WAVE: Leveraging Architecture Observation for Privacy-Preserving Model Oversight

Haoxuan Xu*
University of Southern California
Los Angeles, CA, USA
xuhaoxua@usc.edu

Chen Gong*
University of Southern California
Los Angeles, CA, USA
cgong459@usc.edu

Beijie Liu*
University of Southern California
Los Angeles, CA, USA
beijieli@usc.edu

Haizhong Zheng
Carnegie Mellon University
Pittsburgh, PA, USA
haizhonz@andrew.cmu.edu

Beidi Chen
Carnegie Mellon University
Pittsburgh, PA, USA
beidic@andrew.cmu.edu

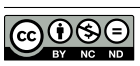
Mengyuan Li†
University of Southern California
Los Angeles, CA, USA
mli49061@usc.edu

Abstract

Large Language Models (LLMs) inference increasingly require mechanisms that provide runtime visibility into what is actually executing, without exposing model weights or code. We present WAVE, a hardware-grounded monitoring framework that leverages GPU performance counters (PMCs) to observe LLM inference. WAVE is built on the insight that legitimate executions of a given model must satisfy hardware-constrained invariants, such as memory accesses, instruction mix, and tensor-core utilization, induced by the model's linear-algebraic structure. WAVE collects lightweight PMC traces and applies a two-stage pipeline: (1) inferring architectural properties (e.g., parameter count, layer depth, hidden dimension, batch size) from the observed traces; and (2) using an SMT-based consistency checker to assess whether the execution aligns with the provisioned compute and the claimed model's constraints. We evaluate WAVE on common open-source LLM architectures, such as LLaMA, GPT, and Qwen, across multiple GPU architectures, including NVIDIA Ada Lovelace, Hopper, and Blackwell. Results show that WAVE recovers key model parameters with an average error of 6.8% and identifies disguised executions under realistic perturbations. By grounding oversight in hardware invariants, WAVE provides a practical avenue for continuous, privacy-preserving runtime monitoring of LLM services.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**; • **Computing methodologies → Machine learning**.

*Co-first authors.
†Corresponding author.



ASPLOS '26, Pittsburgh, PA, USA.

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790247

Keywords: Verifiable Inference; GPU Performance Counters; Model Fingerprinting; Hardware-assisted Security

ACM Reference Format:
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 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 21–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3779212.3790247

1 Introduction

Large Language Models (LLMs) have rapidly expanded from enterprise-only deployments to widespread use by smaller organizations and individual users. As inference usually runs on infrastructures outside the user's direct control, *faithful execution* becomes essential: the provider should faithfully execute the promised model size and expend the intended computational effort, rather than merely returning outputs that look plausible.

A representative example is model-as-a-service (MaaS), where users outsource inference to model providers. Providers are expected to maintain output quality using large model, but they also face strong cost pressures. As a result, a cost minimizing provider may quietly modify the deployment in ways that reduce compute while keeping answers acceptable for many prompts. For example, the provider might substitute a smaller model that still produces superficially adequate responses, thereby lowering GPU time and energy. Such substitutions are often hard to detect from outputs alone, particularly when privacy and isolation constraints prevent users from instrumenting the provider's system or auditing the full software stack [25, 26, 30].

These realities create an urgent need to *monitor whether the provider faithfully executed the promised inference computation, including the intended model configuration and computational effort, under limited observability*, so users can distinguish compliant execution from unexpected or erroneous computation. This is a practical concern across outsourced

LLM serving, multi party coordination, and security sensitive deployments where faithful compute supports both accountability and reliable anomaly identification.

Existing LLM monitoring solutions. The research community has pursued several complementary directions for monitoring and verifying LLM inference. Many systems focus on auditing *model identity* or correctness, which also constrains effort because a specific model entails a specific computation. Software based audits rely on execution derived signals, such as SVIP [34], which embeds secret markers in intermediate outputs and checks provider responses against hidden state challenges. Timing based side channel analysis can infer token counts from response latencies [32], while watermarking embeds statistical patterns for output provenance [17]. Cryptographic approaches aim for stronger guarantees: zero knowledge proofs (ZKPs) provide model agnostic correctness at additional computational cost, with zkLLM and ZKTorch demonstrating end to end proofs for LLM inference [4, 33]. Other work targets *resource accountability* more directly: ALIBI leverages nested virtualization to measure CPU and memory usage [28], and systems like VeriCount and T Counter integrate trusted execution environments (TEEs) to deter resource inflation [10, 35]. More recently, GPU TEEs have sought to protect GPU execution through isolation [24].

1.1 Motivation: Leverage Architecture Observation for Model Monitoring

In this work, we shift the focus to use runtime hardware-level signals to monitor faithful LLM inference serving. LLM inference naturally leaves rich, weight-agnostic traces on the GPU, making the hardware a robust vantage point for observing what computation was actually performed. Building on this perspective, our goal is not to recover or certify exact model weights. Instead, we audit whether the provider executes the *expected* model configuration, such as architecture and model size, which directly determines inference behavior and computational effort.

Leveraging hardware-level observations (also known as GPU performance monitoring counters, or PMCs) for model monitoring offers several compelling benefits: (i) Privacy-preserving. The monitor never inspects model weights or user data directly. Instead, it relies only on content-agnostic aggregate counters (e.g., FLOPs, tensor-core occupancy, memory transactions), revealing nothing about the query contents. (ii) Evasion-resistant. If an adversary substitutes a smaller or different model, such a substitution alters the fundamental compute–memory ratios and the joint distribution of counters. To mimic the footprint of a larger model, the adversary would have to perform commensurate work, which negates any resource savings and introduces detectable inconsistencies across correlated metrics. (iii) Low Overhead Potential. Reading global GPU PMCs is supposed to be lightweight and non-intrusive compared to heavy software instrumentation or cryptographic proofs, and it can run alongside live inference. (iv) Composability. PMC-based monitoring complements existing trust mechanisms such as TEE and attestation. For example, existing GPU TEE only provides evidence of GPU initial status, while PMC collected inside a TEE can serve as tamper-resistant runtime evidence. Ultimately, this approach could be integrated into GPU firmware or drivers for native, on-device attestation of model activity. Collectively, these benefits motivate a hardware-centric approach that treats the GPU as an impartial witness to LLM execution.

1.2 Wave: Hardware Oversight of LLM Inference

In this paper, we present Wave, a hardware-grounded runtime verification framework that uses GPU PMCs to monitor the structure and effective model size during LLM inference. Wave aims at answering a core research question:

How reliably and to what granularity can GPU performance counters be used to fingerprint an LLM inference?

To our knowledge, Wave is the first framework to use GPU PMCs for LLM oversight in a privacy-preserving fashion, which requires no exposure of model weights or user data. Wave operates by collecting a lightweight trace of key PMC metrics during LLM inference, and then applying a two-stage analysis pipeline. In the first stage, the trace is used to infer architectural properties of the model (*etc.* number of layers, hidden dimension, and batch size) using analytical models and calibrated profiling of the model's linear algebra operations. In the second stage, Wave performs a consistency check: it uses the inferred properties, along with an SMT-based constraint solver, to verify that the observed execution aligns with the claimed model architecture and the provisioned computing resources. By verifying hardware-level invariants (e.g., FLOPs vs. memory accesses) against the expectations for the advertised model, Wave can detect discrepancies indicative of an altered model being executed.

We have prototyped and evaluated Wave on a variety of GPU hardware and several popular LLM inference workloads. Our evaluation yields promising results. Wave can accurately recognize architectural invariants of large models from their PMC traces, enabling detection of incorrect or substitute models. For example, Wave can estimate the depth and structures of the model with an average error of 6.8%, effectively distinguishing different LLM models.

Beyond the specific mechanisms explored in this work, we aim to clarify the feasibility and limits of using hardware-level signals for LLM oversight and to extract architectural insights for future systems. Although current GPU platforms lack secure, tenant-accessible interfaces for fine-grained PMC collection, our results suggest that with appropriate hardware and system support, such as low-overhead PMC access and TEE-integrated monitoring, mechanisms like Wave

could become practical. In this context, Wave serves as a design point illustrating how hardware-level observability can enable reliable, privacy-preserving LLM oversight. The contribution of the paper is summarized as follows:

- **Hardware-based Model Oversight:** We propose Wave, the first framework to utilize GPU performance monitoring counters for privacy-preserving LLM inference oversight. Wave introduces the idea of using microarchitectural execution traces as a reliable signature of the size and configuration of a model, without requiring any access to the internals or output of the model.
- **Two-Stage Verification Pipeline:** We design a novel verification pipeline that combines architectural inference (estimating model properties from hardware events) with SMT-based formal verification against claimed specifications. This approach provides a rigorous check on model integrity at runtime, catching discrepancies between the provider's claim and the actual execution.
- **Feasibility and Generality:** This paper evaluates Wave on contemporary GPU hardware and demonstrate its feasibility across several open-sourced LLM inference models. Wave shows that it achieves high accuracy in model fingerprinting (average 6.8% error in key parameters) and effectively detects model misuse or downsizing. Our prototype is available at https://github.com/sept-usc/Wave.

2 Background

2.1 Large Language Models

The core of most state-of-the-art LLM is the Transformer architecture, whose computational footprint is largely governed by dense linear algebra.

Transformer Architecture. Transformer architecture stacks self-attention and feed-forward (FFN) blocks [38]. The bulk of runtime arises from dense matrix multiplications (GEMMs) in (i) the self-attention module, Q/K/V and output projections, and (ii) the FFN (e.g., $d \times d_{\text{ffn}}$ and $d_{\text{ffn}} \times d$). Here d denotes the hidden size of the model state. Contemporary large language models are predominantly *decoder-only* (*e.g.*, GPT [29], LLaMA [36], Qwen [3]). Although different architectures vary slightly, for example in the number of projections in the FFN (denoted λ in this paper, with GPT–2 using $\lambda = 2$ and LLaMA using $\lambda = 3$), their inference proceeds in two phases: the *prefill phase*, which processes the full input prompt and computes keys/values for all tokens to populate the cache, and the *decoding phase*, which generates tokens autoregressively, one at a time. During decoding, the model computes projections only for the current token, while reusing cached keys and values from the prefill and previous steps to compute attention scores.

Model Parameters and Their Effects. Architecture parameters jointly determine both the expressiveness of the model and the execution cost. For example, *hidden size* (d) scales the dimensionality of internal representations, enhancing expressive power but proportionally increasing matrix dimensions and thus computational and memory demands. Likewise, the *number of attention heads* (h) allows the model to capture more diverse relational features, though additional heads raise complexity and resource consumption. Table 4 in Appendix B provides a broader overview of the representative parameters of the LLM model.

2.2 GPU Performance Counter

Performance Monitoring Counter (PMC) is widely used in both CPUs and GPUs for characterizing hardware execution, offering fine-grained visibility into activities such as instruction throughput, memory utilization, and pipeline stalls. Modern GPU vendors provide comprehensive toolchains to collect PMCs, such as NVIDIA Nsight Systems [8] and AMD's `rocprof` [2]. In this work, we use NVIDIA Nsight Compute [7] to profile GPU execution, which has been widely used in kernel-level diagnosis and modeling [31, 42].

However, unlike CPU PMCs (which are deeply integrated into the processor pipeline and incur minimal overhead), current GPU PMC mechanisms have several limitations. First, whereas CPU hardware allows only a small fixed number of PMC events to be measured per run (due to limited hardware counter registers), those events can be captured in real time with negligible overhead. In contrast, GPU profiling tools like Nsight Compute allow collecting a broad range of metrics in one session, but they achieve this by internally re-executing the kernel multiple times if many events are requested. Moreover, GPU counters today are typically aggregated at the kernel level (not per thread), and some fine-grained events are not exposed directly without instrumentation (e.g. using `CUPTI` [6] or `NVBit` [39]). These limitations arise from the GPU's SIMT architecture with thousands of parallel threads. As a result, when using existing GPU PMC toolchains, we restrict our focus to *global execution metrics* (such as the total number of floating-point operations (FLOPs) and aggregate global memory transactions). While current collection still incurs substantial overhead, we expect future GPU architectures to enable direct, real-time reporting of global execution metrics just like CPU PMC.

3 Motivating Example: Inferring LLM Scale via PMC-Based FLOPs Measurement

In this section, we present an illustrative example showing how GPU PMC observations during model inference can be leveraged to estimate and correlate with the model size. We first define a matrix-based computation model involved in LLM inference and then show how these theoretical insights link to actual PMC data to reveal model complexity.

3.1 Modeling LLM Inference

To infer the model scale from hardware signals, we first need an implementation–invariant account of the work performed. Although detailed kernel implementation, models, and schedules may vary between deployments, *math does not*. We therefore model transformer-based LLM inference as a deterministic set of matrix operations.

Insight 1: Architecture-defined computational laws: The execution of Transformer-based LLMs inevitably conforms to the mathematical constraints of their matrix operations, independent of detailed kernel implementations.

In other words, the model's architecture fixes a set of linear algebra operations (matrix multiplications and elementwise additions) that must occur for each token inference, which lets us analytically model the computation required. Any mismatch suggests the GPU may not be executing inference faithfully. By formulating LLM inference in terms of its fundamental linear operations, we can later link hardware observation to the theoretical computation needed from its model parameters (layer, hidden dimension, etc.).

Decoder-only Transformer. Concretely, for the most common *decoder-only* Transformer, we model the per-layer forward pass as a sequence of linear-algebra primitives. Each operation is represented by a tuple of integer parameters, with its arity depending on the operator type. We use the notation $(X \times Y)$ to denote a matrix with X rows and Y columns, and extend it to batches so that A_b indicates the number of parallel matrices.

Matrix multiplication.

$$\mathrm{MUL}\langle A_b, A_x, A_y, B_y\rangle : (A_x \times A_y)\cdot(A_y \times B_y) \;\rightarrow\; (A_x \times B_y),$$

applied A_b times across the batch.

Matrix addition.

$$\mathrm{ADD}\langle A_b, A_x, A_y\rangle : (A_x \times A_y) + (A_x \times A_y) \;\rightarrow\; (A_x \times A_y),$$

also applied A_b times across the batch.

A model execution is then naturally expressed as $S = [OP_1, \ldots, OP_n]$ with $OP_i \in \{\mathrm{MUL}, \mathrm{ADD}\}$.

In this abstraction, we deliberately restrict the operation set to $\{\mathrm{MUL}, \mathrm{ADD}\}$, and Figure 1 illustrates this *abstraction-level* operator set for one Transformer layer. Building on this abstraction, we further designate a simplified *canonical sequence* S^* template for structurally significant matrix operations. This S^* serves as the Transformer architecture-level ground truth for subsequent checks.

Linking PMC Observations to LLM Model. Given the above model of LLM inference, we now consider how GPU performance counters can bridge from observed execution to the model's complexity. This approach is motivated by two key observations. *First*, models of different sizes and parameter configurations exhibit distinct computational complexities and execution patterns, which can be captured through GPU performance counters. *Second*, GPU workloads, which are dominated by operations such as matrix multiplications and additions, involve far fewer branches than typical CPU computations, so the execution path for a given model remains relatively fixed. This consistent execution results in stable PMC readings, allowing observed hardware events to be reliably mapped to the model's theoretical computational complexity and its parameters, without the need to directly access the model's weights or other sensitive data.

3.2 Example: FLOPs-Guided Model Scale Estimation

A straightforward approach to estimating model scale from PMCs is to use the number of floating-point operations (FLOPs) as a proxy for model size, since larger models naturally require more FLOPs to generate a single token. To formalize this, for each kernel k, we denote by $X(k)$ the *observed* value of performance counter X collected from the GPU. Given an execution sequence S, the complete trace is written as $PMC(S)$, representing the vector of selected PMC metrics across all kernels in S. In the context of this example, we focus specifically on the FLOPs dimension $F(k)$.

We can also obtain the theoretical FLOPs from previous LLM inference modeling. Let L denote the number of layers, d the hidden size, h the number of attention heads (with per-head dimension d/h), d_{ffn} the feed-forward size (assumed to be $d_{\text{ffn}} = 4d$), b the batch size, s the sequence length. The model size can be represented as

$$M = L \cdot \left(4d^2 + \lambda d \cdot d_{\text{ffn}}\right),$$

where λ is the number of gates in the FFN. In this expression, the term $4d^2$ corresponds to the query, key, value, and output projection matrices in the MHA (each of shape $d \times d$), while $\lambda d d_{\text{ffn}}$ accounts for the weight matrices in the FFN (each of shape $d \times d_{\text{ffn}}$). This formula approximates the total parameter count, ignoring low-order terms such as biases and normalization parameters.

Table 1 summarizes the FLOPs for each major operation in the *decoding phase*. All operations are applied independently to b examples in parallel. Specifically, the Q/K/V/O projections are realized as matrix multiplications $(1\times d)\cdot(d\times d) \;\rightarrow\; (1\times d)$, while the FFN up/down transformations are $(1\times d)\cdot(d\times d_{\text{ffn}})$ and $(1\times d_{\text{ffn}})\cdot(d_{\text{ffn}}\times d)$. Since the feed-forward dimension d_{ffn} is typically proportional to the hidden dimension d, the FLOPs of all these projections scale *quadratically* with d, as shown in Table 1. Given that each generated token passes through all L layers, the total per-token FLOPs grow *linearly* with L and can be expressed as $O(Lbd^2)$.

On the other hand, the model size can also be expressed as $O(Ld^2)$. The increase of d directly enlarges the FLOPs of a single projection matrix multiplication, while the increase of L is reflected in the total FLOPs across all layers. This correspondence shows that FLOPs and model size grow in a closely aligned manner, which allows accurate estimation of model scale from either measured or theoretical FLOPs.

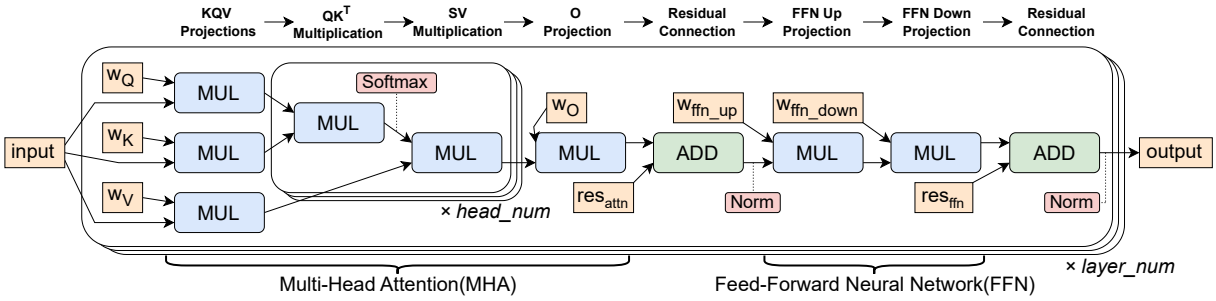
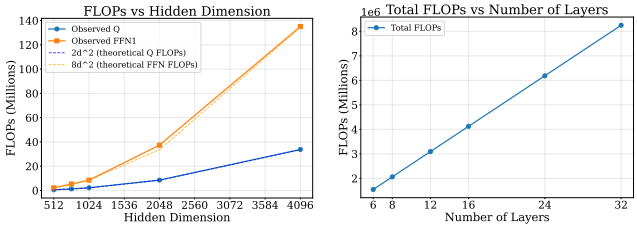


Figure 1. Computation flow of a Transformer layer of GPT-2 style, repeated over heads and layers.

Module	Operation	FLOPs Eq.	O-notation
MHA	QKV proj	$3 \times (2bd^2)$	$O(bd^2)$
	$QK^\top$ mul	$2bsd$	$O(bsd)$
	SV mul	$2bsd$	$O(bsd)$
	O proj	$2bd^2$	$O(bd^2)$
FFN	FFN Up proj	$2bdd_{\text{ffn}}$	$O(bd^2)$
	FFN Dn proj	$2bd_{\text{ffn}}d$	$O(bd^2)$
	Total per layer		$O(bd^2)$

Table 1. FLOPs per operation in a Transformer layer. The last column shows the asymptotic complexity. s represents the current context size of the token.



(a) FLOPs vs. d. **(b)** FLOPs vs. L.

Figure 2. Measured FLOPs under different LLaMA model configurations. Left: scaling with hidden size d (fixed $L = 6$) for Q projection and FFN up projection; Right: scaling with number of layers L (fixed $d = 1024$). Dashed lines indicate theoretical predictions from Table 1.

Experiment Results. We collect FLOP-relevant metrics using Nsight Compute on an NVIDIA 5080 16GB GPU and observe that the aforementioned quadratic and linear relations with d and L are accurately reflected in hardware measurements. Data are collected with batch size $b = 1$ using LLaMA models with varying hidden dimensions d (with $d_{\text{ffn}} = 4d$) and number of layers L. Figure 2a shows that for both Q projection and FFN Up projection, FLOPs increase *quadratically* with the hidden dimension. Figure 2b shows that the total FLOPs grow *linearly* with the number of layers L. These results closely match the theoretical analysis and further confirm the strong correlation between FLOPs and model size.

3.3 Challenges in Practical PMC-based Monitoring

The illustrative example above shows that hardware observations can accurately reflect the claimed model complexity in an experimental setting where we assume that matrix-related kernels can be reliably located via trustworthy kernel names. In practice, however, an untrusted deployer may relabel or obfuscate kernel names or indices and reshape execution (e.g., by fragmenting matrix operations, inserting padding, or reordering kernels) to mislead a monitor. Hence, a practical PMC-based monitor must be robust to untrusted metadata and adversarial workload manipulations, yet still deliver high-confidence conclusions about which model (e.g., model size and configurations) is actually running.

Key challenges. We focus on three challenges that drive the design of a robust verification system:

- **Execution-trace inference from raw PMCs.** Without trustworthy kernel identifiers or indices, the monitor must infer compute patterns directly from noisy, multiplexed hardware signals. This entails (a) selecting a proper subset of GPU PMC events that maximizes identifiability and stability; and (b) mapping PMC trajectories to model execution stages (e.g., prefill vs. decode, MHA vs. FFN) without relying on symbolized labels.
- **Recovering model parameters from traces.** The monitor should infer key model parameters (e.g., layer count, hidden size, parameter scale) at finer granularity.
- **Evasion tactics and robustness.** The monitor must remain reliable in the presence of adversarial behaviors and providing worst-case detectability guarantees under bounded manipulation.

4 Wave Overview

In this section, we introduce Wave, a framework that combines trace-pattern matching with formal verification to reliably recover the parameters of the inference model actually being executed. We begin by describing our threat model and assumptions, followed by an overview of Wave.

4.1 Objectives and Assumptions

The goal of Wave is to verify that an observed LLM inference execution corresponds to an intended model structure without directly scanning model weights or internal instrumentation. In our scenario, we assume that the specific parameters (model size, precision, etc.) of an inference model are public, for example, provided by the model deployer claims to run an inference model with for a user request. However, verifying the exact values of model weights or distinguishing between a base model and its fine-tuned variants is explicitly out of scope for this work. A PMC trace is collected by a trusted GPU hardware and cryptographically signed using hardware-root keys, preventing the provider from modifying or fabricating traces. These signatures can be validated through standard attestation mechanisms, similar to those used in TEEs. The resulting trace is then delivered to the verifier. The verifier uses Wave to check whether the executed model matches the claimed model size. The verifier itself can vary depending on privacy requirements: For example, if inference is not shared across multiple users and it is secure to expose all model structures, the cloud user may serve as the verifier; alternatively, a TEE or in-GPU-device verifier can be employed to ensure that only the verification result is revealed while the raw PMC trace remains hidden. In addition, we make the following assumptions about the deployment environment and adversary capabilities:

Trusted Hardware Counters. We assume that the hardware PMCs are collected directly from the GPU during runtime and cannot be manipulated by the adversary. An attacker (*a.k.a.* model deployer) can only delay, reorder, or pad operations at GPU kernel level to confuse the monitor, but they cannot forge the physical metrics reflecting the real execution (such as instruction counts, etc.) reported by the GPU.

Untrustworthy Model Deployer. We assume the model deployer is untrusted with respect to the model actually executed (*e.g.* they may run a smaller model than claimed or even directly sample tokens without using any model to inflate the token count for monetary gain) and may modify GPU kernels (*e.g.* rename functions or adjust launch parameters). However, we exclude availability attacks that would be easily detectable by the user. In this paper, we assume the model is executed using a standard deep learning framework (e.g., PyTorch or TensorFlow) on a single GPU for inference, with KV Cache enabled during the decoding phase to avoid redundant computation. Distributed inference is not considered. We also exclude system-level optimizations such as vLLM [20] and SGLang [46], which involve complex kernel and operator fusion. Such approaches fall outside our scope and are left for future work.

Secure Connection. We assume the verifier can obtain an authenticated, attested hardware-trace report bound to a specific inference run. This can be accomplished via a TEE or by a trustworthy GPU; with more discussed in Section 7.

4.2 Wave Design Overview

Wave adopts a two stage pipeline: (1) from noisy, implementation–dependent GPU PMC traces, infer an interpretable architectural sketch; and (2) formally check whether any canonical explanation consistent with these traces could violate the provider's promise. This split separates signal interpretation (robust inference) from claim verification (ensuring the execution matches the provider's advertised model).

Stage I — Execution Trace Inference. Stage I converts raw GPU PMCs into an interpretable sketch. Each kernel k is mapped to a 3-D signature $\big(F(k),\, r_{sh}(k),\, B_{tot}(k)\big)$ (detailed in Section 5.1). By detecting token- and layer-level periodicities, we segment the trace into per-layer windows. Within each window, signature patterns are aligned with the canonical per-layer order (Q/K/V/O projections; FFN with 2 or 3 gates) to assign roles and estimate $(L, d, d_{\text{ffn}}, b)$. In this way, Stage I provides a layer-wise sketch and parameter estimates.

Why Stage II after Stage I. While Stage I yields a clean sketch from noisy PMCs, it is *insufficient as a verifier* under adversarial or non-ideal implementations:

1. **Local inference lacks global consistency.** Stage I may infer parameters from a *subset* of matched kernels. Without enforcing cross-kernel consistency within a layer (and across layers), a deployer can minimally alter or disguise a few kernels to bias local estimates while keeping the overall trajectory plausible. The correctness claim must be checked *against the entire canonical structure.*
2. **Self-correlation absorbs small noise, not systematic attacks.** The token-layer recovery permits small shifts (e.g., ±1) and can absorb limited insertions, but it is fragile under persistent fission/fusion/padding or targeted insertions that preserve coarse periodicity while breaking alignment. Hence, Stage I has limited anti-evasion power.
3. **Non-ideal implementations confound kernel identification.** Practical Transformer stacks often split/fuse GEMMs (e.g., QKV split/fusion; FFN with 2 vs. 3 GEMMs), reorder or pad kernels. Such deviations blur kernel lists and defeat signature-only location heuristics. Verification should therefore be based on *explicit space of structure-preserving transformations.*

Stage II — Model Structural Consistency Verification. Stage II verifies whether there exists a canonical sequence S^* (under allowed transformations) that both fits the observed counters (within tolerance) and *violates* the provider's promise (e.g., on model size/configuration). We capture attacker capabilities (e.g., kernel fission, padding) as structure–preserving transformations over a canonical template and encode them, with PMC bounds and hardware granularity, into an SMT query. If the instance is `sat`, the solver returns a violating witness and the verifier outputs `Fail`; otherwise, `unsat` certifies `Pass` under the stated threat model.

5 Wave Design

5.1 Execution Trace Inference

We introduce a lightweight method that infers an interpretable, stage–level alignment of the GPU kernel stream directly from PMC observations and estimates model parameters from key matrix computation kernels. Specifically, this approach has three objectives:

1. **Token- and Layer-level Periodicity Recovery**: This consists of two components: a large period capturing token-level periodicity during the decoding phase, and a small period capturing transformer-layer periodicity for each token. The small-period values are then averaged across layers for subsequent use.
2. **In-layer Kernel Role Assignment**: We identify and assign the correct roles to kernels based on the known computation sequence S in transformer (e.g., identifying which kernel corresponds to Q projection).
3. **Model Parameter Recovery**: We use the relationship between metrics and model parameters to perform direct parameter recovery. If direct recovery is not performed, the metrics can also be passed to Stage II for structural consistency verification.

To achieve the above goals, simple heuristics fail to reconstruct the sequence S^* of matrix operations for two reasons. First, there is no reliable way to identify matrix kernels from labels: kernel names, indices, and launch orders vary across libraries and builds and can be spoofed, making it difficult to locate GEMM-heavy kernels or infer matrix sizes from identifiers. Second, single-counter cues are ambiguous and locally fragile: a FLOP peak does not necessarily indicate a matrix multiplication nor determine matrix size; it may originate from attention or the FFN depending on sequence length, fusion choices, and scheduling. Moreover, in-layer housekeeping kernels (e.g., layout transforms, epilogues) and fused/split implementations perturb local timing and counters while preserving the coarse stage order. Consequently, we adopt a *timeline-aware* and *multi-metric* approach that combines compute-mix ratios, shared-memory intensity, cache/miss-sector patterns, and order context to align kernels and recover parameters across implementations.

Chosen PMC Events and Rationale. We rely on two complementary families of PMCs that answer different needs. Compute-path counters quantify the amount of computation performed by each kernel and identify the active data type (*dtype*), while data-movement counters measure traffic beyond L1/TEX and indicate whether memory accesses predominantly target shared or global memory. These metrics are inherently tied to the underlying computation, making them useful for kernel identification and parameter recovery.

- *Compute-path Counters.* We collect scalar bucket metrics: $\{\mathrm{H(alf)}, \mathrm{F(loat)}, \mathrm{D(ouble)}\} \times \{\mathrm{FMA}, \mathrm{ADD}, \mathrm{MUL}\}$, along with several `sm__ops_path_tensor_src_* .sum` counters (GPU-dependent). These constitute the FLOPs $F(k)$ for a given kernel k, which are subsequently used in periodicity detection and Stage II verification.
- *Data-movement Counters.* For each kernel, we track two key memory signals: (i) off-L1/TEX traffic volume, which scales with d and d_{ffn} and aids in parameter recovery; (ii) the shared-vs-global balance, which serves as a cue to distinguish attention kernels from projection kernels during role assignment. Using relevant metrics, we can compute L1 global load/write bytes ($B_{\mathrm{ld}}(k)$, $B_{\mathrm{st}}(k)$), executed global load/store instruction counts ($I^{\mathrm{GL}}_{\mathrm{ld}}(k)$, $I^{\mathrm{GL}}_{\mathrm{st}}(k)$), as well as the corresponding shared memory counts ($I^{\mathrm{SH}}_{\mathrm{ld}}(k)$, $I^{\mathrm{SH}}_{\mathrm{st}}(k)$).

Detailed metric names and their descriptions are provided in Appendix C.2 (Table 5), with nine PMC metrics in total.

5.1.1 Token- and Layer-level Periodicity Recovery.

Since kernel names are unreliable, we instead characterize each kernel by a compact 3-D signature:

- **Kernel FLOPs ($F(k)$):** The aggregate floating-point operations performed. This metric effectively distinguishes compute-intensive kernels (*e.g.*, GEMMs) from pure memory operations such as KV-cache management.
- **Shared Memory Ratio ($r_{\mathrm{sh}}(k)$):** Defined as $r_{\mathrm{sh}}(k) := \frac{I^{\mathrm{SH}}_{\mathrm{ld}}(k)+I^{\mathrm{SH}}_{\mathrm{st}}(k)}{I^{\mathrm{SH}}_{\mathrm{ld}}(k)+I^{\mathrm{SH}}_{\mathrm{st}}(k)+I^{\mathrm{GL}}_{\mathrm{ld}}(k)+I^{\mathrm{GL}}_{\mathrm{st}}(k)}$. This ratio serves as a structural signature, distinguishing tiling-intensive kernels (*e.g.*, Attention) from standard GEMMs based on their reliance on on-chip shared memory.
- **Off-L1 Traffic ($B_{\mathrm{tot}}(k)$):** Defined as $B_{\mathrm{tot}}(k) := B_{\mathrm{ld}}(k) + B_{\mathrm{st}}(k)$. This measures the global memory traffic volume, which also aids in differentiating memory kernels from compute kernels.

We stack them into $x_t = [F(k), r_{\mathrm{sh}}(k), B_{\mathrm{tot}}(k)]$, where t means the t-th kernel in execution order, and measure kernel similarity via cosine similarity. By self-correlating the feature stream, we uncover two nested periods:

- A *large period* corresponding to kernels per token.
- A *smaller period* nested inside, corresponding to the per-layer structure.

By counting how many large periods fit in a run, we recover the token count. This can be compared with the received reply to detect naive token inflation, where extra tokens are randomly sampled on CPU rather than generated on GPU. By counting how many small periods occur within each large one, we recover the layer count. The detailed algorithm and formulation are given in Appendix C.

We then average the PMCs across both tokens and layers to serve as inputs for subsequent in-layer role assignment. At the same time, we track the variance during averaging: low variance indicates that all tokens are produced by the same model, preventing advanced token inflation attacks

where tokens from different model sizes (*e.g.*, large vs. small) are interleaved.

5.1.2 Score-based in-layer Role Assignment. After layers are segmented, we assign roles to matrix-heavy kernels in fixed order: the $Q/K/V$ projections, the attention block, the output projection, and the FFN blocks. We call a kernel *matmul-like* if it is a candidate of matrix-heavy kernels, and *attn-like* if it matched characteristics of the attention block. We call a kernel *add-like* if it is not matrix-heavy but has nonzero $F(k)$. Elementwise adds appear right after the output projection and after the last FFN block. We define the sequence S^* as:

$$S^* = [\text{QKV, Attn, O, Add, FFN, Add}],$$

where Attn is *attn-like*, Q, K, V, O, FFN are *matmul-like*, and Add is *add-like*. Only the multiplicities vary by implementation:

$$|\text{QKV}| \in \{1\ \text{(fused)}, 3\ \text{(split)}\}, |\text{FFN}| = \lambda \in \{2, 3\}\ \text{gates}.$$

We map kernels to S in two steps:

1. Identify *matmul-like*, *attn-like*, and *add-like* kernels using kernel FLOPs computation $F(k)$ and shared memory ratio $r_{\text{sh}}(k)$.
2. Respecting the template S, assign specific roles (Q/K/V/O, FFN) by comparing off-L1 global traffic $B_{\text{tot}}(k)$ and $F(k)$ within the layer.

Details of the procedure are given in Appendix C.5.

5.1.3 Model Parameter Recovery. In addition to the recovered layer count L and token count T, we can also infer the batch size b and hidden dimensions from the byte signals $B_{\text{ld}}, B_{\text{st}}$. Specifically, QKV stores reveal bd, projections yield d, and FFN loads determine d_{ffn}.

These give direct estimates of the batch size b and FFN ratio $r = d_{\text{ffn}}/d$. From these values, the approximate parameter size of the model follows:

$$M \approx L\left(4\,d^2 + \lambda\, d\, d_{\text{ffn}}\right)$$

We first determine the model's *data width* D by identifying the dominant scalar FMA bucket in {HFMA, FFMA, DFMA}, which corresponds to FP16/BF16, FP32, or FP64 respectively, thereby fixing the data size $D \in \{2, 4, 8\}$ B.

Assuming the global load and store traffic of projection kernels is close to theoretical values in the decoding phase, we can form equations for split-QKV projections (the fused case is similar). For example, for the Q projection:

$$B_{\text{st}}(k_{\text{Q}}) = D \cdot \hat{b}\hat{d}, \quad B_{\text{ld}}(k_{\text{Q}}) = D \cdot (\hat{b}\hat{d} + \hat{d}^2).$$

Solving these equations gives $\hat{b}$ and $\hat{d}$.

Next, we recover the FFN dimension using its relative load ratio:

$$B_{\text{ld}}(k_{\text{ffn\_up}}) = D \cdot \left(\hat{b}\hat{d} + \hat{r}\hat{d}^2\right), \quad \hat{d_{\text{ffn}}} = \text{round}(\hat{r}\hat{d}).$$

Finally, substituting $\hat{d}$ and $\hat{d_{\text{ffn}}}$ into the formula above yields the model size estimate. Further details of the recovery process are given in Appendix C.6.

5.2 Model Structural Consistency Verification

This section casts *model structural consistency verification* as an existence check: given the observed PMCs, we ask whether there exists a layer-wise *canonical* explanation S^* consistent with the architecture and with the semantics of allowed transformations. Rather than reconstructing an implementation-level kernel list, we verify satisfiability of structural (algebraic) constraints induced by the canonical template. Because these constraints are stated at the semantic level, they are invariant to kernel fission/fusion/reordering. The following subsections formalize the observables and assumptions and encode the decision as an SMT query against the provider's promised metric.

5.2.1 Verification Modeling. To enable structural consistency verification, we first formalize the modeling assumptions that connect observable PMCs, canonical layer structure, and possible adversarial transformations.

Observables. The verifier does not directly observe the canonical execution S^*, but only hardware-level performance monitoring counters PMC_{obs}. For any sequence S, let $\text{PMC}(S)$ denote its theoretical counter values aggregation. Because the mapping from sequences to PMCs is underdetermined, multiple realizations S' may satisfy

$$\text{PMC}(S') \approx PMC_{\text{obs}}.$$

"$\approx$" means that for kernel i and metric $X \in \{\text{FLOPs}, \text{GL}, \text{GW}\}$, the observed value X_i must lie between an analytical lower bound lb_i^X and an upper bound ub_i^X. For a multiplication operator $\text{MUL}\langle A_b, A_x, A_y, B_y\rangle$, the bounds are defined per instance (with A_b repetitions across the batch) as $\text{lb}_i^{\text{FLOPs}} = 2A_x^iA_y^iB_y^i$, $\text{lb}_i^{\text{GL}} = A_x^iA_y^i + A_y^iB_y^i$, $\text{lb}_i^{\text{GW}} = A_x^iB_y^i$, with $\text{ub}_i^X = (1 + \epsilon_X^{\text{mul}})\,\text{lb}_i^X$. For an addition operator $\text{ADD}\langle A_b, A_x, A_y\rangle$, the bounds are given (again with A_b repetitions) as $\text{lb}_i^{\text{FLOPs}} = A_x^iA_y^i$, $\text{lb}_i^{\text{GL}} = 2A_x^iA_y^i$, $\text{lb}_i^{\text{GW}} = A_x^iA_y^i$, with $\text{ub}_i^X = (1+\epsilon_X^{\text{add}})\,\text{lb}_i^X$.

Canonical Template. We adopt the canonical subset defined in Section 3.1, which excludes intermediate attention products $(QK^\top, SV)$. As an illustrative case, we show the *GPT-2 style* design, where each layer uses a two-projections feed-forward module. This yields six multiplications (Q/K/V/O projections and FFN up/down) and two additions (residual connections). Formally, we write

$$\begin{aligned} S^* = \big[&\text{MUL}_Q\langle b, s, d, d\rangle,\ \text{MUL}_K\langle b, s, d, d\rangle,\\ &\text{MUL}_V\langle b, s, d, d\rangle,\ \text{MUL}_O\langle b, s, d, d\rangle,\\ &\text{ADD}_{\text{attn}}\langle b, s, d\rangle,\ \text{MUL}_{\text{ffn\_up}}\langle b, s, d, d_{\text{ffn}}\rangle,\\ &\text{MUL}_{\text{ffn\_down}}\langle b, s, d_{\text{ffn}}, d\rangle,\ \text{ADD}_{\text{ffn}}\langle b, s, d\rangle\big]. \end{aligned}$$

This canonical sequence S^* serves as the semantic baseline against which observed executions are verified.

Attack Model. We define the *attack collection* $\texttt{Attack}$ as the set of structural transformations that can be applied to a canonical sequence S^*. Each element $\texttt{atk} \in \texttt{Attack}^*$ denotes a particular *attack sequence*. Such transformations preserve semantic correctness while altering the syntactic form and the corresponding PMC profile.

Formally, for any $\texttt{atk} \in \texttt{Attack}^*$, the canonical execution sequence S^* may be transformed into a syntactically different but semantically equivalent sequence S':

$$S^* \xrightarrow{\texttt{atk}} S'.$$

A representative example is the *row-split* attack, in which one considers an operation $\text{MUL}_t = \langle A_b, A_x, A_y, B_y \rangle$ that can be further split into $\text{MUL}_{t,1} = \langle A_b, p, A_y, B_y \rangle$ and $\text{MUL}_{t,2} = \langle A_b, A_x - p, A_y, B_y \rangle$, where p specifies the chosen row partition size, and the resulting sequence can be expressed as $S' = [\ldots, \text{OP}_{t-1}, \text{MUL}_{t,1}, \text{MUL}_{t,2}, \text{OP}_{t+1}, \ldots]$.

Conversely, given a manipulated sequence S' and a specific attack sequence $\texttt{atk}$, the operator $\texttt{UndoAttack}(S', \texttt{atk})$ recovers the unique canonical sequence S^* that results from reversing $\texttt{atk}$:

$$S^* = \texttt{UndoAttack}(S', \texttt{atk}).$$

Scope and Assumptions. For tractability, we restrict attention to *split attacks*, i.e., transformations that partition one multiplication into several smaller kernels. We impose two simplifying assumptions:

- Each operation in S^* may undergo at most one attack.
- Each kernel in S' originates from a single operation in S^*.

Other adversarial strategies, such as large-scale kernel substitutions or kernel fusion, are excluded: the former is economically implausible in practice, while the latter is rarely applied to plain matrix multiplications and brings little performance advantage. Additionally, all shape variables are assumed to be positive and aligned with a hardware granularity g, i.e., $v > 0$ and either $v < g$ or $v \equiv 0 \pmod{g}$. Finally, PMC noise is assumed bounded within tolerance $\boldsymbol{\varepsilon}$.

5.2.2 Verification Procedure and SMT Encoding.

The goal is to determine whether any canonical execution S^*, consistent with the observed PMCs and allowed attack transformations, violates the provider's promised specification.

Promise Predicate. Let $M(S)$ denote the model size associated with sequence S, with the definition of M already introduced in Section 3.1. The provider's model size promise is represented by $\text{Promise}(S) := [M(S) \geq M_{\text{promise}}]$ or $[M(S) \leq M_{\text{promise}}]$, depending on whether the specification sets a lower or upper bound.

Decision Rule. Verification amounts to checking whether there exists any observed sequence S' and attack $\texttt{atk}$ such that the reconstructed canonical sequence S^*, obtained via $\texttt{UndoAttack}(S', \texttt{atk})$, both fits the observed PMCs and violates the promise predicate. Formally:

$$\begin{aligned} \exists\, S',\ \exists\, \texttt{atk} \in \texttt{Attack}^* \ \text{ s.t. } \text{PMC}(S') \approx PMC_{\text{obs}}, \\ S^* = \texttt{UndoAttack}(S', \texttt{atk}), \\ \text{Promise}(S^*) = \texttt{False} \ \Rightarrow\ \texttt{Fail}, \\ \text{otherwise} \ \Rightarrow\ \texttt{Pass}. \end{aligned}$$

Here, $\texttt{Fail}$ indicates the solver finds a witness (S', S^*) exposing a promise violation, whereas $\texttt{Pass}$ certifies no such violating canonical sequence exists under the threat model.

Verifier Workflow. To implement this decision rule, the symbolic verifier executes three steps:

(1) *Symbolic instantiation.* Introduce symbolic variables $\{A_b^i, A_x^i, A_y^i, B_y^i\}_{i=0}^{N-1}$ to describe the dimensions of observed kernels K_i in S'. Mapping variables hypothesize correspondences between these kernels and canonical operations in S^*.
(2) *Constraint generation.* Encode two categories of constraints:
   - **Attack consistency**: S^* must be recoverable via $\texttt{UndoAttack}(S', \texttt{atk})$ under the split-attack model.
   - **Promise violation**: at least one reconstructed S^* must satisfy $\text{Promise}(S^*) = \texttt{False}$.
(3) *Solving and interpretation.* Discharge the constraints to an SMT solver. The full constraint set instantiated from these assumptions is provided in Appendix D.

5.2.3 Soundness Guarantees and Error Modes.

Let $V \in \{\text{Pass}, \text{Fail}\}$ be the verifier's verdict and $\text{Promise}(\cdot)$ the provider's claim predicate.

Outcome Taxonomy. The verifier's decision maps to four outcomes against the ground-truth promise.

$$\begin{aligned} &\textbf{TP} \quad \text{V} = \texttt{Fail} \ \&\ \neg\text{Promise}(S^*) \\ &\textbf{TN} \quad \text{V} = \texttt{Pass} \ \&\ \ \text{Promise}(S^*) \\ &\textbf{FP} \quad \text{V} = \texttt{Fail} \ \&\ \ \text{Promise}(S^*) \\ &\textbf{FN} \quad \text{V} = \texttt{Pass} \ \&\ \neg\text{Promise}(S^*) \end{aligned}$$

Soundness Guarantees. Under the following assumptions: (A1) the canonical layer template S^* correctly models the architecture, (A2) the attack family $\texttt{Attack}$ is complete for the threat model, and (A3) PMC noise lies within the specified tolerance $\boldsymbol{\varepsilon}$, the verifier is sound in the following sense:

$$\begin{aligned} \textsc{sat} \quad &\Rightarrow\ \exists\, S^* : \neg\text{Promise}(S^*) \quad \text{(TP)}, \\ \textsc{unsat} \quad &\Rightarrow\ \forall\, S^* : \ \ \text{Promise}(S^*) \quad \text{(TN)}. \end{aligned}$$

If the SMT instance is satisfiable, the solver returns a witness (S', S^*) violating the provider's promise and the verifier refuses; conversely, if unsatisfiable, no such violating sequence exists and the verifier passes.

Error Modes. False positives (**FP**) may occur if the canonical template or noise bound is mis-specified, violating A1 or A3. False negatives (**FN**) arise if the adversary employs a

transformation not covered by the attack family (violating A2), or if the true noise exceeds the assumed bound ϵ. These error modes highlight the importance of accurate modeling assumptions and comprehensive threat coverage.

6 Evaluation

In this section, we first evaluate the feasibility and accuracy of execution trace inference. We then examine the results of model structural consistency verification for monitoring both the lower bound and the upper bound of model scale.

Experimental Setup. We conduct experiments on three NVIDIA GPU architectures with different memory sizes: RTX 4090 (Ada Lovelace Architecture, 24 GB DDR6), RTX 5080 (Blackwell Architecture, 16 GB DDR7), and H100 (Hopper Architecture, 80 GB HBM). The software environment includes CUDA 12.8, PyTorch 2.7.0, and Nsight Compute (V2025.1.1.0 for 4090 and H100, and V2025.2.1.0 for 5080). Verification is performed on a MacBook with an M2 Pro CPU and 32 GB of memory, using the `z3-solver` package (version 4.15.0.0).

We evaluate on three open-sourced model architectures, including GPT–2, LLaMA, and Qwen, but only use their architectural templates with customized settings rather than pretrained weights. Our evaluation spans model sizes from 25M to 10B, and the approach naturally generalizes to even larger models. Across test cases, the hidden size d is swept from 512 to 8192; the number of heads follows $h = d/64$ for GPT–2, while for LLaMA and Qwen it follows $h = d/128$. Unless noted, the feed-forward dimension is $d_{\text{ffn}} = 4d$; we also include model-typical variants (LLaMA: $d = 4096$, $d_{\text{ffn}} = 11008$; Qwen: $d = 4096$, $d_{\text{ffn}} = 22016$). To study batching effects, we additionally run $d = 768$ with $b \in \{1, 2, 4, 8, 16\}$. We use FP32 precision by default. The complete grid of test cases and per-GPU verification outcomes (lower-bound FP/FN) is reported in Table 7 (Appendix E).

6.1 Execution Trace Inference Results

Model Identification (GPT–2 vs LLaMA/Qwen). We first assess whether Wave can distinguish different decoder–only Transformer architectures. GPT–2 follows the *vanilla* design with a fused QKV projection (one GEMM before attention) and a GELU MLP with two matrix multiplications per block (*FFN-2G*). In contrast, Qwen consistently employs a *gated* design: separate Q, K, and V projections (three GEMMs; QKV-split) and a SwiGLU MLP with three matrix multiplications per block (*FFN-3G*). LLaMA also uses SwiGLU (*FFN-3G*), but we observe that its QKV behavior varies: some runs show a fused projection, while others show split Q/K/V. Wave can successfully distinguish all test cases across RTX 4090/5080 and H100. Concretely, as shown in Figure 3, GPT–2 (*QKV-fused + FFN-2G*) has one fused QKV GEMM before attention and two FFN GEMMs per block, while LLaMA (*QKV-fused + FFN-3G*) also shows fused QKV but with three FFN GEMMs.

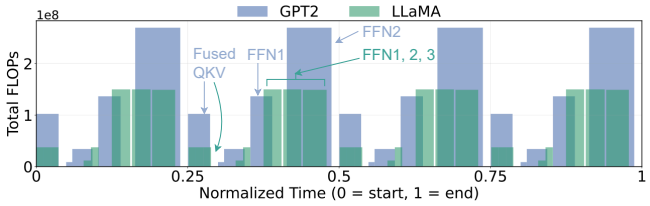


Figure 3. Normalized total FLOPs over normalized time (0=start, 1=end) for representative runs (H100 Case). The repeated layer motif makes patterns stable across traces.

Thus, they are distinguished by FFN count, with signatures stable across layers and independent of kernel names.

Model Parameter Recovery. Table 2 summarizes our parameter recovery accuracy. Several quantities can be read out with negligible overhead: (i) *precision* (dtype) from the dominant scalar-FMA bucket; (ii) *token count* directly from the decoded sequence; and (iii) *number of layers* from periodicity in the kernel stream. Structural parameters are recovered with high accuracy: The hidden size ($\hat{d}/d$) and FFN dimension ($\hat{d_{\text{ffn}}}/d_{\text{ffn}}$) error are 7% and 3% across devices and models on average. We also recover the *model size*. Given the variant identified (QKV split/fused; FFN-2G/3G), the recovered d and d_{ffn}, and the layer count L, we compute parameters per layer while excluding low-order terms such as LayerNorm parameters and FFN biases. The per-layer parameter counts are

$$\hat{P}_{\text{attn}} = 4\hat{d}^2, \qquad \hat{P}_{\text{ffn}} = \lambda\, \hat{d}_{\text{ffn}}\hat{d} \quad (\lambda = 2 \text{ for 2G},\ \lambda = 3 \text{ for 3G}),$$

and total model size as

$$\hat{P} = L\big(\hat{P}_{\text{attn}} + \hat{P}_{\text{ffn}}\big).$$

Note that this excludes the embedding term P_{emb}, which accounts for token embeddings (tied/untied per model). Since P_{attn} and P_{ffn} dominate the total parameter count, accurate recovery of $(\hat{d}, \hat{d_{\text{ffn}}})$ still yields reliable model-size estimates, with an average of 11% error.

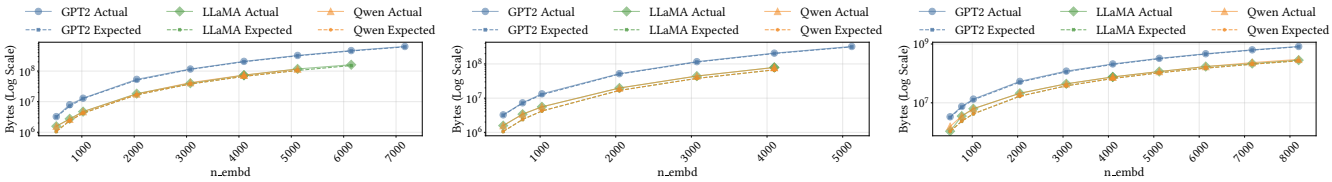
Example: QKV load-byte Observation vs. Theoretical Values. Figure 4 showcases that the measured QKV load-miss bytes closely follow the theoretical curve across devices. Across RTX 4090/5080/H100 and models (GPT–2, LLaMA, Qwen), the measured L1/TEX *load-miss bytes* for QKV (*solid*) closely overlay the *theoretical* curve $(d^2 + b\,s\,d) \cdot D$ (*dashed*), where $s = 1$ in the decoding stage. The small, near-constant offset over d and devices indicates that miss-sector bytes are a stable proxy for off-L1 traffic, validating Wave's inference way to recover $b \times d$ and d.

6.2 Model Structural Consistency Verification

All 44 detailed model settings are listed in Table 7 (Appendix E).

6.2.1 Model Lower Bound ($M(S^*) \geq M_{\text{promise}}$).

We verify structural consistency for the lower bound on GPT–2, LLaMA, and Qwen with hidden dimensions d ranging from 512 to 8192. The number of layers is fixed at $L = 8$, as this



(a) RTX 4090 (24GB GDDR6) **(b)** RTX 5080 (16GB GDDR7) **(c)** H100 (80GB HBM)

Figure 4. QKV load-miss bytes vs. theory on NVIDIA RTX 4090/5080/H100 (solid = measured, dashed = theoretical). The close overlap across d supports the miss-sector model. The missing node occurs because the model does not support the corresponding parameter setting due to insufficient GPU memory.

Metric	RTX 4090			RTX 5080			H100		
	G2	Lla	Qw	G2	Lla	Qw	G2	Lla	Qw
Prec.	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Batch	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
#Layers	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Token Count	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hidden $\hat{d}/d$	1.015	1.064	1.067	1.011	1.130	1.130	1.022	1.080	1.112
FFN $\hat{d_{\text{ffn}}}/d_{\text{ffn}}$	1.016	1.020	1.017	1.023	1.047	1.072	1.018	1.023	0.992
Model size	1.031	1.097	1.098	1.030	1.204	1.226	1.042	1.119	1.135

Table 2. Averaged recovery results across GPUs and models (44 traces). "Yes" indicates successful recovery within tolerance. Abbrev.: G2=GPT–2, Lla=LLaMA, Qw=Qwen.

count can be easily and reliably determined in the inference step and is unrelated to per-layer verification.

Verifier Setup. The verifier is configured with $\varepsilon_{\text{GL}}^{\text{mul}} = 0.5$ and $\varepsilon_{\text{GL}}^{\text{add}} = \varepsilon_{\text{GW}}^{\text{add}} = 0.05$. For the false positive evaluation, the verifier is given a minimum model size of 0.75 times the true per-layer size (*i.e.* $M_{\text{promise}} = 0.75M(S^*)$) and checks whether the collected PMCs could be explained by a smaller model. If so, a false positive occurs. For the false negative evaluation, the verifier is given a minimum model size of 1.25 times the true per-layer size (*i.e.* $M_{\text{promise}} = 1.25M(S^*)$) and checks whether the collected PMCs can be mapped back to the true model setting. If the verifier fails to identify the true setting as a valid solution under this bound, a false negative occurs.

Results. The detailed results are presented in Table 7. No false positives or verification time out are observed on any of the three GPUs traces. For false negatives, the accuracy is 72.7% (4090), 93.2% (5080), and 93.2% (H100) among all 44 test cases. Failure cases can be categorized into two types:

- **Excessive global load.** For some matrix multiplication kernels, the relative deviation of the observed global load from the theoretical value exceeds $\varepsilon_{\text{GL}}^{\text{mul}}$. This maps the allowed d to a range far from the actual value. In contrast, addition kernels remain accurate, mapping d to a range close to the true value. Since these two ranges do not overlap, the verifier finds no valid solution, leading to a false negative. Such cases typically occur with large batch sizes, likely due to repeated loading of the weight matrix. This may be mitigated by incorporating batch size, weight reloading frequency, and hardware characteristics into estimates.
- **Reduced global write.** Some matrix multiplication kernels report smaller global write values than the theoretical expectation. This occurs primarily for $b = 1$ and is likely due to noise or low-level optimizations; further investigation is left for future work.

6.2.2 Model Upper Bound ($M(S^*) \leq M_{\text{promise}}$)).

We verify structural consistency for the upper bound on a self-implemented GPT–2-like model with a single transformer layer, with $d = 1024$ and $d_{\text{ffn}} = 4096$, running on an RTX 4090. The model consists of six splitable linear layers for the Q, K, V, O, FFN-Up, and FFN-Down projections. Each splitable linear layer is instantiated with a specific split configuration that is fixed across inference. A split divides one dimension of a layer into two parts, with the resulting part chosen randomly while respecting a granularity constraint of $g = 256$. The split configuration for each layer is randomly selected from its available split ways. The test cases include: one no-split case, three all-split cases, and ten randomly generated cases with between one and five split linear layers. For false positive and false negative evaluation, the ratios of the promised maximum model size M_{promise} to the actual model size $M(S^*)$ are set to 0.9 and 1.1, respectively.

Results. Detailed results are presented in Table 8. Verification completes in no more than 1 min. No false positives or false negatives are observed, as the verifier considers all possible split attacks and kernel merging strategies, demonstrating robustness against split-based attacks.

7 Discussion

7.1 Overhead

A key limitation of Wave prototype is the significant overhead incurred when collecting all required PMCs. To quantify this, we evaluate sample model configurations and measure the relative slowdown compared to running inference without Nsight Compute. The results are shown in Table 3. Even in the best case on H100, the overhead of collecting Wave metrics exceeds 1196%. To better understand the source of this overhead, we also measure the cost of collecting only a single hardware-level metric (execution time, `gpu__time_duration.sum`). The overhead in this case ranges from 52%

GPU	Metric Type	Min. (%)	Max. (%)	Avg. (%)
RTX 4090	HW metric	993	1333	1149
	Wave metrics	4094	5288	4689
RTX 5080	HW metric	95	121	110
	Wave metrics	2500	3443	3007
H100	HW metric	52	80	68
	Wave metrics	1196	1585	1340

Table 3. Profiling PMC collection overhead relative to baseline inference, reported as percentage. HW refers to monitoring a single hardware time metric: `gpu__time_duration`.

to 1333% across the three GPUs, already far higher than expected. Unlike CPU PMC, which records the PMC counter on-the-fly with under 1% overhead [9, 22, 40], these results suggest that the excessive overhead is primarily due to the current GPU PMC design, including both hardware design and the way Nsight Compute collects results. Importantly, we intentionally select global, rather than per-thread PMC metrics (e.g., $X \in$ FLOPs, GlobalLoad, GlobalWrite, 9 metrics in total), which are speculatively lower-cost to expose. Moreover, the trend of decreasing overhead from 4090 to H100 suggests that newer/powerful GPUs are improving. In deployment, instead of Nsight's on-demand profiling, a periodic reporting service with predetermined events (e.g., NVIDIA's Data Center GPU Manager (DCGM)-like) is shown to be able to introduce real-time monitoring and introduce only minimal steady-state overhead.

7.2 Limited Scope and Future Outlook

In this subsection, we outline aspects that are not yet covered by Wave and discuss possible extensions.

Not an End-to-end System. Wave is a proof-of-concept prototype that demonstrates the feasibility of using hardware observations to fingerprint and reason about model execution behavior, rather than a deployable end-to-end system. However, under today's GPU hardware and cloud deployment models, there is no suitable interface that allows end users to directly access such low-level hardware signals in a trustworthy way. In practice, PMC access is privileged and restricted in multi-tenant environments, meaning that Wave currently serves as a proof-of-concept rather than an on-the-fly, end-to-end system.

Bridging this gap requires several system-level extensions: (i) an low-overhead interface for rapidly obtaining the required PMC observations. This requires architectural or firmware support from GPU vendors, either CPU-style PMC access or appending necessary counters via existing telemetry frameworks such as NVIDIA DCGM, which periodically reports hardware metrics. (ii) A trustworthy, attested channel from GPU hardware to a trusted verifier that authenticates the collected hardware observations and binds them to a specific user query. This could be realized via a TEE-based design (e.g., a CPU TEE with privileged access to PMC traces that verifies and signs a report tied to the user input) or an in-GPU verifier that directly signs a report. Compared to current GPU TEEs that attest only the initial hardware state, such information together with Wave could provide model-aware runtime evidence.

Sensitivity to System Optimizations. Wave's analytical models are sensitive to framework-specific implementations and low-level system optimizations. Our current modeling does not account for techniques such as quantization, kernel fusion, or operator reordering, which are commonly employed by modern inference engines such as vLLM [20] and SGLang [46]. Extending Wave to accurately handle different frameworks and optimization strategies would require additional framework-aware modeling as well as validation and testing under each optimization configuration.

Lack of Multi-GPU and Advanced Multi-Tenant Support. The current framework is restricted to single-GPU inference. In contrast, production deployments commonly adopt multi-GPU configurations with diverse parallelism strategies, which introduce cross-device communication and synchronization that are not captured by Wave 's current model. Supporting these scenarios would require extending our PMC analysis beyond computation to include inter-GPU communication and device-to-device transfer events. Regarding multi-tenancy, we currently consider only static batching and do not address advanced scheduling optimizations such as continuous batching [43].

Exclusion of Sparse Architectures. We consider only dense models and do not address Mixture-of-Experts (MoE) architectures [11]. Since only a subset of experts is active during inference, Wave can observe and verify only the executed computational effort, but cannot infer the total number of experts or the full model size. Additionally, while Chain-of-Thought (CoT) reasoning [41] is not explicitly evaluated in this work, Wave is expected to remain compatible, as the additional hidden reasoning tokens would still manifest as observable PMC events.

Extension to Prefill Phase. In the current Wave framework, we focus primarily on the decoding phase, as it constitutes the majority of the generation process and benefits from a fixed sequence length of 1, which significantly simplifies the verification process. However, extending Wave to the prefill stage is feasible. By introducing a variable sequence length s, we can formulate additional constraints analogous to those used for decoding. Furthermore, since users are able to approximate or even precisely determine the prompt length using a tokenizer, this information can serve as an additional constraint (e.g., $s \geq$ prompt length). While these supplementary constraints can potentially yield tighter bounds on the estimated model size, they necessitate careful calibration of the upper and lower bound thresholds to prevent excessive false positives.

Despite these limitations, we hope Wave can act as a baseline design, with the expectation that it can be extended

to the aforementioned scenarios or applied in distributed environments involving multi-party evidence exchange.

8 Related Work

Wave offers a hardware-based privacy-preserving monitoring solution with low overhead potential. Because it is currently a prototype, we do not make direct performance comparisons with existing monitoring systems. For context, we summarize other mainstream monitoring approaches below.

Black-box auditing and API-level analysis. Black-box audits probe models only through their I/O. Tramèr et al. [37] formalize model extraction from prediction APIs; Knockoff Nets [27] train surrogates to imitate services. For LLMs, timing analysis can recover output-token counts [44]; CoIn [32] audits invisible reasoning tokens. SVIP [34] verifies model identity via challenge–response protocols over intermediate states. However, it necessitates a per-model proxy feature extractor and assumes the provider can expose processed hidden-state representations at inference time. In contrast, Wave relies solely on PMC signals and is model-agnostic, making it privacy-preserving at the root and applicable even when model internals are unavailable.

Zero-knowledge proofs and verifiable ML. ZK systems provide strong, model-agnostic correctness guarantees but at substantial overheads. zkLLM [33] introduces gadgets for attention/lookups; ZKML [5] compiles ML inference to ZK circuits; ZKTorch [4] amortizes proofs over PyTorch graphs; Kang et al. [16] and Hao et al. [13] address scalability for common nonlinearities.

Trusted execution, attestation, and resource accounting. TEEs and verifiable metering offer orthogonal evidence for legitimacy. VeriCount [35] and T-Counter [10] realize CPU usage metering; S-FaaS [1] and AccTEE [12] bring TEE-backed accountability to serverless/Wasm settings. For GPU, SAGE [15] explores software-based attestation for GPU execution. NVIDIA GPU TEE [24] only offers protected GPU execution, without direct interfaces for runtime oversight.

Watermarking and provenance. Statistical watermarks for text [17, 18] and provably robust constructions [45] provide content-side provenance, while transparency logs [21] offer append-only registries for public auditing. However, while useful for general tracking, software-level watermarks remain vulnerable in cloud scenarios, where a malicious provider with full control over the generation pipeline can forge or strip these signals.

PMC and Hardware side-channel information. PMCs expose fine-grained execution signals useful for inference of program semantics. On CPUs, counters can classify DNN layer types [19]. On GPUs, Naghibijouybari et al. [23] demonstrate cross-application leakage via timing, memory APIs, and counters. DeepSniffer [14] reconstructs DNN architectures from GPU memory traffic and kernel patterns.

9 Conclusion

This paper introduces a hardware-based monitoring framework Wave, which utilizes content-agnostic GPU performance counter signals to enable privacy-preserving oversight of LLM inference. Infer key architectural parameters from runtime traces and employs an SMT solver to ensure execution remains consistent with the expected configurations.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their insightful comments, as well as FlexHEGs for their support and helpful discussions. This research received support from the Future of Life Institute and the Blake Borgeson Foundation.

References

[1] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 185–199.

[2] AMD ROCm Team. 2025. ROCm ROCProfiler. https://rocm.docs.amd.com/projects/rocprofiler/en/latest/ Accessed: 2025-08-21.

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[4] Bing-Jyue Chen, Lilia Tang, and Daniel Kang. 2025. ZKTorch: Compiling ML Inference to Zero-Knowledge Proofs via Parallel Proof Accumulation. arXiv:2507.07031 [cs.CR] https://arxiv.org/abs/2507.07031

[5] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. 2024. ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 560–574. https://doi.org/10.1145/3627703.3650088

[6] NVIDIA Corporation. 2025. CUDA Profiling Tools Interface (CUPTI). https://developer.nvidia.com/cupti Accessed: 2025-08-21.

[7] NVIDIA Corporation. 2025. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute Accessed: 2025-08-21.

[8] NVIDIA Corporation. 2025. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems Accessed: 2025-08-21.

[9] Patrick Diehl, Dominic Marcello, Parsa Amini, Hartmut Kaiser, Sagiv Shiber, Geoffrey C Clayton, Juhan Frank, Gregor Daiß, Dirk Pflüger, David Eder, et al. 2021. Performance measurements within asynchronous task-based runtime systems: A double white dwarf merger as an application. *Computing in Science & Engineering* 23, 3 (2021), 73–81.

[10] Chuntao Dong, Qingni Shen, Xuhua Ding, Daoqing Yu, Wu Luo, Pengfei Wu, and Zhonghai Wu. 2022. T-counter: Trustworthy and efficient CPU resource measurement using SGX in the cloud. *IEEE Transactions on Dependable and Secure Computing* 20, 1 (2022), 867–885.

[11] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[12] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/3361525.3361541

[13] Meng Hao, Hanxiao Chen, Hongwei Li, Chenkai Weng, Yuan Zhang, Haomiao Yang, and Tianwei Zhang. 2024. Scalable zero-knowledge proofs for non-linear functions in machine learning *(SEC '24)*. USENIX Association, USA, Article 214, 18 pages.

[14] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. 2020. DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 385–399. https://doi.org/10.1145/3373376.3378460

[15] Andrei Ivanov, Benjamin Rothenberger, Arnaud Dethise, Marco Canini, Torsten Hoefler, and Adrian Perrig. 2022. SAGE: Software-based Attestation for GPU Execution. https://doi.org/10.48550/ARXIV.2209.03125

[16] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. 2022. Scaling up Trustless DNN Inference with Zero-Knowledge Proofs. arXiv:2210.08674 [cs.CR] https://arxiv.org/abs/2210.08674

[17] John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2024. A Watermark for Large Language Models. arXiv:2301.10226 [cs.LG] https://arxiv.org/abs/2301.10226

[18] John Kirchenbauer, Jonas Geiping, Yuxin Wen, Manli Shu, Khalid Saifullah, Kezhi Kong, Kasun Fernando, Aniruddha Saha, Micah Goldblum, and Tom Goldstein. 2024. On the Reliability of Watermarks for Large Language Models. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=DEJIDCmWOz

[19] Bhargav Achary Dandpati Kumar, Sai Chandra Teja R, Sparsh Mittal, Biswabandan Panda, and C. Krishna Mohan. 2021. Inferring DNN layer-types through a Hardware Performance Counters based Side Channel Attack *(AIMLSystems '21)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3486001.3486224

[20] 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*. 611–626.

[21] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962. https://doi.org/10.17487/RFC6962

[22] Sergio Mazzola, Gabriele Ara, Thomas Benz, Björn Forsberg, Tommaso Cucinotta, and Luca Benini. 2025. Data-driven power modeling and monitoring via hardware performance counter tracking. *Journal of Systems Architecture* (2025), 103504.

[23] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks are Practical *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2139–2153. https://doi.org/10.1145/3243734.3243831

[24] NVIDIA. 2023. Confidential Compute on NVIDIA Hopper H100. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf.

[25] OpenAI Forum. 2024. GPT4-Turbo more "stupid/lazy" – it's not a GPT4. OpenAI Community Forum post. https://community.openai.com/t/gpt4-turbo-more-stupid-lazy-its-not-a-gpt4/608008

[26] OpenAI Forum. 2024. OpenAI did made GPT3.5 more stupid? OpenAI Community Forum post. https://community.openai.com/t/openai-did-made-gpt3-5-more-stupid/262979

[27] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2018. Knockoff Nets: Stealing Functionality of Black-Box Models. arXiv:1812.02766 [cs.CV] https://arxiv.org/abs/1812.02766

[28] Ofir Press, Noah A. Smith, and Mike Lewis. 2022. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. arXiv:2108.12409 [cs.CL] https://arxiv.org/abs/2108.12409

[29] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[30] Alex Saltanov. 2024. OpenAI Keeps Dumbing Down ChatGPT. AI Mind (Medium publication) article. https://pub.aimind.so/openai-keeps-dumbing-down-chatgpt-6a6e4a173237

[31] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1075–1092.

[32] Guoheng Sun, Ziyao Wang, Bowei Tian, Meng Liu, Zheyu Shen, Shwai He, Yexiao He, Wanghao Ye, Yiting Wang, and Ang Li. 2025. CoIn: Counting the Invisible Reasoning Tokens in Commercial Opaque LLM APIs. arXiv:2505.13778 [cs.AI] https://arxiv.org/abs/2505.13778

[33] Haochen Sun, Jason Li, and Hongyang Zhang. 2024. zkllm: Zero knowledge proofs for large language models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 4405–4419.

[34] Yifan Sun, Yuhang Li, Yue Zhang, Yuchen Jin, and Huan Zhang. 2025. SVIP: Towards Verifiable Inference of Open-source Large Language Models. arXiv:2410.22307 [cs.LG] https://arxiv.org/abs/2410.22307

[35] Shruti Tople, Soyeon Park, Min Suk Kang, and Prateek Saxena. 2018. VeriCount: Verifiable resource accounting using hardware and software isolation. In *Applied Cryptography And Network Security: 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings 16*. Springer, 657–677.

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

[37] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. arXiv:1609.02943 [cs.CR] https://arxiv.org/abs/1609.02943

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

[39] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[40] Vincent M Weaver. 2015. Self-monitoring overhead of the Linux perf_event performance counter interface. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 102–111.

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[42] Yuchen Xia, Jiho Kim, Yuhan Chen, Haojie Ye, Souvik Kundu, Cong Hao, and Nishil Talati. 2024. Understanding the Performance and Estimating the Cost of LLM Fine-Tuning. arXiv:2408.04693 [cs.CL] https://arxiv.org/abs/2408.04693

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

[44] 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

[45] Xuandong Zhao, Prabhanjan Vijendra Ananth, Lei Li, and Yu-Xiang Wang. 2024. Provable Robust Watermarking for AI-Generated Text. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=SsmT8aO45L

[46] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code for Wave, comprising the PMC data preprocessing analyzer and the SMT-based verifier. The artifact allows users to reproduce the key experimental results presented in the paper:

1. The motivating example illustrating the correlation between model size and PMC signatures (Figure 2).
2. The example of using PMC to recover some parameters (Figure 4).
3. The system's capability to verify model lower bounds (Table 7).
4. The system's capability to verify model upper bounds (Table 8).
5. The runtime overhead evaluation (Table 3).

A.2 Artifact check-list (meta-information)

- **Algorithm:** PMC-based analytical modeling and verification.
- **Model:** GPT–2, LLaMA, QWen, custom split-layer Transformer.
- **Run-time environment:** Ubuntu, CUDA 12.8, PyTorch 2.7.0, Nsight Compute 2025.1+.
- **Hardware:** Nvidia GPUs (Tested on 4090, 5080, H100).
- **Execution:** Bash and Python scripts.
- **Output:** Verification results.
- **Experiments:** Capability testing (lower/upper bounds) and overhead measurement.
- **Workflow preparation time:** ≈ 30 minutes.
- **Experiment completion time:** ≈ 6 hours including metrics collection.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.

A.3 Description

A.3.1 How to access. Wave is available at https://github.com/sept-usc/Wave.

A.3.2 Hardware dependencies. The artifact has been explicitly tested on Nvidia RTX 4090, RTX 5080, and H100 GPUs.

A.3.3 Software dependencies. Requires CUDA 12.8, PyTorch 2.7.0, Nsight Compute 2025.1+.

A.3.4 Models. The experiments utilize customized configurations based on GPT-2, LLaMA, and Qwen architectures, as well as a customized split-layer Transformer.

A.4 Installation

```
git clone git@github.com:sept-usc/Wave.git
bash scripts/setup_environment.sh
```

A.5 Experiment workflow

Please refer to the README for the full detailed workflow.

A.6 Evaluation and expected results

The recovered parameters should be close to theoretical values. The verification results should match Table 7 and Table 8. The motivating example results should match Figure 2, and the overhead results should match Table 3.

B Key Model Parameters Description

Table 4 summarizes the key model parameters used throughout this paper. Each parameter is defined with its symbol, description, and expected impact.

C Execution Trace Inference: Metrics and Operational Details

C.1 Symbols and basic constants

We use d (hidden size), d_{ffn} (FFN intermediate), b (batch), s (sequence length), L (number of layers). Let s_{B} be bytes per L1/TEX sector (we set s_{B}=32 B in our runs). Table 5 maps paper symbols to Nsight metrics. Names may differ by toolkit version.

C.2 From raw counters to features

Compute feature $F(k)$. We weight scalar ops as FMA × 2 + ADD + MUL across dtypes. For each kernel k:

$$\begin{aligned} F(k) = {} & 2\,\text{HFMA}(k) + \text{HADD}(k) + \text{HMUL}(k) \\ & + 2\,\text{FFMA}(k) + \text{FADD}(k) + \text{FMUL}(k) \\ & + 2\,\text{DFMA}(k) + \text{DADD}(k) + \text{DMUL}(k). \end{aligned}$$

The data width $D \in \{2, 4, 8\}$ B is determined by whichever FMA counter among $\{\text{HFMA}, \text{FFMA}, \text{DFMA}\}$ dominates. Tensor-path counters may be logged as a side check but are not required:

$$D = \begin{cases} 2\text{ B}, & \text{HFMA}(k) \text{ is maximal}, \\ 4\text{ B}, & \text{FFMA}(k) \text{ is maximal}, \\ 8\text{ B}, & \text{otherwise}. \end{cases}$$

Memory features. For kernel k, we need to convert sectors to bytes and use instruction counts to form $r_{\text{sh}}(k)$:

$$B_{\text{ld}}(k) = s_{\text{B}} \cdot SC_{\text{ld}}(k), \quad B_{\text{st}}(k) = s_{\text{B}} \cdot SC_{\text{st}}(k),$$
$$B_{\text{tot}}(k) = B_{\text{ld}}(k) + B_{\text{st}}(k).$$

$$r_{\text{sh}}(k) = \frac{I^{\text{SH}}_{\text{ld}}(k) + I^{\text{SH}}_{\text{st}}(k)}{I^{\text{SH}}_{\text{ld}}(k) + I^{\text{SH}}_{\text{st}}(k) + I^{\text{GL}}_{\text{ld}}(k) + I^{\text{GL}}_{\text{st}}(k)}.$$

C.3 Rules and role assignment

Let $c > 0$, $p \in (0, 1)$, and $s > 0$ be fixed (see Table 6). For each layer, define F_{total} as the set of kernel FLOPs. We use the following cut-off to identify matrix-heavy kernels:

$$\tau_F = \max\left(\frac{\overline{F_{\text{total}}}}{c},\ \text{quantile}(F_{\text{total}}, p)\right).$$

Parameter (Symbol)	Brief Description	Effect / Impact
Number of Layers (L)	Total number of Transformer blocks.	Increases representational depth; scales compute and memory roughly linearly with L.
Hidden Size (d)	Dimensionality of the model state.	Larger d raises capacity while quadratically increasing per-layer GEMM cost and activation memory.
Number of Heads (h)	Attention heads per layer (per-head dimension = d/h).	Improves expressivity and routing; changes kernel shapes and constants but not the asymptotic scaling for fixed d.
Feed-forward Dimension (d_{ffn})	Width of the FFN intermediate projection.	Higher d_{ffn} increases FFN compute and parameter count linearly in d_{ffn}.
Batch Size (b)	Sequences processed concurrently (decoding typically $b = 1$).	Increases throughput at the expense of memory footprint; affects bandwidth pressure.
Sequence Length (s)	Tokens per input sequence.	Longer s increases prefill cost and KV-cache memory; per-token decoding cost is insensitive to s.
Token Count (T)	Total number of generated output tokens in a run.	Acts as a time/compute multiplier; total runtime and work scale approximately linearly with T.
Data Width (D)	Bytes per element (numeric precision), e.g., FP16/BF16/FP32.	Lower D reduces bandwidth and storage and can improve throughput, with potential numerical-fidelity trade-offs.
Model Size (M)	Defined as $M = L \cdot (4d^2 + \lambda dd_{\text{ffn}})$, where λ is the number of FFN projections	Serves as a proxy for capacity and cost; a discrepancy $M(S^*) < M_{\text{promise}}$ signals a violation under our verification setting.

Table 4. Core parameters used in our analysis and their effects.

Symbol (Meaning)	Nsight Compute Metric
HFMA$(\cdot)$ (FP16/BF16 scalar FMA inst.)	`smsp__sass_thread_inst_executed_op_hfma_pred_on.sum`
FFMA$(\cdot)$ (FP32 scalar FMA inst.)	`smsp__sass_thread_inst_executed_op_ffma_pred_on.sum`
DFMA$(\cdot)$ (FP64 scalar FMA inst.)	`smsp__sass_thread_inst_executed_op_dfma_pred_on.sum`
HADD, HMUL, FADD, FMUL, DADD, DMUL	`smsp__sass_thread_inst_executed_op_{h,f,d}{add,mul}_pred_on.sum`
Tensor-path (optional)	`sm__ops_path_tensor_src_*.sum`
$SC_{\text{ld}}(\cdot)$ (global load lookup-miss sectors)	`l1tex__t_sectors_pipe_lsu_mem_global_op_ld_lookup_miss.sum`
$SC_{\text{st}}(\cdot)$ (global store lookup-miss sectors)	`l1tex__t_sectors_pipe_lsu_mem_global_op_st_lookup_miss.sum`
$I_{\text{ld}}^{\text{GL}}(\cdot)$ (global load inst. count)	`smsp__sass_inst_executed_op_global_ld.sum`
$I_{\text{st}}^{\text{GL}}(\cdot)$ (global store inst. count)	`smsp__sass_inst_executed_op_global_st.sum`
$I_{\text{ld}}^{\text{SH}}(\cdot)$ (shared load inst. count)	`smsp__sass_inst_executed_op_shared_ld.sum`
$I_{\text{st}}^{\text{SH}}(\cdot)$ (shared store inst. count)	`smsp__sass_inst_executed_op_shared_st.sum`

Table 5. Symbol ↔ Nsight Compute metric (RTX4090/RTX5080/H100). One sector = s_{B} bytes (32 B in our runs).

A kernel is *matmul-like* if $F(k) \geq \tau_F$. Among these, a kernel is *attn-like* if it also satisfies $r_{\text{sh}}(k) \geq s$. Kernels with nonzero $F(k)$ but below τ_F are labeled *add-like*.

Role assignment follows the fixed template

$$S^* = [\text{QKV, Attn, O, Add, FFN, Add}],$$

with multiplicities $|\text{QKV}| \in \{1, 3\}$ and $|\text{FFN}| \in \{2, 3\}$. Within each *matmul-like* group, specific roles (Q/K/V/O, FFN blocks) are determined by comparing off-L1 global traffic $B_{\text{tot}}(k)$ and FLOPs $F(k)$.

C.4 Periodicity detection (Step 1)

For each kernel at sequential index t (the t-th kernel in execution order), form a 3-D signature

$$x_t = [F(k),\ r_{\text{sh}}(k),\ B_{\text{tot}}(k)],$$

Cosine self-correlation. Define

$$C(\ell) = \frac{1}{N-\ell} \sum_{t=1}^{N-\ell} \frac{x_t \cdot x_{t+\ell}}{\|x_t\|\,\|x_{t+\ell}\|}.$$

Use a small shift allowance to absorb one or two insertions:

$$C_{\pm 1}(\ell) = \frac{1}{N-\ell} \sum_{t=1}^{N-\ell} \max_{\Delta \in \{-1,0,1\}} \frac{x_t \cdot x_{t+\ell+\Delta}}{\|x_t\| \, \|x_{t+\ell+\Delta}\|}.$$

We search ℓ in a plausible band (e.g., $\ell \geq 5$); take $\ell^\star = \arg\max C_{\pm 1}(\ell)$. A clear peak at $\ell^\star$ (and its multiples) indicates a period.

Phase and segmentation. Pick the start $r^\star$ whose positions $r, r+\ell^\star, \ldots$ have the highest average similarity. A clear peak at $\ell^\star$ means that the sequence aligns well with itself when shifted by $\ell^\star$, so statistical similarity is maximized. Then march forward by $\ell^\star$, snapping each boundary to the best nearby index in a tiny ±1 window. This absorbs small insertions or deletions while keeping the global beat.

Two nested periods. We observe two levels of repetition in the feature stream. The *large* period counts kernels per token; detecting and counting these outer cycles recovers the total number of tokens, and further separates the prefill cycle from the subsequent decode cycles. Inside each large period lies a *small* period that repeats once per layer. Counting these inner periods within each large cycle recovers the layer count.

C.5 In-layer role assignment (Step 2)

Candidates. Within a detected layer, a candidate assignment maps kernels to roles as follows:

- Only *matmul-like* kernels are assigned to Q, K, V, O, and FFN.
- The unique *attn-like* kernel is assigned to Attention.
- Assignments must respect the canonical order

$$[\text{QKV} \rightarrow \text{Attention} \rightarrow \text{O} \rightarrow \text{FFN}].$$

We consider candidate patterns as the cross product

$$[\text{QKV split, QKV fused}] \times [\text{FFN 2G, FFN 3G}],$$

yielding four possibilities per layer.

Ratio targets. Within a Transformer layer, we define canonical projection shapes as the basis for ratio targets. Each self-attention projection (Q, K, V, O) is a square matrix multiply of shape $\mathbb{R}^{d \times d}$, which we treat as one unit cost. If Q, K, V are fused into a single kernel, the effective computation is three stacked $d \times d$ projections, i.e., 3 units. For the feed-forward network (FFN) (for instance in GPT2), each GEMM has shape

$$\mathbb{R}^{d \times d_{\text{ffn}}}, \qquad \mathbb{R}^{d_{\text{ffn}} \times d},$$

so relative to a $d \times d$ baseline, one FFN GEMM costs $r = d_{\text{ffn}}/d$ units.

With these definitions, we normalize by output projection and set ratio targets that apply to both load bytes and FLOPs.

Score. For each candidate assignment, we compute a linear score blending three terms:

1. *Load–byte ratio term:* role-to-O-projection ratios of B_{ld} compared against the targets $\{1, 3, r\}$ (for Q/K/V/O, fused QKV, and FFN respectively).
2. *FLOPs ratio term:* same targets but using $F(k)$ ratios.
3. *Coverage term:* the fraction of a layer's total $F(k)$ explained by the assigned roles.

The three terms are averaged with fixed weights; the top-scoring candidate yields the role assignment and a confidence value in $[0, 1]$.

C.6 Model parameter recovery (Step 3)

(A) Readily recoverable.

- *Data Width D:* The dominant scalar FMA bucket among {HFMA, FFMA, DFMA} reveals FP16/BF16, FP32, or FP64. This directly fixes $D \in \{2, 4, 8\}$ bytes.
- *Number of layers:* The period $\ell^\star$ detected in Step 1 gives the layer count L.
- *Token count:* In prefill, equal to the number of detected periods; in decode, $s = 1$. This matches the model's actual token generation, so users can spot mismatches with the received reply and detect token-inflation attacks.

(B) Dominant structural parameters. We recover b, d, and d_{ffn} from load/store *lookup-miss sectors*. The order is: solve $bd \rightarrow d \rightarrow b \rightarrow d_{\text{ffn}}$.

(B1) bd from projection stores. Take the Q projection as an example. For each token it produces a d-dimensional vector, so the total number of stored elements is

$$G = b\,s\,d.$$

Note the in the decoding stage, $s = 1$.

On hardware, the measured write traffic is $B_{\text{st}}(\,)(\text{Q})$ sectors of size s_{B} bytes. Dividing by the data width D converts this into elements, and dividing again by s isolates the batch–hidden product. In the decoding stage, $s = 1$, so this division simply cancels out if needed.

$$b\,d \approx \frac{s_{\text{B}}\, B_{\text{st}}((\text{k}_{\text{Q}}))}{s\,D}.$$

In practice we average the Q, K, V estimates if all are available; otherwise a single projection suffices.

(B2) d from projection loads. The Q projection multiplies an input matrix of size $(bs) \times d$ with a weight matrix of $d \times d$. Here $s = 1$ in the decoding stage. Hence the number of elements read is

$$P = (bs)d + d^2 \approx G + d^2,$$

where $G = bd$ is obtained from (B1). On hardware, the load traffic is measured as

$$P \approx \frac{s_{\text{B}}\, B_{\text{ld}}((\text{k}_{\text{Q}}))}{D}.$$

Subtracting G leaves roughly d^2, so

$$d \approx \sqrt{P - G}.$$

We prefer output projection loads when available, since they provide the same relation without requiring multiple projections, and we round d to the nearest architectural multiple.

Table 6. Constants used across runs. Values marked "set per study" are reported with experiments.

Symbol	Value (Note)
s_B	32 B (RTX4090/RTX5080/H100)
c	50 (set per study)
p	0.8 (set per study)
s	0.7 (set per study)
ϵ, Λ	taper for ratio scoring (set per study)
Weights	score weights for bytes/FLOPs/coverage (set per study)
r	4 (set per study)

(B3) Recovering b. After obtaining d, the batch size is given by

$$b \approx \frac{G}{d}.$$

As a sanity check, we compare this against elementwise add kernels, which also operate on bd elements, i.e., $F(k_{\text{elementwise\_add}}) \approx bd$.

(B4) d_{ffn} from FFN loads. With d fixed, the ratio $r = d_{\text{ffn}}/d$ is inferred from FFN up-projection loads:

$$\hat{r} \approx \frac{s_B\, B_{\text{ld}}((k_{\text{FFN\_up}}))}{d^2\, D} - \frac{b\, s}{d}, \qquad d_{\text{ffn}} = \text{round}(\hat{r}\, d).$$

In 3G-FFN, combine the two up-projections by median.

(C) Model size. With L layers, the per-layer parameter count is

$$P_{\text{layer}} \approx 4\, d^2 + \lambda\, d\, d_{\text{ffn}}, \qquad \lambda = \begin{cases} 2 & \text{FFN-2G,} \\ 3 & \text{FFN-3G.} \end{cases}$$

Thus,

$$M \approx L\,(4\, d^2 + \lambda\, d\, d_{\text{ffn}})$$

and the weight memory is $M \times D$ bytes.

C.7 Reliability tests and exclusions

Skip layers with near-zero miss volumes, strong disagreement between (B1), (B2), and (B3), or insufficient coverage of attention/projection kernels. Across layers, we also check for consistency:

- Cross-layer sanity check: if one layer's estimate deviates strongly from the vast majority of layers in the same mode, it is flagged as unreliable and excluded.
- Within the decoding stage, the recovered d should be relatively stable.

C.8 Fixed constants

The constants used across runs are listed in Table 6.

D SMT Constraint Set

We instantiate the verification instance as an SMT problem whose constraints are abstracted from the assumptions and the modeling setup in Section 5.1.

The solver checks whether the conjunction of the six constraints is satisfiable. If SAT, it yields a witness (S', S^*) that is attack-consistent and violates the promise; otherwise the instance is UNSAT.

D.1 Kernel–operation mapping

Each observed kernel is assigned to some canonical operation index:

$$0 \le k2o_i < O.$$

D.2 Operation segments

Each canonical operation corresponds to a contiguous block in S':

$$\{\, i \mid k2o_i = k \,\} = [start_k,\, end_k], \qquad start_{k+1} > end_k.$$

D.3 PMC bounds

For every kernel i and metric $X \in \{\text{FLOPs}, \text{GL}, \text{GW}\}$, the observation must satisfy

$$\text{lb}_i^X \le X_i \le \text{ub}_i^X,$$

where $\text{lb}_i^{\text{FLOPs}} = 2A_x^i A_y^i B_y^i$, $\text{lb}_i^{\text{GL}} = A_x^i A_y^i + A_y^i B_y^i$, $\text{lb}_i^{\text{GW}} = A_x^i B_y^i$, and $\text{ub}_i^X = (1+\epsilon_X)\,\text{lb}_i^X$ (definitions in Section 5.2.1).

D.4 Kernel merging (valid split pattern)

A segment of L_k kernels must match some admissible split pattern:

$$\bigvee_{p \in P_{L_k}} \text{Pattern}_p\big(A_b^{i:i+L_k-1}, A_x^{i:i+L_k-1}, A_y^{i:i+L_k-1}, B_y^{i:i+L_k-1}\big).$$

D.5 Shape and variable validity (granularity)

With granularity g for feasibility/efficiency, all relevant variables $v \in \{A_x^i, A_y^i, B_y^i, d, d_{\text{ffn}}, \dots\}$ satisfy

$$v > 0, \qquad v < g \;\vee\; v \equiv 0 \pmod{g}.$$

D.6 Model-size promise

$$4d^2 + 2d\, d_{\text{ffn}} \ge M_{\text{promise}}.$$

E Verification Results

Table 7 reports the lower-bound verification results across different models and GPUs. Table 8 summarizes the corresponding upper-bound verification results for comparison.

Table 7. Detailed configurations of GPT–2, LLaMA, and Qwen models, along with verification results. The verifier monitors *lower bounds* on both false positives and false negatives across different GPUs (RTX 4090, RTX 5080, H100). A checkmark (✓) indicates success, a cross (✗) indicates failure, and a dash (-) indicates that the model cannot be executed on the given GPU.

Model	d	d_{ffn}	h	b	False Positive			False Negative		
					RTX 4090	RTX 5080	H100	RTX 4090	RTX 5080	H100
GPT–2	512	2048	8	4	✓	✓	✓	✓	✓	✓
GPT–2	768	3072	12	1	✓	✓	✓	✗	✗	✓
GPT–2	768	3072	12	2	✓	✓	✓	✓	✓	✓
GPT–2	768	3072	12	4	✓	✓	✓	✓	✓	✓
GPT–2	768	3072	12	8	✓	✓	✓	✓	✗	✗
GPT–2	768	3072	12	16	✓	✓	✓	✓	✗	✗
GPT–2	1024	4096	16	4	✓	✓	✓	✓	✓	✓
GPT–2	2048	8192	32	4	✓	✓	✓	✓	✓	✓
GPT–2	3072	12288	48	4	✓	✓	✓	✓	✓	✓
GPT–2	4096	16384	64	4	✓	✓	✓	✓	✓	✓
GPT–2	5120	20480	80	4	✓	✓	✓	✓	✓	✓
GPT–2	6144	24576	96	4	✓	-	✓	✓	-	✓
GPT–2	7168	28672	112	4	✓	-	✓	✓	-	✓
GPT–2	8192	32768	128	4	-	-	✓	-	-	✓
LLaMA	512	2048	4	4	✓	✓	✓	✓	✓	✓
LLaMA	768	3072	6	1	✓	✓	✓	✗	✗	✓
LLaMA	768	3072	6	2	✓	✓	✓	✓	✓	✓
LLaMA	768	3072	6	4	✓	✓	✓	✓	✓	✓
LLaMA	768	3072	6	8	✓	✓	✓	✓	✗	✓
LLaMA	768	3072	6	16	✓	✓	✓	✗	✗	✓
LLaMA	1024	4096	8	4	✓	✓	✓	✓	✓	✓
LLaMA	2048	8192	16	4	✓	✓	✓	✓	✓	✓
LLaMA	3072	12288	24	4	✓	✓	✓	✓	✓	✓
LLaMA	4096	11008	32	4	✓	✓	✓	✓	✓	✓
LLaMA	4096	16384	32	4	✓	✓	✓	✓	✓	✓
LLaMA	5120	20480	40	4	✓	-	✓	✓	-	✓
LLaMA	6144	24576	48	4	✓	-	✓	✓	-	✓
LLaMA	7168	28672	56	4	-	-	✓	-	-	✓
LLaMA	8192	32768	64	4	-	-	✓	-	-	✓
Qwen	512	2048	4	4	✓	✓	✓	✓	✓	✓
Qwen	768	3072	6	1	✓	✓	✓	✗	✗	✓
Qwen	768	3072	6	2	✓	✓	✓	✓	✓	✓
Qwen	768	3072	6	4	✓	✓	✓	✓	✓	✓
Qwen	768	3072	6	8	✓	✓	✓	✓	✗	✗
Qwen	768	3072	6	16	✓	✓	✓	✗	✗	✓
Qwen	1024	4096	8	4	✓	✓	✓	✓	✓	✓
Qwen	2048	8192	16	4	✓	✓	✓	✓	✓	✓
Qwen	3072	12288	24	4	✓	✓	✓	✓	✓	✓
Qwen	4096	16384	32	4	✓	✓	✓	✓	✓	✓
Qwen	4096	22016	32	4	✓	✓	✓	✓	✓	✓

Continued on next page

Table 7 (continued)

Model	d	d_{ffn}	h	b	False Positive			False Negative		
					RTX 4090	RTX 5080	H100	RTX 4090	RTX 5080	H100
Qwen	5120	20480	40	4	✓	-	✓	✓	-	✓
Qwen	6144	24576	48	4	-	-	✓	-	-	✓
Qwen	7168	28672	56	4	-	-	✓	-	-	✓
Qwen	8192	32768	64	4	-	-	✓	-	-	✓
				Accuracy (%)	100	100	100	86.8	72.7	93.2

Table 8. Detailed verification results of Q, K, V, O, and FFN splits for monitoring *upper bounds* on false positives and false negatives on RTX 4090. Symbols A_b, A_y, and B_y indicate the split dimension used, while '-' denotes no split. A checkmark (✓) denotes success, while a cross (✗) denotes failure.

Q	K	V	O	FFN-up	FFN-down	Total Split Count	False Positive	False Negative
-	-	-	-	-	-	0	✓	✓
B_y	B_y	B_y	B_y	B_y	B_y	6	✓	✓
A_y	A_y	A_y	A_y	A_y	A_y	6	✓	✓
A_b	A_b	A_b	A_b	A_b	A_b	6	✓	✓
A_y	-	-	-	-	-	1	✓	✓
-	-	-	-	A_y	-	1	✓	✓
A_b	A_y	-	-	-	-	2	✓	✓
A_b	-	-	-	A_y	-	2	✓	✓
A_b	-	-	-	B_y	A_y	3	✓	✓
-	-	A_y	B_y	A_y	-	3	✓	✓
-	B_y	B_y	-	A_y	B_y	4	✓	✓
B_y	-	A_y	A_y	-	A_b	4	✓	✓
B_y	A_y	A_y	B_y	-	A_y	5	✓	✓
-	A_b	A_b	B_y	A_y	A_y	5	✓	✓
						Accuracy (%)	100	100