over possible architectural configurations, identifying the parameters that most closely reproduce the observed timing pattern of an unknown target model.

Our results show that both attacks successfully leak critical information through timing analysis. The first attack reveals that Google Gemini models Flash 1.5, Flash 2.5, and Flash 2.5 Lite all employ speculative decoding. Moreover, we are able to determine the context length of the draft models used in these systems—32 K, 128 K, and 128 K tokens, respectively.

Our second attack infers key architectural parameters, including the number of layers, hidden dimension size, and number of attention heads of unseen models. We show that this approach accurately models token-generation latency and enables recovery of architectural parameters across multiple transformer implementations, including baseline eager execution [63, 67], FlashAttention2 [17], and optimizations such as KV-cache [16, 36]. Our timing predictor generalizes to unseen architectures, achieving a normalized root mean squared error (NRMSE) of 0.12 for eager transformer implementations and an NRMSE of 0.188 for predicting prefill time of a model using FlashAttention2 with KV caching. Using this predictor, we recover the number of layers of unseen architectures within ± 1 of the true value, with top-5 accuracy of 86.15%, 97.69%, and 83.78% for eager, FlashAttention2, and FlashAttention2 with KV caching, respectively. Finally, we demonstrate that the attack can recover parameters of an unknown model hosted on an inference serving platform (*Weights & Biases*). The predicted hidden dimension matches the ground truth, and the correct number of layers appears as the fourth-ranked candidate in our classifier.

Responsible Disclosure. We disclosed our speculative decoding attack to Google on Nov 7, 2025. Google acknowledged our findings on Jan 29, 2026.

2 Background

2.1 Transformer Architecture

The Transformer architecture [63] is the foundation of nearly all modern LLMs, such as GPT [15], Gemini [56], and LLaMA [62]. Decoder-only variants [48, 49], optimized for next-token prediction, are the basis of most generative LLMs today. As shown in Figure 1, a decoder-only transformer consists of a closed loop of repeated decoder blocks—we denote their count as L —along with an *Embedding* block at the input and a *projection* block at the output. Each decoder block is composed of several subcomponents, the most prominent being *Attention*, *MLP*, and *Normalization*. While numerous variants, particularly of the attention mechanism (e.g., GQA [3]), have been introduced, their underlying computational structure remains largely similar.

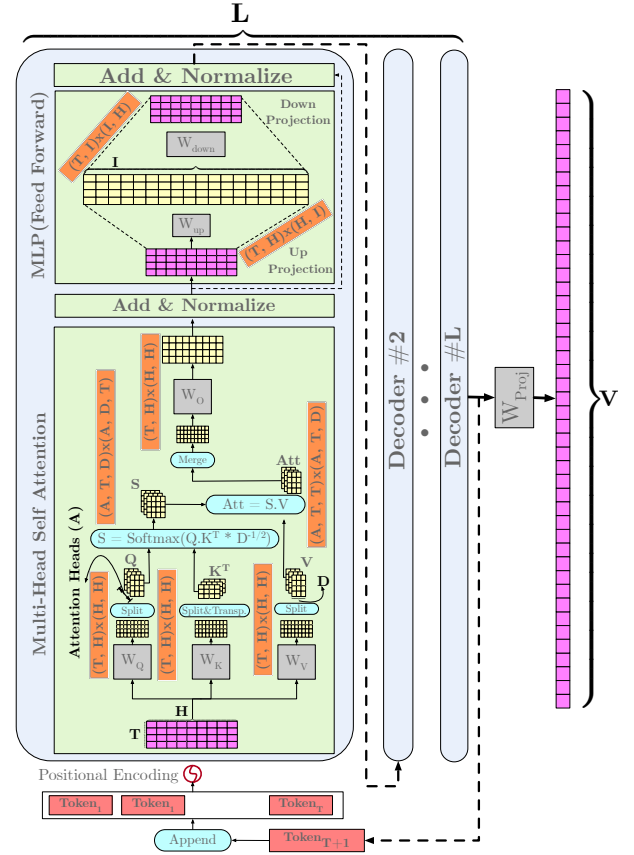


Figure 1: The architecture of a decoder-only transformer. The amount of computation (matmul dimensions) required for generation of each token depends on 5 key parameters: H , L , A , I , and T .

The Transformer processes an input sequence of T tokens, each represented by a hidden vector of dimension H , which serves as the primary dimension propagated through the network. The sequence matrix is projected into multiple attention heads—their count denoted by A —to compute attention scores, followed by an MLP subcomponent that expands the representation to a larger intermediate dimension— I —before projecting it back to H . Normalization layers are interleaved between major components. The embedding and projection layers map the tokens between the one-hot token representation of length equal to the vocabulary size V (typically public) and the hidden dimension space.

Overall, these five parameters— H , L , A , I , and T —dictate most computations within a decoder-only LLM. Among them, the first four (H , L , A , I) define the model’s architectural configuration, while the fifth (T) is externally controlled by the user through the prompt length. We refer to these parameters extensively throughout §5.

Early implementations of attention on modern GPUs simply decomposed the computation into matrix multiplications executed using cuBLAS-based [43] GPU kernels. Later works have focused on improving the efficiency of attention through optimized memory access patterns, operator fusion, and better utilization of modern

GPU hardware features. A prominent example is the FlashAttention family of kernels [17, 18], which leverages on-chip SRAM as a software-managed cache to reduce memory traffic and compute the softmax operation more efficiently.

In this work, we consider both implementations. We use a standard implementation based on native PyTorch linear algebra primitives provided by the HuggingFace Transformers library [67], which we refer to as *eager attention*. For optimized implementation, we use FlashAttention2 as a representative baseline in our evaluations in §5.

2.2 Autoregressive Decoding

The Transformer architecture generates one token per forward pass by computing a conditional probability distribution over the vocabulary, given the starting sequence of tokens for each pass. More precisely, for a sequence of tokens $x_1, x_2, \dots, x_T$ at the start of pass $T + 1$, a Transformer-based LLM predicts the next token x_{T+1} by estimating $P(x_{T+1} \mid x_1, x_2, \dots, x_T)$ and then sampling a token from the resulting distribution. The newly generated token is appended to the end of the current sequence, and the process repeats iteratively to produce subsequent tokens. This property also allows users to observe the model’s output as a continuous stream of tokens, eliminating the need to wait for the entire response to be generated before accessing it. Although this approach is conceptually simple and effective, it is computationally expensive. As model sizes continue to increase and larger context lengths are introduced, the latency of generating each token grows significantly, even taking several seconds per token for large models. As a result, a wide range of optimization techniques has been introduced to reduce token-generation latency [18, 29, 36].

2.3 Key-Value Caching

Caching key–value (KV) pairs is a widely adopted optimization for improving inference efficiency [36]. During generation, the model reuses previously computed attention keys and values for past tokens, avoiding redundant recomputation at each step. As a result, inference is typically divided into two stages: *Prefill*, where the model processes the entire input prompt and constructs the KV cache, and *Decoding*, where one token is generated per forward pass using the cached representations. The prefill stage is compute-intensive due to full-sequence processing, while the decoding stage is typically memory bandwidth–bound, as it incrementally attends to the growing KV cache to generate a single new token per step. This distinction leads to different runtime characteristics across the two stages, which we carefully account for in our modeling in §5.

2.4 Speculative Decoding

The observation that autoregressive models can verify subsequences of their input tokens in parallel with the next-token prediction [54] has led to a new class of optimization techniques [8, 22, 34, 50, 51]. A prominent example is speculative decoding [7, 12, 14, 26, 32, 38, 41, 53, 55, 68], a technique inspired by speculative execution in processors [27]. This optimization relies on a collaboration between two differently sized models: a *main* target LLM, which is large and capable, and a *draft* model, which is smaller, faster, and has modest generation quality. In each generation round, the draft model

proposes a sequence of candidate tokens, which are then verified by the main model in a single forward pass. Depending on the level of agreement between the two models, multiple tokens may be accepted in one pass, reducing the number of expensive invocations of the main model in the average case. Thus, speculative decoding does not uniformly accelerate every generation, but it substantially improves the common-case inference performance. Crucially for this work, the interaction between the draft and main models introduces input-dependent timing variation, creating a timing side channel that we exploit to detect the optimization and recover its deployment details.

3 Threat Model

Goals. We assume an adversary who seeks to infer architectural and deployment information about a production large language model (LLM) by observing its per-token generation timing. The adversary’s goal is to recover information about any or all of the following key architectural parameters: the number of decoder layers (L), hidden dimension size (H), number of attention heads (A), and the intermediate size of the MLP layer (I). Additionally, the adversary aims to identify inference-time optimization techniques employed by the provider, particularly the use of speculative decoding [38] and the context length of the draft model used.

Target Model. We assume the target model is a decoder-only [48] transformer-based [63] model accessible through a standard public web interface or API with streaming generation enabled. In this setting, the model transmits response tokens to the user as they are generated, rather than returning the complete output after generation is finished.

Capabilities. The attacker can observe and measure the timing of the tokens received from the model API. The adversary operates entirely within the standard user interface exposed by the provider. We do not assume any additional control over request or generation parameters, nor access to internal model data such as logits, activations, or system logs. The attack also requires no privileged network position or system-level access. The adversary has no prior knowledge of the target model’s internal architecture or deployment configuration beyond publicly available information, such as the model name, family, advertised capabilities, or maximum context length. For the model architecture leakage attack described in §5, we assume that the attacker knows the GPU type used to execute the target model and has access to timing data collected on the same GPU class to train the runtime predictors. The attack further assumes a single-GPU inference setting and does not model multi-GPU execution.

4 Leaking Inference Optimizations

As large language models scale to billions of users and ever-larger architectures, inference has become increasingly compute-intensive and resource-constrained. Providers must balance model quality, latency, and energy efficiency to sustain acceptable Quality of Service (QoS) at global scale. To achieve this, they aggressively optimize the inference pipeline through specialized hardware [2], custom serving infrastructure [45], and proprietary inference-time techniques [45]. These optimizations may be valuable intellectual property that gives

leading providers a competitive advantage as the market expands and new entrants emerge [5]. Maintaining the confidentiality of these details is therefore a strategic priority, particularly for frontier models operated at massive scale. In this section, we demonstrate that these proprietary deployment and optimization details are vulnerable to timing side channels. Specifically, we present an attack that shows an external observer can infer the presence of speculative decoding in production systems and further estimate key hyperparameters, including the context length of the draft model used in the speculative decoding pipeline.

4.1 Attack Overview

As discussed in Section 2, speculative decoding is an inference optimization technique inspired by speculative execution in modern processors [27]. It aims to reduce end-to-end token-generation latency by using a small, fast draft model to speculatively predict N draft tokens, which are then verified in a single step by a larger target model [38]. If the outputs of the draft and target models agree, all N tokens are accepted, and the draft model proceeds to generate the next speculative tokens. This process reduces the number of full forward passes of the large model by approximately a factor of N in the best case, when the smaller model’s predictions match with the larger model. In contrast, if the target model disagrees with the draft, only one token (the large model’s next token) is accepted, and the remaining draft tokens are discarded, resulting in a worst-case latency comparable to that of standard non-speculative decoding.

Inspired by speculative execution vulnerabilities in modern processors [33], we exploit the input-dependent timing variations that arise from disagreement between the draft and target models during speculative decoding. The key idea is to intentionally induce controllable disagreement between the two models through carefully crafted prompts. If such disagreement exists, it should cause observable slowdowns in per-token generation time, revealing the presence of speculative decoding in the inference pipeline.

To achieve this, we leverage the observation that in most deployments, the draft model used for speculation has a shorter context length than the main model. This design choice is intuitive: smaller models are optimized for speed and memory efficiency, and extending their context window significantly increases both computational cost and memory footprint. We confirm this pattern using publicly available information on speculative decoding deployments including examples from [72].

If a prompt requires a context longer than the draft model’s maximum context length, and the system employs speculative decoding, the draft model inevitably produces incorrect predictions. The speculative tokens are then rejected, and the token-generation latency falls back to that of the large model. In contrast, when the prompt fits within the draft model’s context window and does not demand high model capacity, the draft model’s predictions are verified by the target model, resulting in faster generation.

By gradually increasing the required context length across prompts, we can observe where a sudden increase in per-token latency occurs. The appearance of such a latency spike indicates that the provider is using speculative decoding. Furthermore, the position of this spike reveals the approximate context length of the underlying draft model.

4.2 Crafting the Prompt

We aim to craft a set of prompts that elicit responses with distinctive timing signatures capable of differentiating models that employ speculative decoding from those that do not use any inference-time optimization, as well as from models that implement alternative optimizations discussed in §2.4.

If we observe input-dependent variability in token-generation timing, it suggests the presence of some form of inference-time optimization. The more challenging task, however, is to craft prompts that specifically expose timing patterns unique to speculative decoding, rather than those caused by other inference-time optimizations. A defining characteristic of speculative decoding is that it relies on a smaller draft model, which typically has a shorter context length than the main model, as discussed above. Therefore, when a response requires a context longer than the draft model’s context length, we expect a noticeable spike in generation time. To isolate this effect and ensure that timing variability is indeed caused by context length, we design prompt sets where context length is the only variable, and all prompts require comparable (if not equal) semantic complexity from the model.

To that end, we select a simple task of remembering a large numeric value presented in the early part of the prompt. Concretely, we use the following template:

prologue_length

We have a $\{n\}$ digit number $NUM=\{rand_num\}_n.\{var_pad_str\}_x$
The value of number NUM at the start was equal to

Here $\{s\}_n$ denotes a string s with n characters. String $\{rand_num\}_n$ is a uniformly sampled n -digit integer expressed with ASCII digits, and $\{var_pad_str\}_x$ is a sequence of x randomly sampled alphabetical characters used to reach a desired prompt length. Additionally, *prologue_length* is the length of the beginning part of the prompt before the generated random number.

We avoid fixed numbers or static padding to reduce the chance that other server-side optimizations (e.g., caching) influence timings. By gradually increasing the padding size x , we sample prompt lengths up to the reported maximum context length of the victim API. As we increase x , the context length required to generate a correct response increases. For each attack iteration, we generate a fresh set of prompts of varying lengths from this template. We randomize the order in which prompts of different lengths are sent to the victim API to rule out the effects of potential order-dependent or prompt-length-dependent throttling policies on the observed timing behavior.

4.3 Detection and Parameter Estimation

We perform a two-stage inference procedure to identify and characterize the use of speculative decoding in the victim LLM:

Detection. For each sample prompt, we plot the per-token generation timing of the victim LLM. A detectable timing spike after a certain input length indicates data-dependent behavior that depends solely on prompt length rather than semantic complexity. Such a pattern suggests that the victim employs speculative decoding. Beyond a certain input length, the smaller draft model can no longer generate accurate speculations, causing the generation to be predominantly handled by the large target model, with additional

overhead from the execution of the draft model. If no timing spike is observed, the presence of speculative decoding cannot be ruled out, as it remains possible, though less likely (§4.1), that the draft model operates with the same context length as the main model. Consequently, this detection method may yield false negatives.

Parameter Estimation. Upon detecting speculative decoding, we can further infer the context length of the draft model. Specifically, we identify the critical prompt length at which the timing deviation occurs by performing a binary search within the region of the timing spike. The middle point of each search iteration is tested until the exact breakpoint is located (T_{break}). Using this value, we estimate the context window length of the smaller draft model as ($T_{break} - \text{prologue_length}$), which is the difference between the breakpoint and the number of tokens in the prompt before the first token required to produce the correct answer (i.e., the first character of $\{\text{rand_num}\}_n$).

4.4 Experimental Setup

We conduct two sets of experiments: one using a local implementation of speculative decoding to validate our detection approach, and another using black-box access to unknown production models to demonstrate its applicability in the wild. Below, we describe the experimental setup for each case.

4.4.1 Local Experiments. To characterize the timing behavior of speculative decoding, we use a publicly available third-party implementation [21] of the speculative decoding technique originally proposed by Leviathan et al. [38].

We use Guanaco 13B (a fine-tuned LoRA layer for LLaMA2 13B) [59] as the main target model and TinyLLaMA 1.1B [60] as the draft model. All experiments are performed on a cloud-based system equipped with an NVIDIA L40 GPU with 45 GB of memory, running PyTorch 2.8 and Hugging Face Transformers 4.56 on Ubuntu 22.04. We instrument the inference code with Python’s time library to record timestamps immediately before and after each generation round, enabling the computation of per-token generation timings.

4.4.2 Remote Experiments. We repeat the same attack against several major publicly available LLM APIs, including Google Gemini, OpenAI GPT, Cohere, and Mistral models. Because the attack requires varying prompt lengths up to provider context limits (which can reach up to 1M tokens), we adjust the number of repetitions in the remote experiments to limit the cost. We use the streamed chat-completion interface for all experiments. The client runs on an Ubuntu 22.04 host. To capture per-token timing, we record a timestamp at the arrival of every stream event and log the associated token count without additional on-the-fly processing. An overview of the experiment specifications is presented in Table 1. A detailed analysis of the effect of the network on our measurements can be found in appendix §A.1.

4.5 Results

4.5.1 Local White-Box Model Experiment. We first consider using a locally deployed model that employs speculative decoding to validate our detection technique.

We generate prompts from our prompt template above with prompt length from 82 to 4K and we observe per-token generation

Table 1: Specifications of the inference optimization extraction experiments.

Parameter	LLaMA13B	Gemini	Cohere
Context Length	4K	1M	128K
# Trials	100	30	65
Open Model?	✓	✗	✓
Local?	✓	✗	✗
\$ / 1M Tokens	0	0.3	2.5

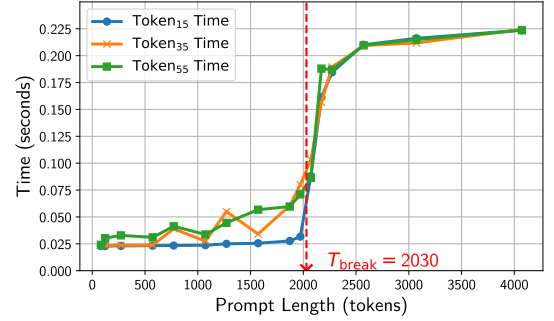


Figure 2: Timing pattern for a local implementation of speculative decoding. After a specific prompt length, the small draft model cannot produce useful predictions due to its smaller context length, causing a noticeable spike in timing.

times. If a generation event produces multiple tokens simultaneously, we divide the total per-event generation time by the number of tokens to approximate the per-token generation time.

Figure 2 presents the results of this experiment. We observe a sharp rise in per-token generation time at a specific input length, consistent with the expected behavior of speculative decoding. In a separate experiment, we observe that disabling speculative decoding eliminates this spike, confirming that the effect originates from the speculative decoding mechanism.

We next evaluate whether the context length of the draft model, TinyLLaMA 1.1B, can be inferred solely from per-token timing measurements. We perform a binary search over input prompt lengths to identify the breakpoint where token generation timing shifts from optimal efficiency to the worst-case scenario. Using the same prompt template, we generate prompts spanning the two ends of the timing jump observed in Figure 2.

After narrowing the search, we observe that the timing jump occurs approximately at $T_{break} = 2030$ tokens. Applying our context estimation formula to this value yields an estimated context length of 2018 tokens for the draft model. Given that model context lengths are typically of powers of two, we refine our estimate to the nearest power-of-two value, 2048 tokens, as the effective context window of the draft model. This matches the context length of TinyLLaMA 1.1B, validating the accuracy of our parameter retrieval method.

4.5.2 Remote Black-Box Model Experiments. We next evaluate our attack on popular publicly available LLM APIs (listed in §A.2) in a black-box setting.

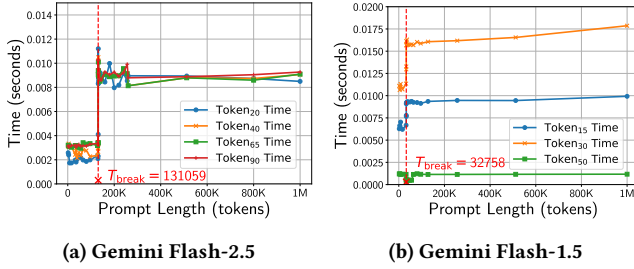


Figure 3: Per-token generation time of Gemini models. These models exhibit speculative decoding behavior. Flash-2.5-Lite also shows a similar timing jump as depicted in §A.2.

Table 2: Retrieved context window lengths of the draft models in speculative decoding-enabled LLMs.

LLM Model	Draft Context Length
Guanaco 13B + TinyLlama 1.1B	2,018 → 2K
Gemini Flash 2.5	131,042 → 128K
Gemini Flash 2.5 Lite	131,042 → 128K
Gemini Flash 1.5	32,743 → 32K

Detection Results. We repeat the same tests from the local experiments on the chat-completion APIs of the target providers. After extracting and processing the per-token generation timings, we plot the results for each remote LLM using the same methodology described earlier. Several models exhibit the timing spike associated with speculative decoding, while others show no such evidence.

Figure 3 presents the per-token generation time for different input prompt lengths for Gemini Flash-2.5 and Gemini Flash-1.5. Both exhibit a distinct spike in the timing curve at a specific prompt length, an indicator of speculative decoding. The Flash-2.5 and Flash-2.5-Lite models are among the most recent and widely used models in Google AI Studio.

In contrast, for several other APIs, we do not observe any such timing pattern. While this absence of detectable signatures prevents us from confirming the use of speculative decoding, it does not preclude its presence as our approach can produce false negatives, as described earlier.

Parameter Estimation. For the Gemini models in which we detect the use of speculative decoding, we perform a binary search similar to that described in the local experiment to estimate the draft model’s context window length. The extracted values are summarized in Table 2.

5 Leaking Model Architecture

The architecture and parameter configurations of leading LLMs are considered highly confidential, both for competitive and strategic reasons. We show that timing leakage from publicly accessible models can serve as a powerful source of information about these systems. To demonstrate this, we present an attack capable of recovering key architectural details, such as the hidden dimension size and number of decoder layers, among others. To achieve this, we build an accurate timing model of the LLM inference pipeline

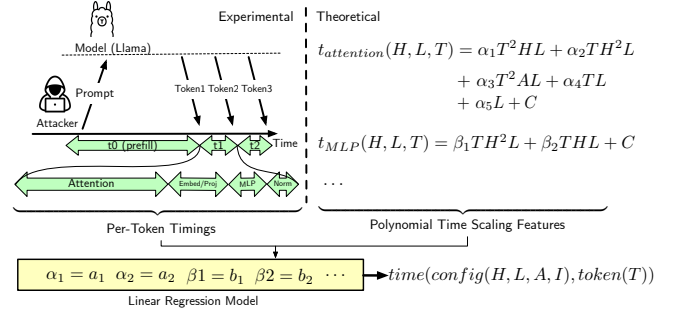


Figure 4: Overview of the process for constructing the LLM runtime predictor model. Empirical data is used to learn coefficients of asymptotic terms derived analytically.

that captures how generation latency varies with different model properties, and we use this model to reverse-engineer unknown architectures.

This section describes the attack, the construction of the timing model, and how it enables inferring parameters of black-box LLMs.

5.1 Attack Overview

A key component of this attack is constructing a model that captures the relationship between per-token generation time and the model parameters we aim to recover, namely, the number of attention heads (A), hidden dimension size (H), number of decoder layers (L), and intermediate size (I). However, accurately modeling computations as complex as transformer inference on modern GPUs is highly challenging. To address this, we adopt a bottom-up framework that begins with theoretical asymptotic analysis and progressively incorporates empirical corrections to capture hardware- and implementation-specific effects.

We leverage the modular structure of the transformer architecture by decomposing its runtime into primitive operations and incrementally increasing modeling complexity by composing these primitives into higher-level components. We perform asymptotic analysis on each primitive to characterize how its runtime scales with respect to input parameters, expressed in terms of computational and memory costs. Next, we introduce multiplicative coefficients and constant terms to capture additional factors influencing runtime. These coefficients are learned from empirical measurements, allowing the model to account for hardware- and implementation-specific effects.

We instrument real implementations of transformer components, measure their runtimes across a range of input sizes and architectural dimensions, and fit regression models using the asymptotic cost terms as features. The learned coefficients capture hardware- and implementation-specific effects. Once calibrated, this hybrid analytical-empirical model can estimate per-token runtime for new architectures and input sequences with good accuracy. The overall workflow is shown in Figure 4.

Although many approaches could be used to predict runtime, including large neural networks, we deliberately adopt simple linear regression. Linear models provide three key advantages: they are explainable, since their terms directly correspond to theoretical

cost components; generalizable, as they are less prone to overfitting across diverse architectures; and configurable, allowing easy adjustment or extension of runtime terms.

We then use the predictor as an oracle to infer architecture for a black-box model. Given a target, we control input prompts and collect per-token timing traces. We then construct a search space of plausible LLM configurations (e.g., H, A, L, I). To avoid explosive search costs, we prune the space using practical observations (e.g., typical alignment multiples such as 64 or 128). With a feasible search grid, we enumerate plausible configurations, synthesize timing traces for each candidate using the predictor, and rank candidates by distance to the observed trace. The top matches are the most likely architectural configurations for the target model.

Different implementations of the transformer architecture exhibit different asymptotic timing behavior. These implementations may rely on different GPU kernels (e.g., FlashAttention vs. cuBLAS kernels) or incorporate optimizations such as KV-cache to avoid re-computation. We begin with a baseline eager attention implementation [63, 67] that decomposes attention into matrix multiplications executed via cuBLAS. We then extend the analysis to more advanced implementations, such as FlashAttention, and finally model the impact of KV-cache optimization.

This approach assumes that once characterized, an implementation’s asymptotic behavior and the associated coefficients remain invariant across input sizes, allowing runtimes to be extrapolated to unseen dimensions. While this assumption generally holds at the level of individual kernels, CUDA libraries such as cuBLAS (used only in eager implementation) dynamically select from a large set of specialized kernels based on operand shapes and internal heuristics. This leads to small but non-negligible deviations from simple scaling terms. To address this, we model kernel selection and per-kernel runtimes in cuBLAS for matrix multiplications and apply online corrections for affected operations and components, improving prediction accuracy for unseen configurations.

This attack shows that timing traces carry an architecture-dependent fingerprint that is not easily reproduced by different architectures. In other words, if the per-token runtime is viewed as a quadratic function of input length, the coefficients of that function are themselves determined by a polynomial combination of the model’s architectural parameters. Different architectures, therefore, yield different coefficient sets. As a result, with sufficiently precise timing observations across a diverse set of inputs, an adversary can reliably infer an approximate internal architecture of a black-box LLM. Moreover, incorporating results from other attacks that leak specific architectural parameters, such as the hidden dimension size [11], can further reduce the size of our search space and improve the accuracy of the remaining inferred parameters.

5.2 Offline Phase

We first construct a per-token runtime predictor by instrumenting and profiling a set of open-access LLM implementations. The collected measurements are analyzed to build an analytical-empirical model that maps architectural parameters and input dimensions to per-token runtime estimates. In addition, we develop a matmul runtime corrector model to address the non-linearities that arise

when handling unseen architectural dimensions during the online phase.

Building Polynomial Time-Scaling Features. We decompose each decoder block from a decoder-only LLM (shown in Figure 1) with L decoder layers into its canonical subcomponents (multi-head attention, MLP, normalization, etc.) and enumerate the primitive algebraic operations performed by each subcomponent (matrix multiplications, softmax, element-wise additions, scalings, ...). For each primitive operation, we derive its theoretical compute and memory cost as a function of the relevant dimensions, including the sequence length T , hidden size H , number of attention heads A , MLP intermediate projection dimension I , and bit-width of each element in bytes b (e.g., 2 for fp16 or 0.5 for int4).

As a concrete example, consider a single decoder block with a naive Q-projection that multiplies a token embedding matrix of size $T \times H$ by a weight matrix W_Q of size $H \times H$. A naive multiplication algorithm requires $O(TH^2)$ operations and $O(bTH + bH^2)$ memory reads.

To convert these theoretical costs to wall-clock time, we use the platform characteristics: the maximum rate by which we can execute arithmetic operations, i.e., peak arithmetic throughput d (ops/sec), and the rate by which we can fetch data from memory, i.e., memory bandwidth c (bytes/sec). The baseline runtime contribution of a cost term is therefore the cost divided by the corresponding rate (e.g., $O(TH^2)/d$ and $O(bTH + bH^2)/c$ for the Q-projection example).

While different implementations may, in theory, exhibit varying asymptotic behavior, we find that in practice these naive expressions provide good approximations. Hardware and software optimizations such as tiling, vectorization, and kernel launch overhead primarily affect the empirical coefficients of these terms rather than their asymptotic dependence on our parameters of interest ($T, H, \dots$). We therefore express the runtime of a subcomponent as a linear combination of theoretical terms with empirical coefficients. For example, we can express the timing of our Q-projection example as

$$time_{Q-proj} \approx \alpha \frac{TH^2}{d} + \beta \frac{bTH + bH^2}{c} + C \quad (1)$$

where α , β , and C are fitted coefficients that capture implementation and hardware overheads. In some implementations, memory costs can be completely hidden by computation when the two operations overlap. Although our model adds these terms linearly, this behavior is captured by the empirically learned coefficients. For example, the memory coefficient becomes effectively zero when memory latency is fully masked by computation.

Following the same process used for the single Q-projection matrix multiplication, we extend the analysis to a multi-head attention block with A heads and per-head dimension $D = H/A$. We derive the theoretical scaling features for all major operations in this subcomponent. A complete list of the derived operations for each subcomponent is provided in the appendix §B.2.

Next, we move one level higher and compose the overall attention block scaling terms by combining the terms from operations of its subcomponents. To generalize the runtime of the attention block across the entire model, we multiply these scaling terms by the

number of decoder layers L , reflecting their repetition throughout the LLM.

$$time_{attention} \sim T^2HL + TH^2L + T^2AL + \dots \quad (2)$$

Following this example for the attention block, we perform a similar detailed analysis for all other subcomponents of the Transformer and extract their corresponding runtime scaling features (appendix §B.3).

Collecting per-Component per-Token Timings. To learn the empirical coefficients, we first collect detailed timing information from the actual implementation of the operations and subcomponents. We instrument an open-access model inference framework with timing utilities and record the runtime behavior of a reference model (e.g., Meta’s Llama) for further analysis. We use a broad range of input prompt lengths (T) to probe the reference model and collect per-token timing data. These records serve as our reference dataset to estimate the runtime contribution of different terms and subcomponents.

To enrich the dataset, we generate multiple architectural variants of the same reference model by altering the number of layers (L) or the hidden dimension (H). This effectively produces many instances within the same model family, allowing us to capture diverse timing fingerprints without requiring access to multiple distinct models.

We construct a dataset that records the architectural properties of the dissected LLMs, input prompt lengths, and measured per-component per-token timings for each configuration, which serve as ground-truth labels. This dataset is then used in the next step of the offline phase to determine the appropriate coefficients for each term in the final runtime model of every component.

Composing the Predictor Model. Since the operations within a subcomponent follow a deterministic and predefined sequence, and the input length and architectural dimensions only affect the runtime of these operations rather than their execution flow, we can model the total runtime of a subcomponent as a simple linear combination of its operations.

We build a separate linear regression model for each subcomponent, which, given the input length and architectural dimensions of a target LLM, predicts the runtime of that specific component. Using separate regressors for individual subcomponents, in line with our bottom-up modeling approach, helps us verify the correctness of each part before aggregating them into the overall predictor. This modular verification process avoids the complexity of validating a large monolithic model and simplifies the integration of later corrections for specific components if needed. The total per-token generation time of the LLM is then obtained by summing the predicted runtimes of all components, scaled by the number of their instances within each transformer layer.

This ensemble of linear models already captures the dominant runtime scaling behavior of the architecture. Since we build the model component-wise and do not fit it directly to the runtime of the entire LLM, each subcomponent only captures the overheads within its own scope. To account for remaining latency sources outside the canonical components, such as kernel launch delays, synchronization costs, and constant per-iteration overheads, we include an additional linear regression component to model the residual overhead, ensuring comprehensive coverage of all timing

contributions. We intentionally avoid using non-linear models such as radial basis function (RBF) regressors or neural networks, as they tend to overfit to residual noise and give a misleading impression of higher precision. Ultimately, we obtain linear equations that accurately approximate the runtime of each subcomponent. For example, for the attention block, we have

$$t_{att} = a_1H^2TLd + a_2T^2HLd + a_3bT^2ALc + \dots + C \quad (3)$$

where $a_1, \dots, a_4$ and C are the coefficients learned by the linear regression model on the runtime dataset. And for the entire model, we approximate the runtime of the entire model as

$$t_{llm} = t_{att} + t_{mlp} + 2 \cdot t_{norm} + t_{embed_proj} + t_{overhead} \quad (4)$$

where t_{att} , t_{mlp} , t_{norm} , t_{embed_proj} , and $t_{overhead}$ denote the runtime of each of the components.

Training the Runtime Corrector. To address the non-linearities in our runtime predictions caused by dynamic kernel selection in the underlying libraries, we prepare two auxiliary components: one that predicts which kernel the NVIDIA cuBLAS library [43] uses for each matmul size (i.e., pair of operand dimensions), and another that predicts the corresponding matmul runtime given its size and the chosen kernel.

To train these components, we generate a comprehensive set of matmul operations representative of those invoked by an LLM across a wide range of configurations and profile both their runtimes and the executed kernel names. Using this dataset, we construct a two-phase ensemble model consisting of (1) a LightGBM classifier [30] that predicts the kernel selected for a given pair of operand shapes, and (2) a collection of small random forest regressors, each modeling the runtime behavior of a specific kernel based on empirical timing data. This two-component model later serves as part of our runtime correction logic, enabling our predictor to generalize effectively to all unseen configurations.

Modeling FlashAttention. To demonstrate that our prediction framework generalizes to alternative Transformer implementations with minimal changes, we construct a new runtime predictor based on the FlashAttention2 kernel. We modify the runtime scaling terms associated with the attention subcomponent to reflect the theoretical behavior of FlashAttention2 [17], and train a new predictor, Flash2, using the same methodology. The primary modification arises from the reduced memory access complexity of FlashAttention, which achieves an average-case of $\Theta(T^2D^2M^{-1})$ memory accesses, where T , D , and M denote the sequence length, attention head dimension, and per-SM SRAM size, respectively. To capture this behavior, we introduce an additional memory-related scaling term into our attention model. These adjustments are derived from the algorithmic specification of FlashAttention rather than any specific implementation, and are listed in §B.5. The runtime scaling terms for the remaining subcomponents (e.g., MLP) remain unchanged from the Eager predictor and are retrained using the new timing data. Finally, since FlashAttention2 does not rely on cuBLAS-based general matrix multiplication kernels, the matmul correction logic is not required for this predictor.

Modeling KV-Cache. To further extend our framework to realistic autoregressive inference settings, we incorporate KV caching as

a representative inference optimization into our runtime modeling. We construct a KV-cache-enabled predictor, KV-Flash2, by extending the Flash2 model with separate runtime scaling terms for the prefill and decoding phases. The prefill stage follows the same scaling terms as the Flash2 predictor, while the decoding stage terms (listed in §B.6) reflect single-token processing with attention computed over the cached sequence. This separation captures the change in computational complexity induced by KV reuse after the initial forward pass. The prefill regressors are trained using only the timings from the first forward pass of each prompt in the dataset, while the decoding regressors are trained on the timing data from the remaining passes. The final model is therefore a hybrid predictor that combines prefill and decoding formulations, enabling accurate modeling of long-context autoregressive inference workloads.

5.3 Online Phase

Building the Search Space. After the LLM runtime predictor is calibrated, we use it as an oracle. Given an architectural configuration and an input length, it returns a per-token runtime prediction for the output sequence. We then employ this oracle to search across the possible architectural configuration space, identifying configurations whose synthesized timing traces best match the observed timing sequence of a black-box target. To make this search tractable, we exploit several key practical observations about modern LLM design.

First, architectural dimensions are integer values, and this immediately discretizes the search space. Second, valid configurations follow implementation conventions and alignment constraints rather than arbitrary integer values. For example, in contemporary models, the number of decoder layers is typically an even integer, hidden dimensions are usually multiples of a base stride (e.g., 64 or 128), and the number of attention heads is chosen such that the per-head dimension divides the hidden size evenly. In addition, some combinations of parameters are inherently implausible, such as configurations with extremely small hidden sizes and very deep layer counts, or the reverse, which can be safely pruned to further reduce the density of plausible configurations.

Leveraging these constraints, we construct a multi-dimensional, discrete search grid over the unknown architectural parameters (e.g., L, H, A, I).

Reverse Engineering the Target Architecture. With the search grid prepared and the predictor model serving as our oracle, we evaluate each candidate configuration using the same set of prompts that were employed to collect the timing traces from the target model. For every configuration–prompt pair, we query the predictor to generate the corresponding predicted timing sequence.

Next, we compute the distance between each predicted sequence and the observed target timings using a simple metric such as the root mean square error (RMSE). We then rank all candidate configurations according to this distance and select the top-ranked ones as potential matches for the target model’s architectural parameters.

Given that timing measurements are inherently noisy and that distinct configurations may occasionally yield similar timing profiles when probed with a limited set of prompts, we avoid over-reliance on the single closest match. Instead, we produce a short

list of the nearest candidates for subsequent inspection, allowing for a small margin of error in the architectural inference.

Scaling Runtime for Unseen Dimensions. When using the Eager predictor to estimate the runtime of configurations not observed during training, we incorporate the two corrector models trained in the offline phase. We first use the lightGBM classifier to predict the cuBLAS kernel that will be selected for each matrix multiplication in the target configuration. We then estimate the runtime of each matmul using the corresponding kernel-specific random forest regressor. These predictions are used to adjust the final runtime estimate of the Eager predictor for unseen architectural configurations. We refer to this enhanced model as Eager(Corrected). A detailed step-by-step description of this correction procedure is presented in Appendix §B.1.

Ranking Strategy for Reverse Engineering with KV-Flash2.

Since we rely on the first output token to characterize the prefill phase, and the timing differences across subsequent decoding steps are minimal, we use the average latency of the remaining output tokens as a robust estimate of per-token decoding time for each prompt length. As a result, we obtain fewer effective data points compared to the earlier experiments, where each output token provided a strong timing signal. To address this, we leverage the hybrid design of the KV-Flash2 predictor in a two-stage ranking procedure. We first narrow the search space to the top-35 candidate configurations using only the prefill component, and then refine the ranking within this subset by incorporating both prefill and decoding predictions. This staged approach improves the ranking of the true configuration and preserves the effectiveness of the attack under KV-cache-enabled inference.

5.4 Experimental Setup

We use two models from the Llama 3.2 family with different sizes: the 1B version serves as our reference model for the offline phase to generate timing data for training the linear regression predictors, while the 3B version is used as the target model to generate test datasets for evaluating runtime prediction accuracy and the architectural search and reverse-engineering procedure. Unless stated otherwise, training (reference) datasets are generated from Llama 3.2 1B, and test (target) datasets are generated from Llama 3.2 3B.

The search grid for the reverse engineering task enumerates candidate values for each target architectural parameter, including but not limited to those observed in the datasets. The grid is strictly inclusive and is constructed to be at least one order of magnitude larger than the number of unique configurations in each dataset. The online search evaluates 1,540 candidate configurations for each of the 130 target configurations, and takes approximately 20 minutes when executed in parallel across 40 CPU cores. We report top-5 accuracy, defined as the fraction of target configurations for which the correct configuration appears within the top five candidates returned by the attack. A retrieval is considered successful if all targeted architectural parameters of a candidate lie within one step of the ground-truth configuration. The step size is parameter-specific, which is 1 for L , 128 for H , 4 for A , and 1024 for I .

We implement a minimal autoregressive inference loop using the standardized Meta Llama implementation provided by the HuggingFace Transformers library [67]. To ensure accurate per-token timing during data collection in both offline and online phases, we insert exactly two CUDA synchronization instructions, one before starting the timer and one before stopping it, thereby isolating the generation time of each token without interference from preceding or subsequent token generations. This synchronization overhead is minimal and does not disrupt GPU execution during single-token inference. We use Python’s standard time library to log timestamps at the necessary points in the inference loop. For per-subcomponent timing measurements in the offline phase, we introduce conditional checks on control parameters that allow enabling or bypassing individual transformer sub-components. This modification does not affect the timing behavior of the model as a whole but effectively removes the contribution of a specific sub-component and the overhead imposed by it from the overall runtime. We run the runtime prediction and architectural classification evaluations on a 40-core Intel(R) Xeon(R) Silver 4416 CPU.

The following are the specific setups used for different experiments in this section:

Eager Attention Experiments. In addition to the test dataset generated from the default Llama 3.2 3B target model, we generate an additional test dataset using the 1B model to evaluate in-domain generalization of the predictor model. When generating test data from the 1B model, we restrict the configurations to those not used in the training dataset. The offline collection of training and testing runtime data yields measurements for 81 unique (H, L) configurations in the training set, 110 configurations in the 1B-model test set, and 130 configurations in the 3B-model test set.

Both the reference and target models are executed on a single NVIDIA GeForce RTX 2080 Ti GPU (11 GB VRAM) with CUDA 12.4 and cuBLAS enabled, minimizing CPU–GPU transfer and PCIe overheads. We use 24 identical GPUs distributed across three servers, each running the same model and experiment on a distinct data partition. To ensure that no attention optimization is applied during these experiments, we explicitly force the model to use the eager attention implementation, with KV-cache disabled.

Open Model Experiments. To evaluate cross-family generalization, we generate new test datasets using three different open models: the 1.5B model from the Qwen2.5 family [58], the Phi3.5-mini 3.8B model [1], and the 2B model from the Gemma2 family [57]. The resulting test datasets consist of 44, 40, and 40 distinct (H, L) configurations, respectively. We run these experiments on the same RTX 2080 Ti setup as the eager attention experiments.

FlashAttention Experiments. For experiments involving FlashAttention2, which requires Ampere or newer GPUs, we use an NVIDIA A10 GPU (24 GB VRAM) with CUDA 12.6 to generate training and test datasets consisting of 81 and 130 configurations, respectively. During inference, we force the transformers library to use PyTorch’s built-in scaled dot-product attention (SDPA) FlashAttention2 [17] implementation.

KV+FlashAttention Experiments. For experiments with KV-cache-enabled inference, we use the same inference setup as in the FlashAttention experiments, with KV caching enabled. These experiments are run on an NVIDIA B200 GPU (180 GB HBM3E

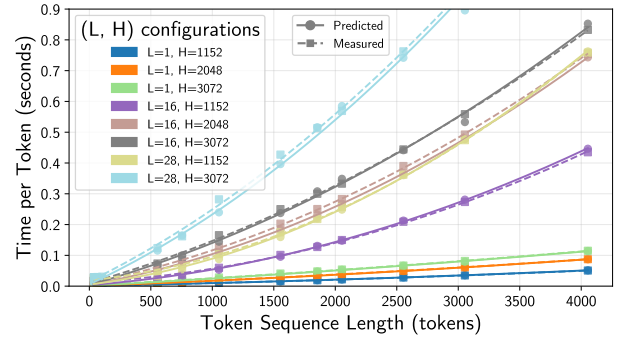


Figure 5: Predicted token-generation times vs. ground-truth times. Our model can accurately predict the per-token generation time for various configurations.

Table 3: Accuracy of the runtime predictor model, measured as normalized root mean squared error (NRMSE).

Step	Eager (Naive)		Eager (Corrected)		Flash2		KV-Flash2	
	Train	Test	Train	Test	Train	Test	Train	Test
Decode	0.106	0.411	0.106	0.119	0.258	0.209	0.218	0.252
Prefill	-	-	-	-	-	-	0.168	0.188

VRAM) from the Blackwell series, with CUDA 13.0, using extended prompt lengths for both the Llama 1B and 3B models. We construct the training and test datasets following the same methodology, consisting of 159 and 148 configurations, respectively.

Remote Inference API Experiment. To evaluate the accuracy of our runtime prediction framework and the feasibility of our architectural leakage attack in real-world deployments, we collect per-token timing traces from the Weights & Biases Llama 3.1 8B Instruct streaming API [66]. In our evaluations, we treat the API as a fully unknown black-box model without access to its internal architectural specifications. For reporting the results, we subsequently use the documented model configuration to validate the inferred parameters and runtime prediction accuracy. Although the target model belongs to the same family used for training, its architectural dimensions lie outside the range of parameters seen by the predictor during training. In this experiment, the target model is hosted on a remote machine to which we do not have direct access and is served using an unknown, unmodified inference framework. Unlike our local experiments, we do not instrument the serving stack for timing collection. The per-token timing collection client follows the same setup described in §4.4.2.

We select this API from a range of providers to minimize the impact of unmodeled optimizations and to ensure that the model is served in its original FP16 precision.

5.5 Results

Accuracy of Token Generation Time Prediction. We first assess the ability of our models in predicting the token generation time of models with different configurations.

Table 4: Top-5 accuracy (%) of parameter leakage attack.

Target Parameter	Eager (Corrected)		Flash2		KV-Flash2	
	Train	Test	Train	Test	Train	Test
L	95.06	86.15	80.56	97.69	100.00	83.78
H	100.00	71.54	99.07	100.00	89.94	70.95
(H, L)	91.36	65.38	73.15	54.62	72.96	45.27

Figure 5 illustrates the predicted runtimes along with the corresponding ground-truth measurements for several test configurations using eager attention. The results highlight the quadratic relationship between the runtime and the sequence length, and show that our simple regression-based model accurately captures this behavior. Even for seemingly similar timing patterns, such as (L=28, H=1152) and (L=16, H=2048), we observe distinct scaling behavior. The former configuration (more layers) starts with lower generation latency but exhibits higher growth rate, eventually surpassing the latter configuration (larger hidden dimension). This further supports our claim that, given a sufficiently large range of input sequences and precise timings, it is possible to uniquely distinguish between different architectural parameters. We present similar example configurations demonstrating these patterns for Flash2 and KV-Flash2 models in Appendix §B.5 and §B.6.

Table 3 shows the normalized root mean squared error (NRMSE) of token-generating time prediction for train and test datasets for various attention implementations.

To evaluate the effectiveness of our correction for non-linearities introduced by CuBLAS kernel selection in eager attention (§5.3), we report results for both Eager (Naive) and Eager (Corrected) variants. Both variants achieve low error on the train dataset generated by Llama 1B model. However, Eager (Naive)’s performance plummets on the test dataset generated by the 3B model, while the Eager (Corrected) variant maintains its accuracy, showing the effectiveness of our non-linearity correction in generalizing to unseen architectures.

The table also shows that our Flash2 and KV-Flash2 can also predict the token generation time of unseen architectures, albeit with lower accuracy than the simple Eager predictor. This reduced accuracy stems from the highly fused and complex nature of the FlashAttention2 kernel, whose runtime behavior depends on multiple interacting factors beyond the dominant memory access term captured in our model. While we incorporate the primary scaling effect introduced by FlashAttention, additional lower-order factors and implementation-specific details are not explicitly modeled. Furthermore, discrepancies between the theoretical specification and practical implementations introduce small variations in runtime that are not fully captured by our scaling terms.

For inference with KV-cache enabled, decoding processes one token per step, resulting in timing signals that are less informative due to limited variation across sequence lengths. In contrast, the pre-fill stage, where the KV-cache is constructed, is heavily influenced by context length and architectural parameters, and thus provides a more useful signal. Therefore, for this predictor we also model prefill timings as well. The results in Table 3 show that our model can accurately capture the timing of KV-cache prefill, achieving an NRMSE of 0.18 on the test dataset.

Table 5: NRMSE across different test models.

Model	Llama3.2 1B	Llama3.2 3B	Qwen2.5 1.5B	Phi3.5-mini 3.8B	Gemma2 2B
NRMSE	0.221	0.119	0.159	0.183	0.186

Table 6: Top-5 accuracy (%) across different test models.

Target	Llama3.2 1B	Llama3.2 3B	Qwen2.5 1.5B	Phi3.5-mini 3.8B	Gemma2 2B
L	100.00	86.15	77.27	97.50	100.00
H	93.64	71.54	100.00	97.50	77.50
(H, L)	90.91	65.38	50.00	80.00	47.50

Accuracy of Model Parameter Leakage Attack. We next evaluate how our models can be used to infer the architectural dimensions based on the observed timing traces. As discussed in §5.3, we use our runtime predictor model to build a classifier that ranks the candidate architectural configurations for a given observed timing trace, based on their similarity to the trace generated by the predictor.

We classify each timing trace in the datasets to one of the classes in a large configuration space (discussed in §5.4), and report the top-5 accuracy of this classification. We repeat this for different set of target parameters that we aim to leak. Table 4 presents the results of these experiments.

On the unseen test datasets, the attack can achieve a high top-5 accuracy (> 70%) across all predictors when inferring only one unknown architectural parameter. When both H and L are unknown, the task becomes significantly more challenging, as the label space is much larger. In addition, most asymptotic terms in our runtime model depend on the product $H \cdot L$, making it difficult for the classifier to separate their individual impact. The lower accuracy of our runtime predictor for Flash2 and KV-Flash2, compared to the Eager variant as discussed above, directly impacts the classification accuracy. Therefore, the classification accuracy of these models is generally lower than the Eager predictor.

Cross-Family Generalization. We next evaluate whether a predictor trained on one model family generalizes to decoder-only architectures from different model families. We reuse the Eager (Corrected) predictor trained on timing data from the Llama 1B model and evaluate its performance on test datasets generated from our three representative open models. Table 5 shows the runtime prediction accuracy for these models, and Table 6 shows the top-5 success rate of the architectural configuration leakage attack. The high success rate for different models indicates that our linear regression-based prediction framework does not overfit to a specific model family and generalizes well across architectures. Additional details on prediction accuracy for the open models are presented in Appendix §B.4.

Leaking Architectural Parameters from a Remote Inference API. Next, we evaluate our attack in leaking architectural parameters of an unknown black-box model accessible only through a remote inference service API. First, we use our KV-Flash2 predictor to estimate the expected timing behavior of a model with the same architecture as our remote Llama3.1 8B target (on Weights&Biases). As shown in Figure 6, the predicted timings closely match the observed measurements from the remote API generation times. Next,

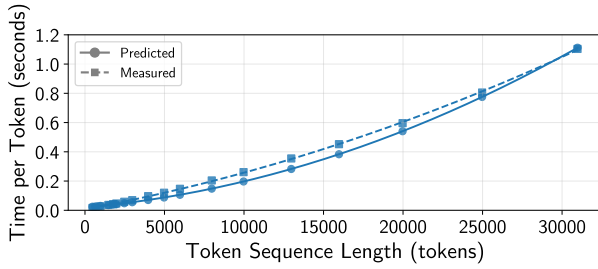


Figure 6: Predicted token-generation times vs. ground-truth prefill stage times for W&B’s Llama 3.1 8B Instruct remote API.

we perform the architectural parameter recovery experiment over an extended $[H, L]$ search grid to avoid bias toward larger configurations. When assuming L is known and inferring H , the correct value is recovered as the top-ranked candidate. When fixing L and inferring H , the attack ranks the correct configuration within the top-5 (ranked 4th). When searching over both dimensions, a near match ($H = 4096, L = 34$) is ranked 8/2068, and the true configuration ($H = 4096, L = 32$) is ranked 28/2068. These results demonstrate that our approach can leak architectural parameters in realistic API settings.

6 Mitigation

Mitigating timing leakage without compromising QoS is challenging. Below, we discuss potential mitigation strategies.

Constant Per-Token Time. Our attacks exploit per-token timing information that depends on both model and deployment characteristics. In particular, for the model leakage attack, due to the autoregressive nature of transformers, later tokens typically take longer to generate than earlier ones, and the rate of this increase reveals information about key model dimensions. To defend against such attacks, the ideal approach would be to eliminate the dependency of generation time on model parameters. However, achieving this in practice would require making all token generations as slow as the worst-case scenario, potentially increasing latency by several times. This mitigation strategy would therefore defeat the purpose of inference optimizations entirely.

Buffering. An alternative defense is to reduce timing granularity by buffering token outputs instead of streaming them immediately. For example, the provider could aggregate a fixed number of tokens N and transmit them at constant intervals T , thereby flattening timing variations. Yet, this approach may be impractical for large models or latency-sensitive applications, since delaying token transmission can degrade Quality of Service (QoS) and user experience.

7 Related Work

Side-Channel Attacks on Large Language Models. Prior works have demonstrated attacks that exploit various side channels, including timing [20], cache and memory access patterns [28, 71],

power consumption [23], electromagnetic emissions [6], and GPU-level side channels [65], to extract model weights, output labels, or hardware properties from classical deep neural networks.

Side-channel attacks have also emerged for modern LLMs. Recent works exploit timing channels stemming from inference-time optimizations [10], caching and memory behaviors [25, 52, 74], network-level artifacts such as token-count or packet-length side channels [40, 64, 73], and cache-based attacks [24]. These efforts primarily focus on fingerprinting models, leaking user–LLM interaction details under specific conditions, or, in some cases, recovering limited architectural or deployment information, such as through GPU performance counters [70] or long-range EM leakage [69].

There also exist model-stealing attacks that don’t rely on side channels. Carlini et al. [11] recover the hidden dimension size H and projection layer W_{proj} of production models with near-exact precision by exploiting the returned logit bias from the API. Compared to our architecture-leakage attack in §5, their attack relies on extra information returned by the API and is more costly, since all interaction must occur online.

Modeling of Large Language Models. Several previous works have developed performance-modeling tools and analytical frameworks for understanding or predicting LLM behavior, including trace-based simulation frameworks [47], analytical models for distributed training and inference [35], forecasting frameworks for model performance on unseen GPUs [37], and hardware-agnostic analytical modeling of LLM inference [46]. Accurately modeling GPUs is crucial for insightful performance models for LLMs. Accel-Sim [31] provides a detailed GPU simulation environment, while work such as [13] offers a component-wise perspective on GPU execution costs. Amali [9] provides analytical modeling for inference workloads on modern GPUs. However, most of these approaches adopt a system-centric view (e.g., focusing on memory transactions, kernel launches) and lack the theoretical parameters governing LLM scaling laws. Our work bridges this gap by combining theoretical scaling terms with empirical measurements.

8 Conclusion

This paper presents LeakyLMs, the first systematic study demonstrating that fine-grained per-token generation timings from production large language models can leak sensitive information about both model architecture and deployment strategies. This poses a significant threat, as such information provides a competitive advantage to leading organizations in the ongoing AI race. Defending against such timing side channels without degrading latency or user experience remains an open and difficult challenge. This work represents an initial step toward understanding and mitigating timing-based model leaks in large language model deployments.

Acknowledgments

The authors would like to gratefully acknowledge Longview Philanthropy for their support of our ongoing research in this area.

References

- [1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benham, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Lilian Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacrose, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL] <https://arxiv.org/abs/2404.14219>
- [2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kacmar. 2020. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 145–158. doi:10.1109/ISCA45697.2020.00023
- [3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit K. Sanghvi. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *ArXiv abs/2305.13245* (2023). <https://api.semanticscholar.org/CorpusID:258833177>
- [4] Artificial Analysis. [n. d.]. Models. <https://artificialanalysis.ai/models>. Accessed: 2025-11-11.
- [5] Baseten. 2025. Inference Platform: Deploy AI models in production. <https://www.baseten.co/>. Accessed: 2025-11-14.
- [6] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2019. {CSI} {NN}: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium (USENIX Security 19)*. 515–532.
- [7] Nikhil Bhendawade, Irina Belousova, Qichen Fu, Henry Mason, Mohammad Rastegari, and Mahyar Najibi. 2024. Speculative streaming: Fast llm inference without auxiliary models. *arXiv preprint arXiv:2402.11131* (2024).
- [8] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. MEDUSA: Simple LLM inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning (Vienna, Austria) (ICML '24)*. JMLR.org, Article 203, 27 pages.
- [9] Shiheng Cao, Junmin Wu, Junshi Chen, Hong An, and Zhibin Yu. 2025. AMALI: An Analytical Model for Accurately Modeling LLM Inference on Modern GPUs. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1495–1508. doi:10.1145/3695053.3731064
- [10] Nicholas Carlini and Milad Nasr. 2024. Remote timing attacks on efficient language model inference. *arXiv preprint arXiv:2410.17175* (2024).
- [11] Nicholas Carlini, Daniel Paleka, Krishnamurthy (Dj) Dvijotham, Thomas Steinke, Jonathan Hayase, A. Feder Cooper, Katherine Lee, Matthew Jagielski, Milad Nasr, Arthur Conmy, Eric Wallace, David Rolnick, and Florian Tramèr. 2024. Stealing part of a production language model. In *Proceedings of the 41st International Conference on Machine Learning (Vienna, Austria) (ICML '24)*. JMLR.org, Article 221, 26 pages.
- [12] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318* (2023).
- [13] Jolly Chen, Ana Lucia Varbanescu, and Monica Dessoie. 2025. Component-Based Analytical Modeling of GPU Runtime Performance: a Case-Study in Scientific Computing. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering (Toronto ON, Canada) (ICPE '25)*. Association for Computing Machinery, New York, NY, USA, 192–203. doi:10.1145/3676151.3719367
- [14] Ziyi Chen, Xiaocong Yang, Jiacheng Lin, Chenkai Sun, Kevin Chen-Chuan Chang, and Jie Huang. 2025. Cascade Speculative Drafting for Even Faster LLM Inference. arXiv:2312.11462 [cs.LG] <https://arxiv.org/abs/2312.11462>
- [15] OpenAI Corporation. [n. d.]. GPT-5 System Card. <https://cdn.openai.com/gpt-5-system-card.pdf>. Accessed: 2025-11-13.
- [16] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. arXiv:1901.02860 [cs.LG] <https://arxiv.org/abs/1901.02860>
- [17] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv:2307.08691 [cs.LG] <https://arxiv.org/abs/2307.08691>
- [18] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] <https://arxiv.org/abs/2205.14135>
- [19] Deutsche Welle. 2025. “China is going to win the AI race” — Nvidia CEO Jensen Huang decries the price of electricity in the U.S., contrasts it with China’s subsidized pricing. <https://www.dw.com/en/china-ai-artificial-intelligence-deepseek-us-chatgpt-semiconductors-graphics-technology-v2/a-74361630>. Accessed: 2025-11-14.
- [20] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E. Balas. 2019. Stealing Neural Networks via Timing Side Channels. arXiv:1812.11720 [cs.CR] <https://arxiv.org/abs/1812.11720>
- [21] feifeibear. [n. d.]. Fast inference from transformers via speculative decoding implementation. <https://github.com/feifeibear/LLMSpeculativeSampling>. Accessed: 2025-11-13.
- [22] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of LLM inference using LOOKAHEAD DECODING. In *Proceedings of the 41st International Conference on Machine Learning (Vienna, Austria) (ICML '24)*. JMLR.org, Article 561, 20 pages.
- [23] Yansong Gao, Huming Qiu, Zhi Zhang, Binghui Wang, Hua Ma, Alsharif Abuadba, Minhui Xue, Anmin Fu, and Surya Nepal. 2024. Deeptheft: Stealing dnn model architectures through power side channel. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3311–3326.
- [24] Zibo Gao, Junjie Hu, Feng Guo, Yixin Zhang, Yinglong Han, Siyuan Liu, Haiyang Li, and Zhiqiang Lv. 2025. I Know What You Said: Unveiling Hardware Cache Side-Channels in Local Large Language Model Inference. arXiv:2505.06738 [cs.CR] <https://arxiv.org/abs/2505.06738>
- [25] Chenchen Gu, Xiang Lisa Li, Rohith Kuditipudi, Percy Liang, and Tatsunori Hashimoto. 2025. Auditing Prompt Caching in Language Model APIs. arXiv:2502.07776 [cs.CL] <https://arxiv.org/abs/2502.07776>
- [26] Zhenyu He, Zexuan Zhong, Tianle Cai, Jason Lee, and Di He. 2024. REST: Retrieval-Based Speculative Decoding. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Kevin Duh, Helena Gomez, and Steven Bethard (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 1582–1595. doi:10.18653/v1/2024.naacl-long.88
- [27] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture: A Quantitative Approach* (6 ed.). Morgan Kaufmann, San Mateo, CA.
- [28] Weizhe Hua, Zhiru Zhang, and G Edward Suh. 2018. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.
- [29] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [30] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: a highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (Long Beach, California, USA) (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 3149–3157.
- [31] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-sim: an extensible simulation framework for validated GPU modeling. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (Virtual Event) (ISCA '20)*. IEEE Press, 473–486. doi:10.1109/ISCA45697.2020.00047
- [32] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. 2023. Speculative decoding with big little decoder. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (New Orleans, LA, USA) (NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1705, 21 pages.
- [33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.
- [34] Siqi Kou, Lanxiang Hu, Zhezhi He, Zhijie Deng, and Hao Zhang. 2024. CLLMs: consistency large language models. In *Proceedings of the 41st International Conference on Machine Learning (Vienna, Austria) (ICML '24)*. JMLR.org, Article 1018,

- 15 pages.
- [35] Joyjit Kundu, Wenzhe Guo, Ali BanaGozar, Udari De Alwis, Sourav Sengupta, Puneet Gupta, and Arindam Mallik. 2024. Performance Modeling and Workload Analysis of Distributed Large Language Model Training and Inference. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 57–67. doi:10.1109/IISWC63097.2024.00015
 - [36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) (SOSP '23). Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165
 - [37] Seonho Lee, Amar Phanishayee, and Divya Mahajan. 2025. Forecasting GPU Performance for Deep Learning Training and Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 493–508. doi:10.1145/3669940.3707265
 - [38] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) (ICML '23). JMLR.org, Article 795, 13 pages.
 - [39] Jiachen Liu, Jae-Won Chung, Zhiyu Wu, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. 2024. Andes: Defining and Enhancing Quality-of-Experience in LLM-Based Text Streaming Services. arXiv:2404.16283 [cs.DC] <https://arxiv.org/abs/2404.16283>
 - [40] Geoff McDonald and Jonathan Bar Or. 2025. Whisper Leak: a side-channel attack on Large Language Models. arXiv preprint arXiv:2511.03675 (2025).
 - [41] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) (ASPLOS '24). Association for Computing Machinery, New York, NY, USA, 932–949. doi:10.1145/3620666.3651335
 - [42] National Telecommunications and Information Administration (NTIA). 2024. AI System Disclosures. <https://www.ntia.gov/issues/artificial-intelligence/ai-accountability-policy-report/developing-accountability-inputs-a-deeper-view/information-flow/ai-system-disclosures>. Accessed: 2025-11-14.
 - [43] NVIDIA Corporation. 2024. cuBLAS Library User Guide / Reference Manual. <https://docs.nvidia.com/cuda/cublas/index.html> Version 12.4, Accessed: 2025-11-13.
 - [44] OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastian Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrlyov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpouras, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] <https://arxiv.org/abs/2508.10925>
 - [45] Sihyeong Park, Sungryeol Jeon, Chaelyn Lee, Seokhun Jeon, Byung-Soo Kim, and Jemin Lee. 2025. A Survey on Inference Engines for Large Language Models: Perspectives on Optimization and Efficiency. arXiv:2505.01658 [cs.CL] <https://arxiv.org/abs/2505.01658>
 - [46] Rajeev Patwari, Ashish Sirasao, and Devleena Das. 2025. Forecasting LLM Inference Performance via Hardware-Agnostic Analytical Modeling. arXiv:2508.00904 [cs.PF] <https://arxiv.org/abs/2508.00904>
 - [47] Huwan Peng, Scott Davidson, C.-J Shi, and Michael Taylor. 2025. ReaLLM: A Trace-Driven Framework for Rapid Simulation of Large-Scale LLM Inference. 85–92. doi:10.1109/ASAP65064.2025.00022
 - [48] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training. <https://api.semanticscholar.org/CorpusID:49313245>
 - [49] Alec Radford, Jeff Wu, Rewon Child, David Luu, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
 - [50] David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. 2024. Mixture-of-depths: Dynamically allocating compute in transformer-based language models. arXiv preprint arXiv:2404.02258 (2024).
 - [51] Andrea Santilli, Silvio Severino, Emilian Postolache, Valentino Maiorca, Michele Mancusi, Riccardo Marin, and Emanuele Rodolà. 2023. Accelerating Transformer Inference for Translation via Parallel Decoding. 12336–12355. doi:10.18653/v1/2023.acl-long.689
 - [52] Linke Song, Zixuan Pang, Wenhao Wang, Zihao Wang, XiaoFeng Wang, Hongbo Chen, Wei Song, Yier Jin, Dan Meng, and Rui Hou. 2025. The Early Bird Catches the Leak: Unveiling Timing Side Channels in LLM Serving Systems. *IEEE Transactions on Information Forensics and Security* 20 (2025), 11431–11446. doi:10.1109/TIFS.2025.3622954
 - [53] Benjamin Spector and Chris Re. 2023. Accelerating LLM Inference with Staged Speculative Decoding. arXiv:2308.04623 [cs.AI] <https://arxiv.org/abs/2308.04623>
 - [54] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) (NIPS '18). Curran Associates Inc., Red Hook, NY, USA, 10107–10116.
 - [55] Ziteng Sun, Ananda Theertha Suresh, Jae Hun Ro, Ahmad Beirami, Himanshu Jain, and Felix Yu. 2024. SpecTr: Fast Speculative Decoding via Optimal Transport. arXiv:2310.15141 [cs.LG] <https://arxiv.org/abs/2310.15141>
 - [56] Gemini Team. 2025. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] <https://arxiv.org/abs/2312.11805>
 - [57] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charlie Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagig, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Wely, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozńska, Dustin Erbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshov, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucińska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonnell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjoesund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, Lilly McNealus, Livio Baldini Soares, Logan Kilpatrick, Lucas Dixon, Luciano Martins, Machel Reid, Manvinder Singh, Mark Iversen, Martin Görner, Mat Velloso, Mateo Wirth, Matt Davidow, Matt Miller, Matthew Rahtz, Matthew Watson, Meg Risdal, Mehran Kazemi, Michael Moynihan, Ming Zhang, Minsuk Kahng, Minwoo Park, Mofi Rahman, Mohit Khatwani, Natalie Dao, Nenshad Bardoliwalla, Nesh Devanathan, Neta Dumai, Nilay Chauhan, Oscar Wahltinez, Pankil Botarda, Parker Barnes, Paul Barham, Paul Michel, Pengchong Jin, Petko Georgiev, Phil Culliton, Pradeep Kuppala, Ramona Comanescu, Ramona Merhej, Reena Jana, Reza Ardeshtir Rokni, Rishabh Agarwal, Ryan Mullins, Samaneh Saadat, Sara Mc Carthy, Sarah Cogan, Sarah Perrin, Sébastien M. R. Arnold, Sebastian Krause, Shengyang Dai, Shruti Garg, Shruti Sheth, Sue Rønstrom, Susan Chan, Timothy Jordan, Ting Yu, Tom Eccles, Tom Hennigan, Tomas Kocisky, Tulsee Doshi, Vihan Jain, Vikas Yadav, Vilobh Meshram, Vishal Dharmadhikari, Warren Barkley, Wei Wei, Wenming Ye, Woohyun Han, Woosuk Kwon, Xiang Xu, Zhe Shen, Zhitao Gong, Zichuan Wei, Victor Cotruta, Phoebe Kirk, Anand Rao, Minh Giang, Ludovic Peran, Tris Warkentin, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, D. Sculley, Jeanine Banks, Anca Dragan, Slav Petrov, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Sebastian Borgeaud, Noah Fiedel, Armand Joulin, Kathleen Kenealy, Robert Dadashi, and Alek Andreev. 2024. Gemma 2: Improving Open Language Models at a Practical Size. arXiv:2408.00118 [cs.CL] <https://arxiv.org/abs/2408.00118>
 - [58] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. <https://qwenlm.github.io/blog/qwen2.5/> Accessed: 2026-02-05.
 - [59] Tim Dettmers. [n. d.]. Guanaco-13B (finetuned chatbot models, 4-bit QLoRA). <https://huggingface.co/timdetters/guanaco-13b>. Accessed: 2025-11-13.
 - [60] TinyLlama. [n. d.]. TinyLlama-1.1B-Chat-v1.0. <https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0>. Accessed: 2025-11-13.
 - [61] Tom's Hardware. 2025. China is going to win the AI race — Nvidia CEO Jensen Huang decries the price of electricity in the U.S., contrasts

- it with China’s subsidized pricing. <https://www.tomshardware.com/tech-industry/artificial-intelligence/china-is-going-to-win-the-ai-race-nvidia-ceo-jensen-huang-decries-the-price-of-electricity-in-the-us-constrasts-it-with-chinas-subsidized-pricing>. Accessed: 2025-11-14.
- [62] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *ArXiv abs/2302.13971* (2023). <https://api.semanticscholar.org/CorpusID:257219404>
- [63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) (NIPS’17). Curran Associates Inc., Red Hook, NY, USA, 6000–6010.
- [64] Jiankun Wei, Abdulrahman Abdulrazzag, Tianchen Zhang, Adel Muursepp, and Gururaj Saileshwar. 2025. When Speculation Spills Secrets: Side Channels via Speculative Decoding in LLMs. *arXiv:2411.01076 [cs.CL]* <https://arxiv.org/abs/2411.01076>
- [65] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 125–137.
- [66] Weights & Biases. 2024. Meta Llama 3.1 8B Instruct. https://wandb.ai/inference/coreweave/cw_meta-llama-3.1-8B-Instruct. Weights & Biases Inference page, accessed 2026-04-27.
- [67] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. <https://www.aclweb.org/anthology/2020.emnlp-demos.6>
- [68] Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. 2023. Speculative Decoding: Exploiting Speculative Execution for Accelerating Seq2seq Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 3909–3925. doi:10.18653/v1/2023.findings-emnlp.257
- [69] Rui Xiao, Sibo Feng, Soundarya Ramesh, Jun Han, and Jinsong Han. 2026. Peering Inside the Black-Box: Long-Range and Scalable Model Architecture Snooping via GPU Electromagnetic Side-Channel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. <https://www.ndss-symposium.org/ndss-paper/peering-inside-the-black-box-long-range-and-scalable-model-architecture-snooping-via-gpu-electromagnetic-side-channel/>
- [70] Haoxuan Xu, Chen Gong, Beijie Liu, Haizhong Zheng, Beidi Chen, and Mengyuan Li. 2026. Wave: Leveraging Architecture Observation for Privacy-Preserving Model Oversight. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (USA) (ASPLOS ’26)*. Association for Computing Machinery, New York, NY, USA, 2212–2231. doi:10.1145/3779212.3790247
- [71] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2020. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2003–2020. <https://www.usenix.org/conference/usenixsecurity20/presentation/yan>
- [72] Penghui Yang, Cunxiao Du, Fengzhuo Zhang, Haonan Wang, Tianyu Pang, Chao Du, and Bo An. 2025. LongSpec: Long-Context Speculative Decoding with Efficient Drafting and Verification. doi:10.48550/arXiv.2502.17421
- [73] Tianchen Zhang, Gururaj Saileshwar, and David Lie. 2024. Time Will Tell: Timing Side Channels via Output Token Count in Large Language Models. *arXiv:2412.15431 [cs.LG]* <https://arxiv.org/abs/2412.15431>
- [74] Xinyao Zheng, Husheng Han, Shangyi Shi, Qiyan Fang, Zidong Du, Xing Hu, and Qi Guo. 2024. InputSnatch: Stealing Input in LLM Services via Timing Side-Channel Attacks. *arXiv:2411.18191 [cs.CR]* <https://arxiv.org/abs/2411.18191>

A Additional Details on Leaking Inference Optimizations Attack

A.1 Network Effect on Time Measurements

Prior to measurement, we profile the network to establish baseline latency and noise. For API access, we use each provider’s official Python package when available, otherwise we use Python’s requests library. Our measurements for the two remote models in

Table 7: Results of speculative decoding detection across different LLM models.

LLM Model	Uses Speculative Decoding?
LLaMA2(Guanaco) 13B + TinyLlama 1.1B	✓
Gemini 2.5 Pro	–
Gemini Flash 2.5	✓
Gemini Flash 2.5 Lite	✓
Gemini Flash 2.0	–
Gemini Flash 2.0 Lite	–
Gemini Flash 1.5	✓
OpenAI gpt-4o	–
OpenAI gpt-4-0613	–
OpenAI gpt-3.5-turbo	–
Mistral large-2411	–
Cohere command-r-plus-08-2024	–
Perplexity sonar	–
Cerebras qwen-3-235b-a22b-instruct-2507	–
Groq llama3-8b	–

Table 1 show that the average round-trip time (RTT) for Gemini and Cohere is 26.65 ms and 5.75 ms, respectively, with RTT standard deviations of 0.07 ms and 0.08 ms. Note that a constant RTT between our client and the APIs only produces a fixed offset in the timing traces. It does not alter the relative per-token timing pattern. The principal network effect that can confound our measurements is RTT variability. If the standard deviation of arrival times is on the same order of magnitude as the per-token generation time, the introduced noise can obscure the LLM’s true timing behavior. The measured RTT standard deviations for our tested remote models are below 5% of the minimum per-token generation time.

A.2 Remote Black-Box Model Results

Table 7 summarizes all the tested models for inference optimization and the detection outcomes.

Figure 7 depicts the successful detection of the speculative decoding signature timing jump in Gemini Flash-2.5-Lite. The gap between the two extremes of the timing jump is slightly smaller than in other successful cases. This may be due to the smaller difference in size between the main and draft models.

Figure 8 shows examples of timing responses from APIs where no speculative decoding signature was detected. As shown in Figure 8b, the Cohere Command R+ model exhibits a non-constant yet stable per-token timing pattern, likely reflecting natural variations due to KV-cache utilization rather than speculative decoding. In contrast, other models with no detected speculative behavior, such as Gemini Flash-2.0-Lite (Figure 8a), show nearly constant and consistent timing across input lengths. These observations confirm that our method depends on the consistency of timing rather than model-specific scaling laws affected by other optimizations, thereby amplifying its robustness and reducing the likelihood of false positives.

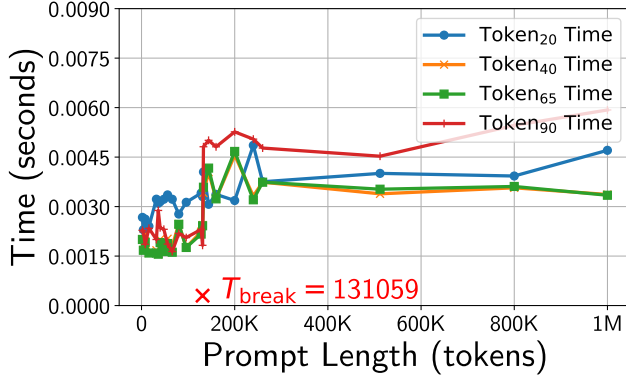


Figure 7: Per-token generation time of Gemini Flash-2.5-Lite exhibiting speculative decoding behavior.

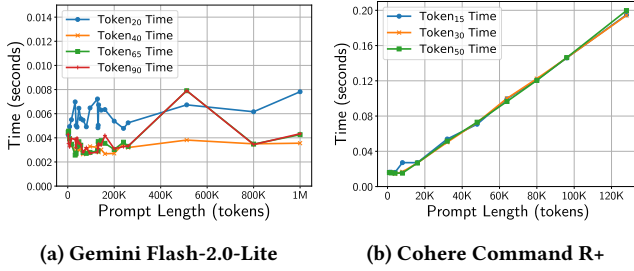


Figure 8: Per-token generation time of models where speculative decoding was not detected.

B Additional Details on Leaking Model Architecture Attack

B.1 Details of Scaling Runtime for Unseen Dimensions

Figure 9 shows our method for correcting the nonlinearities that arise from varying CUDA kernel selections described in §5.1. Assume a candidate architectural configuration and input length (H, L, A, I, T) . When applying the correction pipeline to the original runtime predictor, we take the following steps:

- In step ①, the candidate configuration is expanded into the full set of matmul operand shapes required to be executed to generate one output token for a prompt of length T and an LLM with the dimensions (H, L, A, I) .
- In step ②, we query the original training dataset and retrieve the closest configuration–prompt pair to the candidate. We also enumerate the matmul operand shapes for that closest sample.
- In step ③, for each matmul in both the candidate and the closest configurations, we feed the operand dimensions into the LightGBM kernel classifier (trained in the offline phase) to predict which cuBLAS kernel is likely to be selected.
- In step ④, for each matmul and its predicted kernel in ③, we use the kernel-specific random-forest regressor to predict a heuristic runtime. Denote these as $heuristic_{cand}$ and $heuristic_{close}$ for candidate and closest matmuls, respectively.

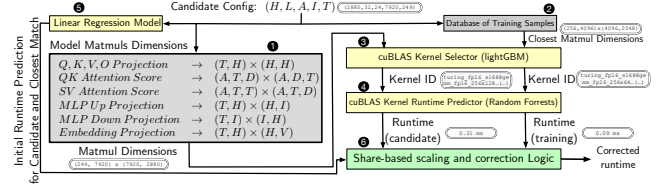


Figure 9: Non-linearity correction model. We use two predictors to predict the runtime of an unseen matmul dimension and correct the output of our model.

- In step ⑤, we run the original linear regression predictor on both the candidate and the closest configuration to obtain initial predicted runtimes $time_{cand}$ and $time_{close}$, respectively.
- In step ⑥, for every matmul that requires correction, we compute a scaling factor $f = \frac{heuristic_{cand}}{heuristic_{close}}$. We then scale the closest sample’s initial linear-predicted runtime $time_{close}$ for that matmul by f . To preserve consistency of this component-wise correction approach, we apply only the portion of this scaled time that is contributed by the target matmul, i.e., $f \cdot time_{close} \cdot share$, where $share$ is approximated by the matmul’s OPs divided by the total OPs of the block. If a matmul is marked as not needing correction, we instead use its share from the candidate’s initial linear prediction $time_{cand} \cdot share$. After applying this procedure to all matmuls and summing their adjusted contributions, we obtain the final corrected runtime for the component.

B.2 Detailed Theoretical Terms for All Operations

Following the theoretical analysis of runtime scaling terms per operation described in §5.2, we present the complete list of theoretical scaling terms extracted for each operation. These runtime scaling terms apply to a single decoder only.

Q/K/V/O projection:

$$time = \alpha_1 TH^2 d + \alpha_2 H^2 bc + \alpha_3 THbc \quad (5)$$

S = QK and Att = SV attention multiplications:

$$time = \alpha_1 T^2 Hd + \alpha_2 THbc + \alpha_3 T^2 Abc \quad (6)$$

attention softmax:

$$time = \alpha_1 T^2 Ad + \alpha_2 T^2 Abc \quad (7)$$

MLP up/down projections:

$$time = \alpha_1 THId + \alpha_2 THbc + \alpha_3 HIbc + \alpha_4 TIbc \quad (8)$$

B.3 Detailed Theoretical Terms for All Subcomponents

Below, we compose the scaling terms for the subcomponents from their underlying operations, accounting for the contributions of all decoder layers.

multi-head (eager) attention subcomponent:

$$\begin{aligned} time_{Att} = & \alpha_1 TH^2 Ld + \alpha_2 T^2 HLd \\ & + \alpha_3 T^2 ALd + \alpha_4 H^2 Lbc \\ & + \alpha_5 T^2 ALbc + \alpha_6 THLbc \end{aligned} \quad (9)$$

$$time_{Att} \sim T^2, H^2, A, L \quad (10)$$

MLP subcomponent:

$$\begin{aligned} time_{MLP} = & \alpha_1 THILd + \alpha_2 HILbc \\ & + \alpha_3 TILbc + \alpha_4 THLbc \end{aligned} \quad (11)$$

$$time_{MLP} \sim T, H^2, I, L \quad (12)$$

add and normalization subcomponent:

$$\begin{aligned} time_{Norm} = & \alpha_1 THLd + \alpha_2 HLbc \\ & + \alpha_3 THLbc \end{aligned} \quad (13)$$

$$time_{Norm} \sim T, H, L \quad (14)$$

embedding/projection (naive) subcomponent:

$$\begin{aligned} time_{EmbedProj} = & \alpha_1 THVd + \alpha_2 HVbc \\ & + \alpha_3 THbc \end{aligned} \quad (15)$$

$$time_{EmbedProj} \sim T, H, V \quad (16)$$

The term $THVd$ in EmbedProj can be replaced with HVd depending on whether the implementation projects the entire sequence or only the newly generated token in each round.

B.4 Cross-Family Generalization

As shown in Figure 10, Figure 11, and Figure 12, the predictor accurately captures the timing behavior of different target open models. The quadratic scaling patterns learned by the Eager (Corrected) predictor on the Llama 1B training dataset generalizes well to per-token generation time prediction across different model families, including Qwen2.5 1.5B, Phi-3.5-mini 3.8B, and Gemma2 2B. This holds despite differences in model architectures and parameter scales. This ability to generalize across model families further supports our argument that the generalizability of linear regression is one of the key reasons for adopting it as the foundation of our prediction approach.

B.5 Modeling FlashAttention

Runtime Scaling Terms. Below, we derive the terms for the FlashAttention2 kernel based on its original description [17].

$$\begin{aligned} time_{Flash} = & \alpha_1 TH^2 Ld + \alpha_2 T^2 HLd \\ & + \alpha_3 T^2 D^2 LM^{-1} bc + \alpha_4 THLbc \\ & + \alpha_5 H^2 Lbc + \alpha_6 DHLbc \end{aligned} \quad (17)$$

$$time_{Flash} \sim T^2, D^2, H^2, M^{-1}, L \quad (18)$$

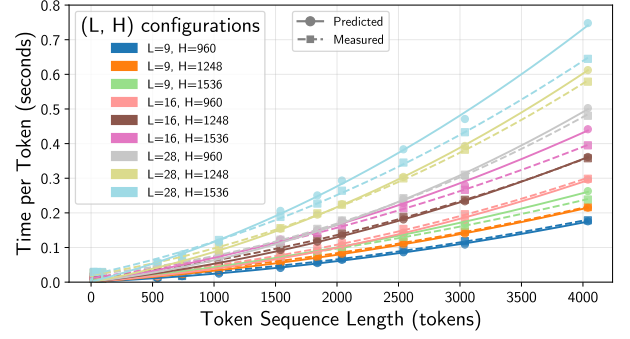


Figure 10: Predicted token-generation times (generated by a predictor trained on Llama 1B) vs. ground-truth times (obtained from inference on the Qwen2.5 1.5B model).

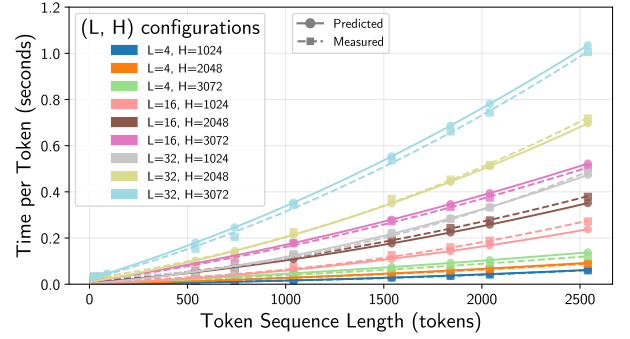


Figure 11: Predicted token-generation times (generated by a predictor trained on Llama 1B) vs. ground-truth times (obtained from inference on the Phi-3.5-mini 3.8B model).

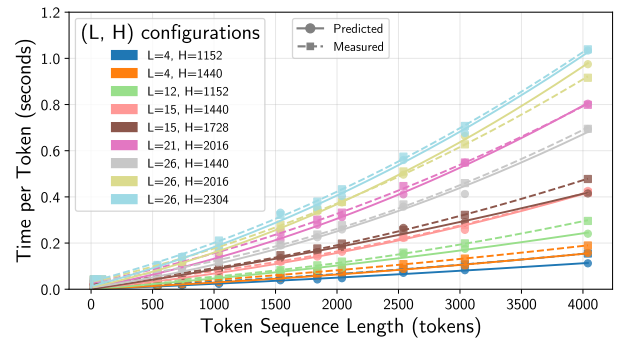


Figure 12: Predicted token-generation times (generated by a predictor trained on Llama 1B) vs. ground-truth times (obtained from inference on the Gemma2 2B model).

In these scaling terms, D denotes the head dimension of the model, and M represents the size of the GPU SRAM. In our experiments on the A10 GPU, we set M to 96 KB. For the experiments on the B200 GPU, we increase M to 228 KB.

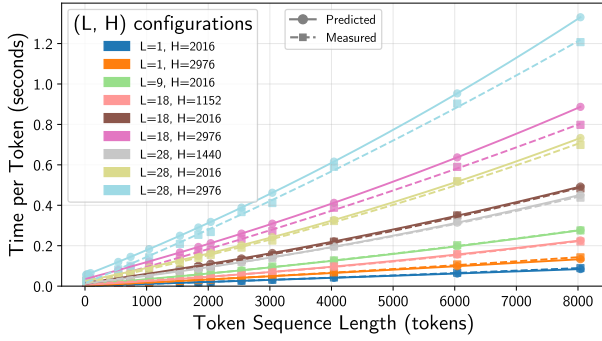


Figure 13: Predicted token-generation times vs. ground-truth times for FlashAttention2 kernel.

Prediction Accuracy. Figure 13 shows the per-token generation times predicted by our Flash2 predictor for different architectural configurations, which closely follow the measured FlashAttention inference times. These results demonstrate that our configurable prediction approach can be extended to support various implementations of the transformer architecture by adjusting a small number of runtime scaling terms, without requiring a new predictor design.

B.6 Modeling KV-Cache

Runtime Scaling Terms. We define two sets of runtime scaling terms: one for the prefill stage and one for the decoding stage of inference. For the prefill stage, we use the same scaling terms as in §B.5. We then revise the scaling terms for all transformer components to account for decoding-time execution, where each pass processes a single token instead of the full sequence, and attention is computed using cached key-value representations rather than recomputing them. Below, we describe the runtime scaling terms for the decoding stage of KV-cache-enabled inference.

attention subcomponent:

$$\begin{aligned} time_{Att}^{decode} = & \alpha_1 H^2 L d + \alpha_2 T H L d \\ & + \alpha_3 T H L b c + \alpha_4 H^2 L b c \\ & + \alpha_5 H D L b c + \alpha_6 H^2 L b c \\ & + \alpha_7 H L b c \end{aligned} \quad (19)$$

$$time_{Att}^{decode} \sim T, H^2, D, L \quad (20)$$

MLP subcomponent:

$$\begin{aligned} time_{MLP}^{decode} = & \alpha_1 H I L d + \alpha_2 H I L b c \\ & + \alpha_3 I L b c + \alpha_4 H L b c \end{aligned} \quad (21)$$

$$time_{MLP}^{decode} \sim H^2, I, L \quad (22)$$

add and normalization subcomponent:

$$time_{Norm}^{decode} = \alpha_1 H L d + \alpha_2 H L b c \quad (23)$$

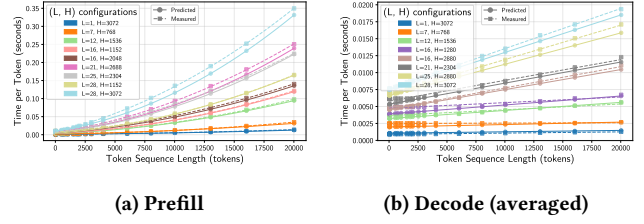


Figure 14: Predicted token-generation times vs. ground-truth times for KV-cache-enabled inference using FlashAttention2 kernel.

$$time_{Norm}^{decode} \sim H, L \quad (24)$$

embedding/projection (naive) subcomponent:

$$time_{EmbedProj}^{decode} = \alpha_1 H V d + \alpha_2 H V b c \quad (25)$$

$$time_{EmbedProj}^{decode} \sim H, V \quad (26)$$

Prediction Accuracy. Figure 14 shows the performance of our KV-Flash2 predictor across different architectural configurations. Figures 14a and 14b depict per-token generation times for the prefill and decoding stages, respectively. We observe that the hybrid runtime predictor accurately captures both the quadratic dependence on sequence length in the prefill stage and the linear behavior in the decoding stage. These results demonstrate that our runtime prediction framework can be extended to incorporate widely used inference-time optimization techniques, such as KV caching, with minimal modifications.

B.7 Grid Search Dimensions

We evaluated the following grids, comprising 1540 unique configurations, during the online phase evaluations in §5.5, where the only unknown parameters are H and L :

► Hidden size

$$\begin{aligned} H \in \{ & 32, 64, 128, 256, 512, 640, 768, 896, 960, \\ & 1024, 1152, 1248, 1280, 1408, 1440, 1536, 1664, \\ & 1792, 1824, 1920, 2016, 2048, 2112, 2240, 2304, \\ & 2432, 2560, 2688, 2880, 2944, 3072, 3168, 3200, \\ & 3264, 3360, 3456, 3582, 3710, 3840, 3968, 4096, \\ & 4608, 5120, 6144 \} \end{aligned}$$

► Number of layers

$$L \in \{1, 2, \dots, 35\}$$

When we fix all architectural parameters except one, we expand the search space for the target parameter to reduce bias induced by the discrete values covered in the grid on retrieval accuracy. The grid includes 47 distinct values for H and 44 distinct values for L . The resulting ranges for H and L are as follows:

► Hidden size

$H \in \{32, 64, 128, 256, 512, 640, 768, 896, 960, 1024, 1152, 1248, 1280, 1408, 1440, 1536, 1664, 1792, 1824, 1920, 2016, 2048, 2112, 2240, 2304, 2432, 2560, 2688, 2880, 2944, 3072, 3168, 3200, 3264, 3360, 3456, 3582, 3710, 3840, 3968, 4096, 4352, 4608, 4864, 5120, 5632, 6144\}$

► Number of layers

$L \in \{1, 2, \dots, 45\}$

For the remote API experiment, we use the same expanded parameter ranges as in the single-parameter retrieval setting, resulting in a search space of 2,068 (H, L) configurations.

C Open Science

To support the replicability of our results, we commit to making all artifacts related to this research publicly available. The scripts used to run the LLM inference, make the API calls, measure the timings, and analyze the experiment results are all accessible through a public repository (i.e., <https://anonymous.4open.science/r/LeakyLMs-615B/>), and we provide detailed documentation to facilitate their use by other researchers. We organized the repository to reflect the structure of the paper, allowing readers to easily locate the experimental setup corresponding to each attack and its subsections.