

Hardware-Rooted Inference-Time Verification for LLM Execution on Untrusted GPUs

Beijie Liu*, Haoxuan Xu*, Jie Peng†, Minghao Lin*, Chen Gong*, Tianlong Chen†, Mengyuan Li*
*University of Southern California
†The University of North Carolina at Chapel Hill
{beijieli,xuhaoxua,mlin7282,cgong459,mli49061}@usc.edu
pjie6140@gmail.com, tianlong@cs.unc.edu

***Abstract*—When LLM inference is served across trust boundaries, the receiver has no technical means to verify that a response came from the intended model under a trustworthy runtime. This creates a need for per-inference execution evidence that can be checked before a downstream client consumes the result. Existing approaches do not yet provide such evidence for practical online LLM serving. Cryptographic proof systems remain too slow for online serving; GPU Trusted Execution Environment (TEE)-based systems require specialized hardware unavailable on most deployments; and prior CPU TEE-based systems lack the inference-specific optimizations required for practical evidence generation at serving time.**

We present ARGUS, a system that uses a CPU confidential VM (CVM) as a hardware root of trust to generate per-inference execution evidence for GPU-accelerated LLM serving. The GPU remains an untrusted accelerator and runs the full LLM forward pass, while an attestable verifier inside the CVM checks the execution and signs a receipt binding the response to the model, request, runtime, and verification policy. ARGUS is built around system-level optimizations that reduce CPU-GPU data movement, trust-boundary crossings, and token-level synchronization. Together, these optimizations make it possible to generate evidence at inference time while preserving most of the GPU serving throughput. On six open-source models (1B–8B parameters), ARGUS adds only 18–74% end-to-end overhead on balanced workloads, relative to unverified GPU inference, compared to $45\times$ GPU-prefill latency for a reimplemented Slalom-style synchronous verifier. Across six arithmetic-attack classes, ARGUS detects every token-changing corruption with zero false positives across our honest evaluation.

I. INTRODUCTION

LLM inference increasingly crosses trust boundaries. A response may come from a cloud API [1] [2], a rented accelerator [3] [4], or a third-party agent embedded in a larger workflow [5]. In these settings, an inference response alone does not justify trust in the result: it does not show that the response is calculated via an intended LLM inference, or that the inference occurred under an expected and secure environment.

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.233233
www.ndss-symposium.org

This creates an urgent need for *per-inference execution evidence*: an artifact generated during inference that binds a response to the model, request, runtime, and policy under which the response was produced. Such evidence becomes especially important when model outputs feed active decision pipelines or are produced by an external computing substrate: evidence available at delivery time in turn makes the result safer to consume.

Existing approaches to verifiable inference fall into two families: cryptographic proof systems and Trusted Execution Environments (TEEs). Crypto-based proof systems offer strong cryptographic guarantees, but current DNN proof systems require expensive non-native arithmetic and remain too slow for online inference [6], [7], [8]. TEE-based approaches provide a more practical hardware root of trust, but they face a deployment split between trusting the GPU itself and using a CPU TEE to supervise an untrusted GPU. One line of work relies on GPU TEE-based inference systems, where the GPU that performs inference itself has TEE support. This design can directly attest to GPU execution, but is now available only on selected datacenter GPUs and software stacks, such as NVIDIA H100-class deployments [9], leaving much of the existing LLM serving infrastructure unsupported. A second line relies on CPU TEE-based inference systems [10], [11], [12], [13]. Compared with emerging GPU TEEs, CPU TEEs are more mature and broadly available across almost all server-class processors, including enclave mechanisms, such as Intel SGX [14], and confidential VMs (CVMs), such as AMD SEV-SNP [15], Intel TDX [16], and ARM CCA [17]. In this paper, we target the second CPU TEE-based setting: the CPU provides the deployable hardware root of trust, while the LLM computation remains accelerated by an untrusted GPU.

However, existing CPU-TEE inference systems fall short of generating practical evidence at inference time for three reasons. (1) Most systems are designed to provide a protected cloud execution environment rather than faithful-execution evidence [10], [11], [18]. In other words, they aim to provide both confidentiality and integrity, which leads to additional mechanisms such as activation obfuscation. For per-inference execution evidence, however, confidentiality is often not the central requirement. The request is already sent to the inference service provider, and the model may be provider-owned or public. The verifier instead needs integrity evidence that the

delivered response was produced by the intended inference computation. (2) Most existing systems are built on enclave-style TEEs such as Intel SGX [14], [11], whose enclave-based designs have well-known performance limitations. CVMs, on the other hand, provide a more practical deployment substrate with greater performance potential [15], [16], [17]. (3) Existing systems leave substantial room for system optimization. For example, systems such as Slalom [10], SOTER [11], and ArrowCloak [18] split the model between the TEE and the GPU, enforcing synchronization at layer boundaries. This places the CPU-side check on the critical path of every computation, resulting in a $45\times$ slowdown for a Slalom-style synchronous verifier compared with an unverified GPU prefill baseline.

These observations point to a different design point: using the CVM as the deployable hardware root of trust and optimizing it specifically for LLM inference verification. This leads to the research question of this paper: *Can a CVM-backed verifier, with optimizations tailored to inference verification, generate inference-time evidence for faithful LLM serving?*

In this paper, we present ARGUS, a CVM-rooted runtime verification system for practical inference-time evidence generation in GPU-accelerated LLM serving. ARGUS relies on a CVM as the deployable hardware root of trust and treats the GPU as an untrusted accelerator. For each request, an attestable verifier inside the CVM checks GPU execution and signs an evidence receipt for the accepted inference. The receipt binds the response to the model and request identities, the measured runtime, the verification policy, and the checked execution, so a client can validate the trusted runtime and policy before consuming the response.

ARGUS achieves practical inference-time evidence generation through four optimizations for CVM-based inference verification. (1) *Per-layer verification pipelining* reduces synchronization overhead by letting the GPU execute the forward pass uninterrupted while the CVM verifies each layer's outputs concurrently with subsequent GPU computation. (2) *Minimum transfer set* reduces CPU-GPU data movement by transferring only matmul outputs that cannot be regenerated inside the CVM, while locally recomputing cheap non-linear intermediates. (3) *Packed frame transfer* reduces trust-boundary crossing overhead by placing each layer's transferred tensors in one contiguous DMA frame and decoupling the untrusted SWIOTLB frame from the trusted CVM copy. (4) *Prefill-decode scheduling* reduces critical-path latency by overlapping prefill verification with later GPU execution and replacing per-token decode checks with one batched verification pass over the verified prefill KV cache. Together, these optimizations keep the CVM verifier off the token-by-token critical path while preserving the evidence receipt for the returned response.

We implement ARGUS on AMD SEV-SNP and evaluate it on six open-source models from 1B to 8B parameters. ARGUS only adds 18–74% end-to-end overhead on balanced workloads relative to native unverified GPU inference. In the same benchmark, full CPU recomputation runs at $172\times$ the GPU prefill baseline, and a Slalom-style synchronous verifier runs at $45\times$ the GPU prefill baseline. Beyond performance, we evaluate whether ARGUS accepts honest GPU executions under floating-point variation while rejecting corruptions that affect generation. Its per-role tolerance calibration preserves a measured worst-case $7.7\times$ margin between the calibrated tolerance and maximum honest residual. Across six arithmetic-attack classes, ARGUS detects every corruption that changes a decoded token, with zero false positives across our honest evaluation. The token-fabrication probe separately evaluates optional top-k membership checking.

In summary, this paper makes the following contributions:

1) **CVM-rooted evidence for GPU-accelerated LLM inference.** We introduce ARGUS, a system that uses a broadly deployable CPU CVM to generate per-inference execution evidence for GPU-accelerated LLM serving. ARGUS defines the evidence record and the verifier interface needed to connect a delivered response to an attested runtime and an accepted execution.
2) **Full-stack optimizations for CVM-based inference verification.** We design a set of optimizations that reduce the dominant overheads in CPU-CVM-rooted verification. Per-layer verification pipelining reduces synchronization overhead, the minimum transfer set and packed DMA frames reduce data movement across the trust boundary, and prefill-decode scheduling removes per-token verifier stalls from generation, making the CVM verifier practical for online LLM serving without requiring GPU TEE support.
3) **Practical inference-time evidence generation.** We implement ARGUS on AMD SEV-SNP and evaluate it on six open-source models from 1B to 8B parameters. ARGUS generally adds 18–74% end-to-end overhead on balanced workloads relative to unverified GPU inference. In the prefill-verification ablation on Qwen3-1.7B, the same prefill verifier completes in 214 ms versus 5,103 ms for full CPU recomputation ($23.8\times$ faster) and 1,350 ms for a Slalom-style synchronous verifier ($6.3\times$ faster), while detecting every token-changing corruption in our six arithmetic-attack classes, with zero false positives across our honest evaluation.

II. BACKGROUND

A. Confidential VMs and Device I/O

A Confidential Virtual Machine (CVM) protects a guest from a privileged host, including the VMM/hypervisor and devices outside the CPU trust boundary [15], [16]. Guest-private memory is protected by hardware-rooted memory-encryption keys, and the guest's architectural state is isolated from the host [19], [15], [16]. Remote attestation lets a client validate the platform configuration and the guest's measured launch state before trusting software inside the CVM [15], [16].

This protection changes the I/O path for DMA devices such as GPUs. Because CVM-private memory is protected by CPU-controlled encryption and integrity mechanisms, an external device cannot directly DMA into private pages as ordinary host memory [15], [20]. Instead, device I/O must go through memory that the guest explicitly shares with the untrusted host

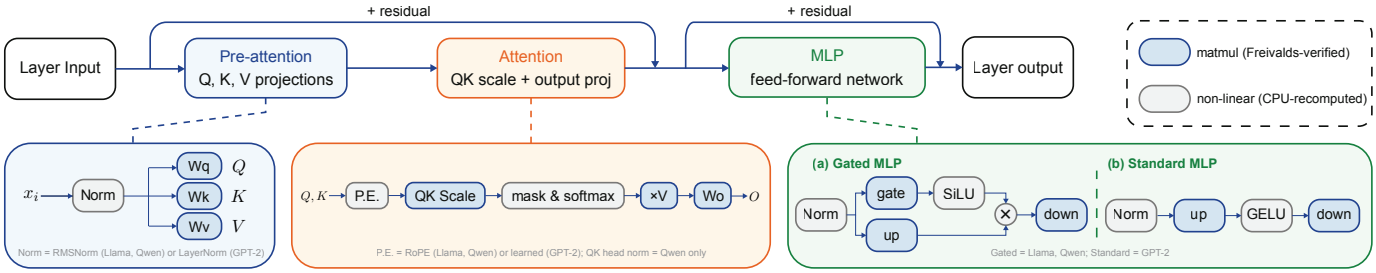


Fig. 1. A generic transformer layer grouped into the three verification stages (§IV-D). Blue blocks are matmuls verified by Freivalds; the rest are non-linear ops recomputed in the TEE.

or device. On Linux, this is commonly implemented through the DMA API and SWIOTLB bounce buffers [21]: the device reads from or writes to a shared, unencrypted staging buffer, and the guest CPU copies data between that buffer and private CVM memory. The shared buffer is therefore part of the untrusted I/O path, not a trusted communication channel.

Representative x86 CVM platforms include Intel TDX [16], [20] and AMD SEV-SNP [15], [19]. Both expose the same high-level distinction between private guest memory and shared memory used for host/device communication. In the rest of this paper, we instantiate this model using AMD SEV-SNP, which is the platform used in our prototype.

B. LLM Inference: Prefill and Decode

Autoregressive LLM inference proceeds in two phases [22], [23]. We denote the input prompt length by T, the number of tokens generated during decode by G, the number of transformer layers by L, the hidden dimension by d, and the per-head dimension by d_h. **Prefill** processes the entire input prompt $x_1, \ldots, x_T$ in a single forward pass through all L layers, producing a key-value (KV) cache for the generation that follows. This phase is compute-bound; attention complexity is $O(T^2 d_h)$ per layer. **Decode** generates tokens one at a time. Each step runs a forward pass for the new token, attending over the cached KV pairs from all previous tokens. Decode is memory-bound (weight loading dominates over computation) and inherently sequential.

Fig. 1 shows the operations within one transformer layer [24]. The layer consists of an attention block and an MLP block, connected by residual additions and layer normalizations. The operations fall into two categories: *matrix multiplications* (Q/K/V projections, the QK and QKV products in attention, the output projection, and MLP projections) and *non-linear operations* (normalizations, RoPE, residual additions, activation functions). This split is central to our design: the matrix multiplications that dominate compute are exactly the operations Freivalds' algorithm verifies cheaply, and the remaining operations are cheap enough to recompute inside the TEE.

C. Matrix Verification Techniques

Freivalds' algorithm [25] is a classical randomized method for verifying matrix multiplication [26]. Given matrices $X \in \mathbb{R}^{n\times d}$, $A \in \mathbb{R}^{d\times m}$, and a claimed product $Y \in \mathbb{R}^{n\times m}$, it samples a random vector $r \in \{-1, 1\}^m$ and tests:

$$X(Ar) \stackrel{?}{=} Yr \tag{1}$$

If $Y = XA$ the test always passes. If $Y \neq XA$, let $\Delta = Y - XA$ and let $\boldsymbol{\delta}$ be any row of Δ with a non-zero coordinate δ_j. Decomposing the inner product as $\boldsymbol{\delta}\cdot r = \delta_j r_j + \sum_{k\neq j} \delta_k r_k$ and conditioning on r_k for $k \neq j$, the inner product vanishes only when $r_j = -\sum_{k\neq j} \delta_k r_k/\delta_j$. Since r_j is uniform on $\{-1, 1\}$, at most one of its two values satisfies this equality, so the per-check false-acceptance probability is at most $1/2$. This bound is independent of the workload dimensions d and m. Repeating the test with k independent vectors drives false-acceptance to at most 2^{-k}.

The key property is the asymmetry in cost. The multiplication XA costs $O(ndm)$; the check costs only $O((d+n)m)$. When A is known in advance, $r'{=}Ar$ can be precomputed, which reduces the per-query cost to $O(nd)$ versus $O(nd^2)$ for a square projection.

The equality test in (1) is exact only under integer or finite-field arithmetic. In floating point, non-associative accumulation leaves the residual $X(Ar) - Yr$ rarely exactly zero even on honest runs. The test becomes a bounded-residual check $\|X(Ar) - Yr\| \leq \varepsilon$, where ε must separate inherent FP noise from adversarial corruption.

III. ARGUS: INFERENCE WITH RUNTIME EVIDENCE

This section defines the setting in which ARGUS generates runtime evidence for faithful inference without assuming a GPU TEE. We first state the threat model, then describe the goals that ARGUS is designed to satisfy under this threat model.

A. Threat Model

ARGUS targets deployments where GPU-accelerated inference is performance-critical but the GPU does not provide a TEE capable of directly attesting the full inference execution. In this setting, ARGUS does not attempt to make the GPU trusted. Instead, it produces runtime evidence that links an inference response to the claimed model, input, and execution path, allowing a verifier to assess whether the response is consistent with faithful inference. We inherit the AMD SEV-SNP trust model: the CPU, its memory-encryption engine, and the software inside the attested CVM are trusted. The measured software TCB includes the guest OS/kernel, the

guest CUDA runtime and driver, the ARGUS verifier, and the inference service. In the intended deployment, these components are included in the measured CVM image, and the model checkpoint is loaded at boot.

The adversary controls everything outside the CVM private-memory boundary. This includes the hypervisor and host OS, the GPU device and its firmware, microcode, command processor, VRAM, and compute behavior, as well as the unencrypted SWIOTLB bounce buffer and PCIe path. The adversary may intercept and modify any value crossing this boundary, including GPU outputs, bounce-buffer contents, and in-flight PCIe traffic. Concretely, the adversary may return tensors produced using modified weights, downgraded arithmetic, skipped kernels, stale or replayed values, or arbitrary fabricated data. The adversary knows the model architecture, model weights, and verification algorithm, but not the verifier's internal randomness, which remains protected within the CVM.

ARGUS does not provide model or input confidentiality against the GPU or service provider. We also do not defend against side channels, model extraction through access patterns, denial of service, or privacy properties beyond those provided by the underlying CVM platform.

B. Security and Performance Goals

Under this threat model, ARGUS aims to release a GPU-served LLM response only when an attested CVM verifier accepts runtime checks for the computation that produced it. For each accepted request, the verifier produces a receipt binding the response to the request, the committed model, the measured runtime, the verification policy, and the checks that passed. This goal decomposes into two requirements.

Soundness against an adversarial GPU. Every tensor crossing from the untrusted GPU domain into the CVM is treated as untrustworthy until checked. An accepted response must pass the verifier's checks for the committed model, request, and returned token sequence. Specifically, response-relevant corruptions such as tampered matmul outputs, modified weights, downgraded arithmetic, skipped kernels, stale values, or fabricated tensors should all be detected. ARGUS checks token selection against verified logits under the committed sampling policy (§IV-F).

Practical serving latency. Runtime evidence must be generated on the online serving path without changing the order of inference latency. Each check, cross-boundary transfer, and coordination step must fit within the ordinary autoregressive serving budget. Adding ARGUS should therefore introduce only a small constant-factor overhead while preserving the invariant that no accepted response is released without valid runtime evidence.

IV. ARGUS DESIGN

We first give the verification workflow (§IV-A), then describe the memory domains and runtime components that implement the trust boundary (§IV-B). We next define the minimum per-layer transfer set (§IV-C), give the per-layer verification pipeline (§IV-D), extend the pipeline to autoregressive generation (§IV-E), and check token selection against the verified logits under a committed sampling policy (§IV-F).

A. Verification Workflow

ARGUS exploits a structural property of the transformer to bound the GPU's ability to forge verifiable outputs. Layer operations partition into two classes with opposite security properties. Matrix multiplications are computationally expensive for the CVM to rerun but efficient to check: Freivalds reduces verification to $O(nd)$ against a committed weight projection, at a false-acceptance probability the verifier controls. Non-linear operations are cheap for the CVM to recompute from a trusted anchor; any GPU claim over a non-linear output is therefore redundant by construction.

This partition determines what must cross the trust boundary. Each layer's packed frame supplies `LAYER_INPUT` as the CVM-trusted anchor and the GPU's claimed matmul outputs as the values to be checked. The first layer's input is accepted only after its secret projection matches the embedding projection of the input token IDs, precomputed from the trusted checkpoint (including positional embeddings where applicable); subsequent inputs are linked by the cross-layer residual check. Non-linear outputs are excluded: the CVM derives normalization, positional encoding, masking, softmax, residual addition, and activations in encrypted memory from the accepted anchor and the matmul outputs the Freivalds test has cleared. A GPU-claimed non-linear value would enter the verification chain only if accepted as a matmul output, which the Freivalds check prevents. Fig. 2 traces this path for one transformer layer.

ARGUS accepts at two granularities. A layer frame is accepted when every matmul checkpoint passes its Freivalds test against the weight projection the CVM holds and the input the CVM derives independently; a failing check aborts the request and releases no output. A request is accepted when the accepted layer frames form a consistent residual chain, with each layer's anchor equal to the accumulated residual of its accepted predecessors, and token-selection checks against the verified logits pass under the committed sampling policy. Arithmetic corruptions and cross-layer inconsistencies are subject to the probabilistic checks and floating-point tolerances analyzed in §VI; token selection is checked against the accepted logits.

B. Memory Domains and Runtime Components

ARGUS enforces the trust boundary through three memory domains that differ in trust level and I/O role. We assign computation, transport, and verification to separate domains so that no single component crosses a trust boundary in both directions.

GPU VRAM. GPU VRAM holds the model weights and all GPU-side computation buffers. A compute stream executes the model layer by layer; a dedicated copy stream issues asynchronous device-to-host (D2H) transfers for packed layer frames. The two streams run on separate CUDA hardware queues, so the GPU can begin computing layer $i{+}1$ while layer i's frame transfers. Every tensor emitted from this domain

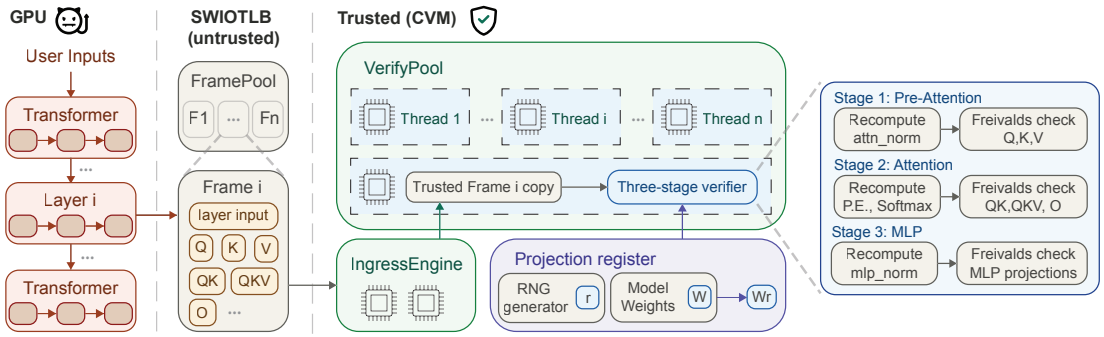


Fig. 2. ARGUS end-to-end verification workflow. Each layer's tensors are packed into one frame, copied into encrypted memory, and verified by a three-stage CPU verifier (right) that recomputes non-linear operations and Freivalds-checks matmuls against the precomputed projection Wr.

lies outside the TCB: the GPU, its firmware, VRAM contents, and compute behavior are all adversary-controlled, and every emitted tensor is an adversarial claim that the verifier must check before use.

SWIOTLB bounce buffer. The SWIOTLB bounce buffer is the DMA-accessible staging area mandated by the SEV-SNP I/O path. Because it is outside the trusted memory boundary, ARGUS treats it as a transient transport region only. We manage it through a FramePool: a pre-allocated pool of contiguous slots, each sized for one layer's verifier-visible tensors. A slot cycles through FREE $\rightarrow$ INFLIGHT (D2H in progress) $\rightarrow$ READY (awaiting trusted copy) $\rightarrow$ FREE. The IngressEngine reclaims a slot immediately after copying it into encrypted memory; the verifier never rereads the SWIOTLB region. An adversary that modifies the bounce buffer after the copy has no path to the values the verifier checks.

Encrypted CVM memory. Encrypted CVM memory holds all verifier state and every tensor that has crossed the trust boundary. The IngressEngine is a small pool of copy threads that moves ready frames from SWIOTLB into encrypted memory; this copy is the sole ingress point for untrusted data. Any tampering in transit is subject to downstream Freivalds rejection; any tampering after the copy cannot reach the verifier.

The ProjectionRegistry stores the secret random vectors $r \in \{-1, +1\}^m$ and the precomputed projections $r' = Wr$ for each checked matmul. Holding compact projections rather than full weight matrices avoids storing full model weights in verifier memory; projection storage still scales with layer count, dimensions, and verification repetitions. The VerifyPool is a fixed set of worker threads that executes Freivalds checks and local recomputation on trusted frames. We pin IngressEngine and VerifyPool workers to disjoint core sets to prevent bandwidth-bound copying and compute-bound verification from contending on the same physical cores.

C. Minimum Transfer Set

The memory architecture fixes where tensors reside and how they cross the trust boundary. What remains is establishing which tensors must cross it. The minimum sufficient set is exactly the state from which the CVM can derive every other layer intermediate and check every matmul output without accepting any additional GPU claim as trusted.

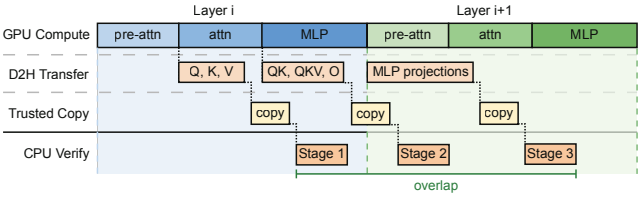


Fig. 3. Per-layer verification pipeline (schematic). Each layer's matmul outputs are emitted in three groups; the CPU verifies each group concurrently with subsequent GPU compute. Only the last layer is exposed (drain).

Transferring every GPU-produced intermediate exceeds this minimum and makes D2H bandwidth the bottleneck. At FP32, one layer of an 8B-class model produces tens of megabytes of intermediates per token; the SWIOTLB path caps GPU-to-CPU bandwidth well below GPU compute throughput. Transferring every intermediate would make data movement, not Freivalds checking, the dominant cost.

The minimum set is `LAYER_INPUT` plus the GPU-claimed matmul outputs. For a standard transformer block this means the Q, K, V projections, the attention products QK and QKV, the output projection O, and the MLP projection outputs. For gated MLPs, the MLP fields are the gate, up, and down projection outputs; for standard MLPs, they are the up and down projection outputs.

The frame excludes outputs of normalization, positional encoding, masking, softmax, residual addition, and activation functions. The CVM derives all of these locally from the layer anchor and the accepted matmul outputs, so no GPU-produced non-linear value enters the verification chain as an independent claim. Excluding them provides no additional attack surface to the adversary: the GPU cannot forge a non-linear output that the CVM will accept without first forging the matmul output that serves as that operation's input, which the Freivalds check would detect.

Each layer therefore contributes one contiguous frame containing `LAYER_INPUT` and the layer's matmul outputs: one bulk D2H transfer into SWIOTLB and one trusted copy into encrypted memory per layer.

D. Per-Layer Verification Pipeline

Placing the verifier serially after each GPU layer would add its full latency to the inference critical path. ARGUS avoids

this by overlapping verification with GPU compute at two granularities. Across layers, it runs a four-stage pipeline (GPU compute → D2H transfer → trusted copy → Freivalds verify) so that verification of layer i proceeds while the GPU computes later layers. Within each layer, it decomposes the Freivalds checks into three stages that follow the transformer's data dependencies, creating three fine-grained synchronization points rather than one coarse barrier per layer.

Inter-layer pipeline. Fig. 3 shows the inter-layer schedule. In steady state, verification of layer i is hidden by some combination of layer i's own transfer tail and layer $i{+}1$'s GPU compute. The exposed overhead reduces to the pipeline drain: the verification of the last few layers after the GPU finishes, not the sum of per-layer verification times.

Acceptance criterion. Each Freivalds check is parameterized by the committed weight W, a secret random vector $r \in \{-1, +1\}^m$ held in encrypted CVM memory, and the precomputed projection $r' = Wr$ from the ProjectionRegistry. For a GPU-claimed output $\hat{T}$ with verifier-derived input x, the verifier computes the normalized residual $\rho = \|x \cdot r' - \hat{T} \cdot r\|/(\sqrt{n} \cdot \|r'\|_2)$ and accepts if $\rho \leq \varepsilon$. The normalization makes ε scale-invariant across layers and model sizes.

Intra-layer stages. Each layer decomposes into three chained stages that follow the transformer's data dependencies. Each stage spans the matrix multiplications that share a downstream non-linear barrier: the CVM must complete the barrier recomputation (normalization, RoPE, softmax, residual addition, or activation) before supplying the trusted input to the next stage. This yields three synchronization points per layer.

Let x_i denote the input to layer i. Each layer produces an attention output a_i and an MLP output m_i; the next layer's input is $x_{i+1} = x_i + a_i + m_i$.

Stage 1 (Projection). The CPU recomputes $\bar{x}_i = \mathrm{Norm}(x_i)$ locally. Against this trusted input, three Freivalds checks cover the Q, K, V projections:

$$\bar{x}_i \cdot r'_Q \approx \hat{Q} \cdot r_Q, \qquad \bar{x}_i \cdot r'_K \approx \hat{K} \cdot r_K, \qquad \bar{x}_i \cdot r'_V \approx \hat{V} \cdot r_V,$$

where $r'_Q = W_Q r_Q$ (and analogously for K, V) is retrieved from the ProjectionRegistry. Any deviation in $\hat{Q}$, $\hat{K}$, or $\hat{V}$ from the output $\bar{x}_i$ would produce under the committed weights fails the corresponding check.

Stage 2 (Attention). The CPU applies model-specific head normalization and positional encoding to the accepted $\hat{Q}$ and $\hat{K}$, then checks the scaled dot-product:

$$\hat{Q} \cdot (\hat{K}^\top r_{\mathrm{QK}}) \approx \widehat{QK} \cdot r_{\mathrm{QK}}.$$

For the attention-value product, the verifier first evaluates $\hat{V} \cdot r_{\mathrm{QKV}}$ from the accepted V, then checks:

$$\mathrm{softmax}(\widehat{QK} + \mathrm{mask}) \cdot (\hat{V} \cdot r_{\mathrm{QKV}}) \approx \widehat{QKV} \cdot r_{\mathrm{QKV}}.$$

This reduces the check from a matrix-matrix to two matrix-vector products. Softmax is evaluated with a streaming implementation that avoids materializing the full score matrix. The output projection check closes the attention block:

$$\widehat{QKV} \cdot r'_O \approx \hat{O} \cdot r_O, \qquad r'_O = W_O r_O.$$

The accepted attention output is $a_i = \hat{O}$.

Stage 3 (MLP). The CPU reconstructs the first residual $x_i + a_i$ and recomputes the post-attention normalization $\bar{h}_i = \mathrm{Norm}(x_i + a_i)$. For gated architectures (e.g., Llama, Qwen), the gate and up projections are checked against this trusted input:

$$\bar{h}_i \cdot r'_{\mathrm{gate}} \approx \widehat{\mathrm{gate}} \cdot r_{\mathrm{gate}}, \qquad \bar{h}_i \cdot r'_{\mathrm{up}} \approx \widehat{\mathrm{up}} \cdot r_{\mathrm{up}}.$$

The CVM then recomputes $\mathrm{SiLU}(\widehat{\mathrm{gate}}) \odot \widehat{\mathrm{up}}$ locally and uses the result as the trusted input to the down-projection check:

$$(\mathrm{SiLU}(\widehat{\mathrm{gate}}) \odot \widehat{\mathrm{up}}) \cdot r'_{\mathrm{down}} \approx \widehat{\mathrm{down}} \cdot r_{\mathrm{down}}.$$

For standard architectures (e.g., GPT-2), the MLP has two projections with a GELU activation; the verifier checks up, recomputes $\mathrm{GELU}(\widehat{\mathrm{up}})$ locally, and checks down. The accepted MLP output is $m_i = \widehat{\mathrm{down}}$.

Cross-layer chain consistency. After all per-layer checks pass, a batch check confirms the residual identity $x_{i+1} = x_i + a_i + m_i$ for all $i \in [0, L{-}2]$. This closes a specific attack surface: a GPU that passes every intra-layer check yet drops or substitutes a residual addition between layers. The check is a floating-point consistency test, not a cryptographic commitment; it uses a separate absolute residual tolerance.

E. Prefill-Decode Verification Protocol

The per-layer pipeline applies directly to prefill: the GPU processes the prompt in a single forward pass, emits one packed frame per layer, and the verifier drains those frames concurrently. Autoregressive decode breaks this structure. Each token is generated in a separate forward pass; attaching hook capture and D2H transfer to every decode step places verifier overhead on the sequential per-token critical path, eliminating the concurrency the pipeline provides.

ARGUS decouples verification from generation with a three-phase protocol. It verifies prefill in the background, runs decode at native GPU speed with hooks disabled, and verifies the generated suffix in one post-generation pass that reuses the verified prefix KV cache. The response is released only after this final pass succeeds (Fig. 5).

Prefill Verify. The GPU executes prefill with hooks enabled and emits one packed frame per layer. A background verifier thread consumes these frames through the per-layer pipeline and retains the accepted K and V tensors per layer, forming the verified prefix KV cache that Decode Verify will use.

Decode. Hooks are disabled during generation; the GPU produces tokens at native speed while Prefill Verify drains concurrently in the background. If Prefill Verify rejects any layer frame, the request aborts before any response is released, regardless of whether decode has finished.

Decode Verify. After all G tokens are generated, the verifier runs one post-generation pass over the suffix using the verified prefix KV cache. The causal mask enforces the same dependency order as autoregressive decode: each generated position attends only to the prompt and to earlier generated positions. This pass submits the same layer frames that per-step

decode verification would submit, so it reuses the per-layer pipeline and Freivalds checks without modification. It avoids the G-fold D2H and hook-capture cost of per-token verification by collapsing all generated positions into one compute-bound forward pass.

In our measured balanced configurations ($T/G \leq 2$), Prefill Verify completes during decode and contributes zero wall-clock latency. The protocol enforces the invariant that no unverified GPU output is returned to the client.

F. Token-Selection Verification

Decode Verify establishes that the GPU's matmul outputs are consistent with the committed weights, but it operates on the token sequence the GPU *returned*. A malicious GPU that computes every projection honestly can still substitute the emitted tokens: matmul residuals are determined by layer computations, not by the token indices the sampler reports, so arbitrary token substitution leaves every Freivalds check intact.

By default, ARGUS verifies deterministic greedy decoding: the first emitted token must equal the argmax of the checked prefill logits, and subsequent tokens must equal the argmax at the corresponding positions of the checked suffix pass. A mismatch rejects the request.

ARGUS also checks token membership under a committed top-k sampling policy. The LM-head projection is itself a Freivalds-checked matmul, so Decode Verify already produces a verified logit vector $\hat{\ell}$ at every decoded position. The verifier extracts the top-k indices of $\hat{\ell}$ and rejects the request if the emitted token is absent from that set. This check is binding: a GPU that returns a token outside the verified top-k is detected with certainty, regardless of how honest its preceding layer computations were.

The additional cost beyond Decode Verify is one partial sort and one membership test per decoded position. Overhead is position-dependent and is reported in §VI-D and Table X.

Tokens inside the verified top-k are indistinguishable from honest samples by construction. Sampling policies without a finite candidate set, such as unconstrained top-p or full-vocabulary temperature sampling, do not admit a deterministic accept/reject test of this form. We discuss these boundaries in §VII.

V. EVALUATION

In this section, we evaluate ARGUS's performance. We first describe the prototype and experimental setup, then measure end-to-end overhead and break down where the cost comes from. Finally, we ablate the main system optimizations and study sensitivity to the Freivalds repetition count k, which reduces false-accept probability at the cost of additional CPU verification work.

A. Implementation and Setup

Environment setup. We evaluate ARGUS on an AMD EPYC 9115 16-core CPU with an NVIDIA RTX PRO 6000 Blackwell Workstation Edition GPU (96 GB GDDR7) attached through PCIe passthrough. Ubuntu 24.04 runs inside a QEMU SEV-SNP CVM with 54 GiB encrypted RAM and a 2 GB SWIOTLB bounce buffer. We test six open-source checkpoints: GPT-2-XL (1.6B), Llama-3.2-1B, Llama-3.2-3B, Qwen3-1.7B, Qwen3-4B, and Qwen3-8B. All models run in FP32 with eager attention unless otherwise stated. The ARGUS prototype comprises ∼2,700 lines of C++ and ∼3,200 lines of Python.

Arithmetic and kernel configuration. We use eager attention because it exposes the per-operation intermediate tensors that ARGUS hooks for verification. Fused kernels such as FlashAttention would require modifying the kernel source to export those intermediates. We use FP32 because lower precisions such as FP16 and BF16 narrow the margin between honest floating-point noise and adversarial corruption, raising security concerns for tolerance calibration that we discuss in §VII.

Model-agnostic adapter framework. Modern transformer families differ in attention layout, normalization, activation function, and weight format. To avoid special-casing each model inside the verifier, ARGUS exposes a single `ModelAdapter` interface that captures these differences in one place. A new model family is supported by writing an adapter that tells the verifier where layers live in the model graph, which tensors to hook for capture, and which recomputation options to use; the rest of the verifier remains adapter-agnostic.

The three adapters in our prototype cover Qwen3, Llama, and GPT-2. They differ along four axes: attention layout (GQA vs. MHA), normalization (RMSNorm vs. LayerNorm), MLP variant (gated vs. standard), and weight format (linear vs. Conv1D). GPT-2 is the only family that requires more than configuration: its Conv1D weight layout and packed QKV projection are transposed and split inside the adapter so that the rest of the pipeline sees ordinary linear weights.

Methodology. We compare ARGUS against an unverified GPU baseline running the same model, prompt, and decoding configuration on the same hardware. Unless otherwise stated, inference measurements use 10 warmup iterations followed by 10 measured repetitions, and we report medians. The end-to-end study (§V-B) evaluates all six models over the measured (T, G) grid from $(64, 128)$ to $(1024, 2048)$. The ablation (§V-C) and repetition-sensitivity study (§V-D) use Qwen3-1.7B because its smaller size keeps CPU-side baselines and repeated verification runs tractable; the end-to-end heatmap shows that the same overhead trends hold across the larger models. Greedy token-selection verification is enabled by default; top-k verification is optional.

We implement the Slalom-style baseline in the same CVM without confidentiality mechanisms. We report steady-state inference latency, excluding one-time initialization costs.

B. End-to-End Performance

We evaluate end-to-end cost from three angles. We first measure latency overhead across all six models and a grid of prompt and generation lengths to identify the workload trend. We then zoom in on Qwen3-4B to separate total wall-clock

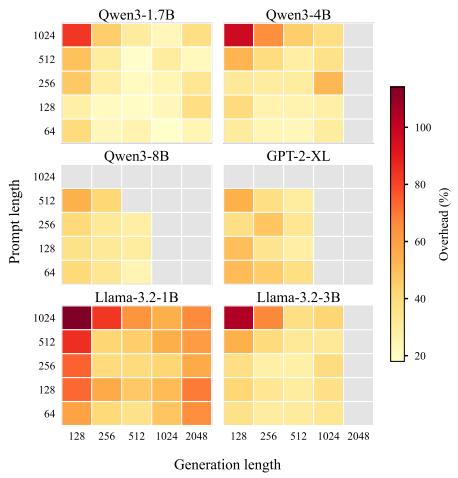


Fig. 4. ARGUS latency overhead (%) over (T, G) across all six models on a shared color scale. Each panel is one model; rows vary prompt length T and columns vary generation length G. Grey cells did not fit or were not measured. Numeric values in Appendix B.

latency from the per-output-token cost. Finally, we break the total into protocol phases to explain which verification work is hidden by decode and which remains exposed. We also report the memory footprint and the cost of producing an attested verification record.

Overall latency overhead. Fig. 4 shows a consistent workload trend across architectures: overhead is highest for prompt-heavy workloads, where the prompt is long and generation is short, and decreases as generation length grows. At T=512, overhead drops from 49–85% at G=128, through 30–43% at G=256, to 19–45% at G=512 across the tested models.

This trend follows the structure of the concurrent protocol. Prefill Verify is tied to the prompt-side computation, but it can overlap with Decode. When generation is long enough, Prefill Verify has time to drain in the background and adds no wall-clock latency. When generation is short, less of that work can be hidden, so prompt-heavy configurations have higher overhead. Smaller models also show higher relative overhead because the same verifier work occupies a larger fraction of their shorter unverified inference time.

Wall-clock and per-token latency. To make the aggregate trend concrete, Table I zooms in on Qwen3-4B with representative (T, G) configurations. The per-output-token cost is roughly stable across generation lengths: at T=512, ARGUS adds about 6–9 ms per output token over the unverified GPU baseline (∼31 ms vs. ∼24 ms). Total overhead, however, depends on the prompt/generation balance: it is lowest when generation is long enough to amortize verification, reaching +29% at $(512, 512)$, and highest when a long prompt is followed by a short generation, reaching +97% at $(1024, 128)$.

Phase breakdown. The reason appears in the phase-level breakdown. Table II reports the four ARGUS phases for two representative Qwen3-4B configurations, and Fig. 5 illustrates the same measurements as example wall-clock timelines. Prefill and Decode run on the GPU, while the CPU verifier runs Prefill

TABLE I
WALL-CLOCK AND PER-TOKEN DECODE COST ON QWEN3-4B. ARGUS DEC/TOK INCLUDES DECODE VER., THE COST A USER SEES PER OUTPUT TOKEN. FULL BREAKDOWN IN APPENDIX A.

		GPU		ARGUS		
T	G	total (ms)	dec/tok (ms)	total (ms)	dec/tok (ms)	Δ
256	128	3,252	24.9	4,284	31.2	+32%
	256	6,230	24.1	8,079	30.2	+30%
	512	12,555	24.4	16,363	31.2	+30%
	1024	22,905	22.3	34,736	33.4	+52%
512	128	3,196	24.2	4,891	32.9	+53%
	256	6,237	24.0	8,642	31.0	+39%
	512	12,727	24.7	16,436	30.6	+29%
	1024	24,773	24.1	33,625	32.0	+36%
1024	128	3,321	24.5	6,533	36.1	+97%
	256	6,112	23.1	10,115	32.5	+65%
	512	12,238	23.5	17,744	31.0	+45%
	1024	25,420	24.6	35,165	32.4	+38%

TABLE II
ARGUS PHASE COSTS (MS) ON QWEN3-4B. PREFILL VER. RUNS CONCURRENTLY WITH DECODE AND IS HIDDEN FROM WALL-CLOCK; ONLY DECODE VER. IS EXPOSED.

T	G	**Prefill**	**Prefill Ver.**	**Decode**	**Decode Ver.**
512	512	751	864	13,768	1,653
1024	128	1,911	2,177	3,654	684

Ver. (Prefill Verify) and Decode Ver. (Decode Verify). Prefill Ver. starts shortly after Prefill begins and continues during Decode; in both examples it finishes before Decode ends, so it is hidden from the request's wall-clock latency. Decode Ver. begins only after generation completes and is therefore the verification phase exposed to the user.

At $(512, 512)$, Decode dominates the request, so the exposed overhead is mainly the 1.65 s Decode Ver. pass after a 13.8 s Decode. At $(1024, 128)$, the prompt is much larger relative to the generation, but Prefill Ver. still fits inside the Decode window; only the 684 ms Decode Ver. pass remains on the critical path.

The latency results above explain the request critical path. We next report two deployment costs that are not captured by the phase timeline: the memory footprint across the three runtime memory regions, and the one-time cost of producing an attested verification record.

Memory footprint. Table III reports peak memory along the three deployment-relevant regions: untrusted GPU VRAM, trusted encrypted CPU memory inside the CVM, and the unencrypted SWIOTLB bounce buffer. The baseline is unverified GPU inference at the same configuration, which holds only the model weights and KV cache. Percentages are relative to this baseline. ARGUS increases GPU memory by 9.7%, mainly from packed staging buffers for verifier-visible tensors. The SWIOTLB footprint is the unencrypted FramePool used for GPU-to-CPU transfers. The trusted CPU footprint holds the

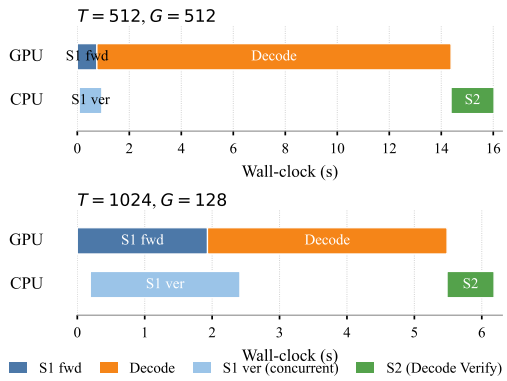


Fig. 5. Wall-clock timeline examples. Prefill Ver. overlaps Decode and finishes first; only Decode Ver. extends past the GPU's last token.

TABLE III
PEAK MEMORY ON QWEN3-4B AT $(T, G) = (1024, 256)$. PERCENTAGES ARE OVERHEAD VS. UNVERIFIED GPU BASELINE.

Region	Baseline	ARGUS
GPU (untrusted)	16.85 GB	18.48 GB (+9.7%)
CPU (trusted)	–	2.60 GB
SWIOTLB (unencrypted)	–	1.60 GB

Freivalds projection registry, trusted-side verification buffers, residual scratch space, and the verified prefill KV cache reused by Decode Verify; model weights are not persistently stored in TEE memory.

Receipt cost. Finally, each completed request produces a verification record containing the model identifier, the active tolerance ε, the Freivalds repetition count k, the emitted token IDs, and a client-supplied nonce. The record is hashed into an SEV-SNP attestation report, binding the verified response to that client request. Table IV reports the signing latency over 100 calls. The median cost is 18.7 ms, adding only $\sim$0.1% to the 16.4 s end-to-end latency of Qwen3-4B at $(T, G) = (512, 512)$. This cost is small compared with the inference and verification phases, so attestation is not a performance bottleneck.

C. Ablation Study

This subsection quantifies how much the main system mechanisms reduce that overhead. We use Qwen3-1.7B at $(T, G) = (256, 512)$, where expensive CPU-only and synchronous baselines remain tractable. The prefill ablation follows the path from CPU-only recomputation to Slalom-style per-matmul verification, full-pass capture, inter-layer pipelining, and packed transfers. The decode ablation compares decode-side verification strategies and the full concurrent protocol.

Table V and Table VI trace the path from baseline to the full ARGUS design by adding one mechanism at a time on Qwen3-1.7B at $(T, G) = (256, 512)$. The first table isolates prefill verification; the second adds decode verification on top of the best prefill variant.

Prefill verification. Table V isolates prefill-side improvements. A CPU-only forward pass costs 5.1 s, more than two orders of magnitude above the unverified GPU prefill, confirming that full CPU recomputation is not viable. A Slalom-style design that round-trips each matmul individually to the GPU and runs a synchronous Freivalds check on the CPU [10] reduces this to 1.4 s, a 3.8× improvement, but still runs 45× the unverified GPU prefill because per-matmul launch and transfer overhead dominate.

Letting the GPU execute the full prefill forward pass and verifying the captured layer frames afterward (*full-pass, serial*) reduces the cost to 1,033 ms, a further 1.3× improvement over the Slalom-style variant, but still 34× the GPU prefill baseline because verification is serialized after GPU execution. Adding a 4-frame inter-layer pipeline (*pipelined*) cuts this by 4.5× to 229 ms by overlapping transfer, trusted copying, and Freivalds checks across layers. Packing each layer's transferred tensors into one contiguous frame (*pipelined + packed*) saves another 7%, reaching 214 ms. This is the prefill configuration used in the full system.

Decode verification. Building on the best prefill variant (*pipelined + packed*), Table VI compares decode-side verification strategies. Verifying each token synchronously before emitting the next (*per-token verify*) inflates decode from 9,855 ms to 51,712 ms, dominated by per-token hook and transfer overhead. The full ARGUS configuration removes this per-token cost by deferring verification to a single post-generation suffix pass and running Prefill Verify concurrently with Decode, bringing total overhead to 18% over the unverified baseline.

TABLE IV
EVIDENCE REPORT LATENCY (MS) OVER 100 CALLS. AMD-SP SIGNING DOMINATES; THE TIGHT SPREAD (MEDIAN WITHIN 0.03 MS OF MEAN, $\sigma = 0.17$ MS) INDICATES NEGLIGIBLE CONTENTION.

Mean	Median	Std	Min	Max	P95	P99
18.75	18.72	0.17	18.35	19.28	19.07	19.28

TABLE V
PREFILL VERIFICATION ABLATION ON QWEN3-1.7B AT $(T, G) = (256, 512)$. SAFE = DETECTS A TAMPERED GPU. OVERHEAD IS VS. THE UNVERIFIED GPU PREFILL (30 MS); SPEEDUP IS VS. THE PREVIOUS ROW.

Variant	Safe	Wall (ms)	Overhead *vs. GPU*	Speedup *vs. prev.*
Unverified GPU	✗	30	–	–
CPU-only	✓	5,103	172×	–
Slalom-style	✓	1,350	45×	3.8×
Full-pass, serial	✓	1,033	34×	1.3×
Pipelined	✓	229	7.7×	4.5×
Pipelined + packed	✓	214	7.2×	1.07×

D. Freivalds Repetition Sensitivity

After isolating the system mechanisms, we vary the verifier's main soundness parameter: the Freivalds repetition count k. The full system uses k=1 by default. Increasing k repeats each

TABLE VI
DECODE VERIFICATION ABLATION ON QWEN3-1.7B AT $(T, G) = (256, 512)$. EACH ROW ADDS ONE MECHANISM ON TOP OF THE BEST PREFILL VARIANT. SAFE = DETECTS A TAMPERED GPU. DECODE IS THE GPU GENERATION PHASE; TOTAL INCLUDES ALL CRITICAL-PATH PHASES. OVERHEAD IS VS. THE UNVERIFIED GPU BASELINE.

Variant	Safe	Decode (ms)	Total (ms)	Overhead *vs. GPU*
Unverified GPU	✗	9,855	9,885	–
Per-token verify	✓	51,712	51,985	+426%
ARGUS	✓	10,842	11,637	+18%

TABLE VII
EFFECT OF FREIVALDS REPETITION COUNT k ON ARGUS OVERHEAD (QWEN3-1.7B, $(T, G) = (128, 128)$, n=10). BOTH PREFILL VER. AND DECODE VER. SCALE ROUGHLY LINEARLY WITH k, WHILE THE GPU PHASES (SUBSUMED INTO TOTAL) ARE UNCHANGED. OVERHEAD IS VS. THE UNVERIFIED GPU BASELINE (3,233 MS).

k	Prefill Ver. *(ms)*	Decode Ver. *(ms)*	Total *(ms)*	Overhead *vs. GPU*
1	230	329	4,260	+32%
3	345	500	4,667	+44%
5	465	685	4,866	+50%

matmul check with independent random vectors, reducing the false-accept probability while adding CPU verification work.

Table VII reports the cost for $k \in \{1, 3, 5\}$ on Qwen3-1.7B at $(T, G) = (128, 128)$. GPU execution is unchanged; only Prefill Ver. and Decode Ver. grow. End-to-end overhead increases from 32% at k=1 to 50% at k=5, showing that stronger Freivalds repetition remains within the same order of magnitude as unverified GPU inference.

VI. SECURITY ANALYSIS AND VALIDATION

This section analyzes ARGUS's security guarantees and validates them against measured executions and injected attacks. We first give the Freivalds soundness argument under secret verifier randomness and independent repetitions. We then calibrate floating-point tolerances from honest executions. Next, we evaluate the calibrated verifier against arithmetic corruptions that modify matmul outputs or weights. Finally, we discuss attacks outside a single arithmetic residual, including TOCTOU on the untrusted I/O path and token fabrication during decode.

A. Freivalds Soundness under Floating-Point Tolerance

The exact Freivalds check (§II-C) bounds a single check's false-accept probability at $1/2$ and k independent repetitions at 2^{-k}. In native floating point, honest residuals are nonzero and the equality test becomes a bounded-residual check against a calibrated tolerance ε. This subsection bounds the residual the adversary must produce to slip past that tolerance, and explains why the same bound depends on whether the corruption is diffuse or sparse.

For a matmul with honest output $Y = XA$ and adversarial output $\hat{Y} = Y + \Delta$, the Freivalds residual reduces to Δr, where $r \in \{-1, +1\}^m$ is the verifier's secret random vector. Consider any nonzero row $\boldsymbol{\delta}$ of Δ. The projected error $\boldsymbol{\delta} \cdot r = \sum_j \delta_j r_j$ has mean zero and variance $\|\boldsymbol{\delta}\|_2^2$, and for diffuse corruptions spread across many coordinates it is well approximated by $\boldsymbol{\delta} \cdot r \dot{\sim} \mathcal{N}(0, \|\boldsymbol{\delta}\|_2^2)$. The single-check miss probability under tolerance τ is therefore approximately

$$p_{\text{miss}} = \Pr[|\boldsymbol{\delta} \cdot r| \le \tau] \approx \Pr[|Z| \le \tau / \|\boldsymbol{\delta}\|_2], \; Z \sim \mathcal{N}(0, 1). \tag{2}$$

Large corruptions ($\|\boldsymbol{\delta}\|_2 \gg \tau$) are unlikely to fall inside the tolerance window; very small corruptions can be accepted, which is why ARGUS calibrates ε from honest floating-point noise (§VI-B) and evaluates detection using token divergence as the operational criterion (§VI-C). Repeating the check with k independent vectors yields independent miss probabilities, so $p_{\text{miss}}^{(k)} = (p_{\text{miss}})^k$.

Sparse corruptions. The Gaussian approximation in (2) fits diffuse errors but not sparse ones. A single-element perturbation $A_{ij}+\delta$ produces a deterministic residual $|\delta| \cdot \|X_{:,i}\|_2$, so detection depends on the perturbation magnitude and the input activation at the injection site rather than on a Gaussian tail. In the evaluated attacks, when this product falls within tolerance, the same perturbation is also too small to change any decoded token; the attack experiments (§VI-C) confirm this boundary empirically.

Freshness of test vectors. The bound above assumes that the GPU commits to its claimed output independently of the verifier's secret r. ARGUS generates r inside encrypted CVM memory and never exposes it to the GPU. The corresponding weight projections $r' = Wr$ for the committed weights are precomputed during initialization as a one-time setup cost.

B. Tolerance Calibration

The soundness argument above treats the Freivalds tolerance as a parameter. We instantiate it by profiling honest native-FP executions: the threshold must be loose enough to accept floating-point noise, but tight enough to leave little room for adversarial residuals. Because residuals differ by matmul role, we calibrate separate tolerances for projection and attention checks, then use the same thresholds in the attack validation of §VI-C.

Honest residual profile. Table VIII reports the per-role residual distribution on Qwen3-1.7B at $(T, G) = (512, 256)$, aggregated across all layers and honest inference runs. Projection roles (Q, K, V, O, gate, up, down) have a tight residual range: their maximum residual stays below 1.1×10^{-5}. The attention products are larger. QK reaches 5.41×10^{-4}, and QKV reaches 8.30×10^{-5}.

This gap follows the operation structure. Projection checks have fixed-width output dimensions, so their residual distribution is stable across sequence lengths. Attention products involve the cached key–value context. In particular, the QK check accumulates rounding differences across T_{kv} attended positions, so its random-signed residual grows approximately with $\sqrt{T_{\text{kv}}}$. Models with QK head normalization add another rounding step before the dot product; we account for this with a conservative normalization factor in the attention tolerance.

TABLE VIII

HONEST FREIVALDS RESIDUAL DISTRIBUTION BY MATMUL ROLE ON QWEN3-1.7B AT $(T, G) = (512, 256)$, 420 OBSERVATIONS PER ROLE. ATTENTION-SCORE ROLES (QK, QKV) PRODUCE RESIDUALS ROUGHLY AN ORDER OF MAGNITUDE ABOVE PROJECTION ROLES.

Role	Max	P99	P95	Median
Q_proj	1.06×10^{-5}	7.88×10^{-6}	5.64×10^{-6}	3.15×10^{-7}
K_proj	8.21×10^{-6}	7.60×10^{-6}	4.25×10^{-6}	2.87×10^{-7}
V_proj	1.02×10^{-5}	5.88×10^{-6}	4.29×10^{-6}	2.30×10^{-7}
QK	5.41×10^{-4}	1.81×10^{-4}	1.24×10^{-4}	5.30×10^{-5}
QKV	8.30×10^{-5}	3.47×10^{-5}	2.38×10^{-5}	3.13×10^{-6}
O_proj	2.97×10^{-6}	1.91×10^{-6}	1.19×10^{-6}	1.26×10^{-7}
Gate	1.21×10^{-6}	1.05×10^{-6}	8.05×10^{-7}	1.72×10^{-7}
Up	2.05×10^{-6}	9.92×10^{-7}	6.18×10^{-7}	1.23×10^{-7}
Down	8.38×10^{-6}	3.40×10^{-6}	1.86×10^{-6}	1.58×10^{-7}

Per-role tolerances. Based on the honest profile, we use two tolerance rules. Projection roles use a fixed threshold:

$$\varepsilon_{\text{proj}} = 10^{-4}.$$

The sequence-dependent attention products use a context-length scaled threshold:

$$\varepsilon_{\text{attn}}(T_{\text{kv}}) = C\sqrt{T_{\text{kv}}}\,(1 + \mathbf{1}[\text{qk_norm}])\,, \quad C = 7.5 \times 10^{-5}.$$

We apply this attention rule to QK and QKV; QK is the worst case in our measurements.

At $(T, G) = (512, 256)$, this gives a worst-case honest margin of $\varepsilon/\rho_{\max} \approx 7.7\times$ for the attention products, determined by QK, and $\approx 9.4\times$ for projection roles, determined by Q_proj. Across all honest runs in the calibration set, these thresholds produce zero false positives. The next subsection evaluates the same thresholds under injected corruptions.

C. Arithmetic Attack Detection

Using the calibrated per-role tolerances above, we inject arithmetic corruptions into matmul outputs or weights and measure whether ARGUS rejects the run before releasing the response. The experiments use Qwen3-1.7B at $(T, G) = (512, 256)$ and cover six attack classes, ranging from small weight perturbations to wholesale computation skipping.

Table IX reports three outcomes for each attack. **Detect** is the fraction of injected runs flagged by the verifier. **Diverged** is the fraction whose decoded token sequence differs from the honest baseline. **Caught/Div** measures the safety-critical case: among runs that change decoded tokens, how many are detected. The key result is that every token-diverging corruption is caught; missed detections occur only when the injected corruption does not change any decoded token.

Precision reduction. The GPU computes the correct matmul but truncates the returned output to k mantissa bits ($k \in \{4, 8, 12, 16\}$), modeling silent arithmetic downgrade. ARGUS detects all 40 runs, including k=16, which removes only 7 mantissa bits. None of these runs changes the decoded tokens, but the result shows that post-computation tampering with matmul outputs is visible to the Freivalds check.

TABLE IX

ATTACK DETECTION ON QWEN3-1.7B AT $(T, G) = (512, 256)$ USING THE PER-ROLE TOLERANCES OF §VI-B. DETECT = FRACTION OF RUNS FLAGGED. DIVERGED = FRACTION WHOSE DECODED TOKENS DIFFER FROM THE HONEST BASELINE. CAUGHT | DIV = OF THE DIVERGED RUNS, FRACTION DETECTED. SAFE = EVERY DIVERGED RUN WAS DETECTED (DIVERGED $\subseteq$ DETECTED); MISSED DETECTIONS OCCUR ONLY WHEN THE PERTURBATION IS TOO SMALL TO CHANGE ANY DECODED TOKEN.

Attack	Param.	Detect	Diverged	Caught / Div	Safe
Precision reduction (all matmuls)					
	$k = 4$	10/10	0/10	–	✓
	$k = 8$	10/10	0/10	–	✓
	$k = 12$	10/10	0/10	–	✓
	$k = 16$	10/10	0/10	–	✓
Single-element perturbation					
	$\delta = 10^{-3}$	6/45	0/45	–	✓
	$\delta = 10^{-1}$	34/45	1/45	1/1	✓
	$\delta = 1$	45/45	3/45	3/3	✓
Row noise (single row)					
	$\sigma = 10^{-3}$	39/45	0/45	–	✓
	$\sigma = 10^{-2}$	45/45	5/45	5/5	✓
Weight quantization (full model)					
	INT8	10/10	10/10	10/10	✓
	INT4	10/10	10/10	10/10	✓
Weight quantization (single layer)					
	INT8	10/10	10/10	10/10	✓
	INT4	10/10	10/10	10/10	✓
Selective matmul skip					
	all sites	60/60	60/60	60/60	✓
Full layer skip					
	all layers	15/15	15/15	15/15	✓

Single-element perturbation. One weight element W_{ij} is shifted by $\delta \in \{10^{-3}, 10^{-1}, 1\}$, modeling a surgical weight corruption. Detection rises from 6/45 at δ=10^{-3} to 34/45 at δ=10^{-1} and 45/45 at δ=1. This reflects how a surgical perturbation interacts with the calibrated tolerance: at δ=10^{-3}, no run actually changes a decoded token, although 6/45 trials are still detected. Every perturbation that does change a decoded token is detected at every magnitude.

Row noise. One row of a weight matrix is perturbed by i.i.d. Gaussian noise with $\sigma \in \{10^{-3}, 10^{-2}\}$, modeling distributed firmware or arithmetic corruption. At $\sigma = 10^{-2}$, detection is 45/45 and all five token-diverging runs are caught. At $\sigma = 10^{-3}$, detection is 39/45 and no run changes decoded tokens, so the missed detections remain below the output-changing regime.

Weight quantization. All weights in the target scope are round-tripped through a lower precision (INT8 or INT4), modeling a GPU that secretly operates on cheaper representations. Detection is 100% under both full-model and single-layer scopes, and every run changes decoded tokens. FP16 quantization is not detected because the checkpoint weights are natively BF16 (7-bit mantissa); FP16 offers higher precision (10-bit mantissa), so the round-trip preserves all original bits. This gap is benign: FP16 quantization does not change any decoded token, so there is no output corruption for the verifier to detect.

Selective matmul skip. A single matmul output is replaced with a zero tensor. Detection is 60/60 across injection sites: every transferred matmul output is Freivalds-checked, so the skipped computation fails its own check. At non-final layers, the cross-layer residual consistency check independently rejects the corruption, providing a second detection path for the same attack.

Full layer skip. All matmul outputs within one layer are zeroed. Detection is 15/15 across layer positions. The scheduled Freivalds checks reject the zeroed matmul outputs, and at non-final layers the cross-layer residual consistency check rejects them independently.

Across all six attack classes, every corruption that changes a decoded token is detected, with zero false positives across the honest calibration runs (§VI-B). The calibrated tolerances therefore satisfy both ends of C3: tight enough to reject every output-changing arithmetic corruption we inject, and loose enough to never reject an honest native-FP execution.

D. Architectural and Token-Selection Defenses

Two further attacks fall outside a single Freivalds residual: a TOCTOU attack on the untrusted I/O path, and a token-fabrication attack on the binding between verified logits and emitted tokens. The IngressEngine handles the former by construction; token-selection verification handles the latter: greedy decoding is checked by default, and top-k membership is checked when the serving policy commits to top-k sampling.

TOCTOU. A time-of-check-to-time-of-use attack would modify a tensor in the SWIOTLB bounce buffer after the GPU writes it but before the verifier depends on it. The IngressEngine (§IV-B) closes this window: each frame is copied into encrypted TEE memory exactly once, and all subsequent verification reads only the encrypted copy. Tampering before the copy is caught by the downstream Freivalds check.

Token fabrication. Layer-level Freivalds checks alone do not bind the emitted tokens to the verified logits: an adversary could substitute decoded tokens while every such check still passes. ARGUS therefore checks token selection against the verified logits (§IV-F). By default, it verifies deterministic greedy decoding: each emitted token must equal the argmax of the verified logits at its position, and a mismatch rejects the request. ARGUS also checks token membership under a committed top-k sampling policy: Decode Verify already verifies the lm-head logits, and the verifier extracts the top-k indices from those verified logits and rejects any emitted token outside the set. Table X exercises the top-k check by replacing one decoded token at three positions across five prompts (15 runs total): detection is 15/15 with no positional blind spot. Under top-k sampling, tokens chosen from *inside* the verified top-k are undetectable by construction; we discuss this boundary in §VII.

TABLE X
TOKEN-SUBSTITUTION DETECTION (QWEN3-1.7B, T=512, G=256, TOP-k=50). ONE RANDOM TOKEN REPLACES THE INDICATED POSITION; FIVE PROMPTS PER ROW.

Position	Gen idx	Detect	$\rho_{\text{top-}k}$	Overhead
First	0	5/5	0.025–0.049	57–77 ms
Middle	128	5/5	0.028–0.035	143–183 ms
Last	255	5/5	0.028–0.035	143–458 ms

VII. LIMITATIONS

ARGUS is built and evaluated on a specific point in the design space: eager attention, dense transformers, single-step decoding, batch size 1. This section discusses the configurations the current prototype does not yet cover and how the verification architecture extends to them.

Fused attention kernels. Production LLM serving widely uses fused attention kernels such as FlashAttention, which collapse QK, softmax, and QKV into a single GPU kernel without materializing the intermediate products. Our prototype uses eager attention because it exposes those intermediates as separate tensors that the verifier can hook (§V-A); fused kernels by default do not, so the current implementation does not support them. This is not a fundamental obstacle: the per-layer pipeline only needs the same matmul outputs that the eager path already exports, namely $\widehat{QK}$ and $\widehat{QKV}$. Supporting a fused kernel therefore reduces to exposing those two tensors at the kernel boundary, either through a kernel-source modification or a side-channel export, and the rest of the verifier is unchanged.

Mixture-of-experts. Mixture-of-experts (MoE) layers route each token to a subset of expert MLPs, replacing a single MLP block with a dispatched set of per-expert matmuls. Our prototype evaluates dense transformers and does not yet implement an MoE adapter, so MoE deployments are unsupported in the current build. The matmul portion is, however, structurally identical to the dense case: each activated expert performs the same gate, up, and down projections, and the per-layer pipeline verifies them with the same Freivalds checks once they are exported. The new component is the routing decision itself, which selects which experts a given token visits. We can address this by layering a separate verification step on top of ARGUS that binds the routing decision to the verified inputs, rather than redesigning the verifier.

Speculative decoding. Speculative decoding lets a small draft model propose K candidate tokens that a target model verifies in one $(K+1)$-token forward pass. The draft model is untrusted by construction: its proposals are accepted or rejected based on the target model's logits, so only the target pass requires Freivalds verification. That pass produces the same rectangular attention shape the per-layer pipeline already handles, and the acceptance logic runs inside the CVM on verified logits from the Freivalds-checked LM-head projection (§IV-F), so the core verification path is unchanged. The engineering gaps are dynamic frame sizing (the number of accepted tokens varies per round), KV-cache rollback for rejected suffixes, and per-round orchestration of the verification pipeline. We consider these integration tasks rather than design changes and leave them to future work.

TABLE XI
CLOSEST TEE-SHIELDED/OFFLOADED SYSTEMS RELATED TO ARGUS. ROWS ARE GROUPED BY TARGET WORKLOAD (CNN/DNN ABOVE, LLM BELOW).

System	Confidentiality	Integrity	Decode	Decode mode	Arithmetic	Reported perf. (explicit baseline)
CNN / DNN systems						
Slalom [10]	input only	Freiv.	✗	–	finite-field	6–20× throughput vs SGX-only
SOTER [11]	model + IR	fingerprints	✗	–	native FP	up to 72.6% lower latency vs TEE-shielding
ShadowNet [27]	model	–	✗	–	scaled / fixed-pt	1.34–1.39× Slalom runtime[b]
GroupCover [28]	model	Freiv.	✗	–	native FP	$<19\%$ overhead vs unverified model
Tempo [29]	input + model	MAC-style	✗	–	native FP	3.09–4.92× speedup vs secure TEE
TSQP [30]	model	fingerprints	✗	–	quantized	1.01–1.02× Slalom runtime[b]
LLM systems						
TransLinkGuard [31]	model	–	✓	sync	native FP	negligible; $<0.1\%$ extra FLOPs
KV-Shield [32]	KV only[c]	–	✓	sync	native FP	$\sim$5 min for >20-layer model
CoreGuard [33]	model	–	✓	sync	native FP	$<1.17\times10^{-3}\%$ TEE overhead
ArrowCloak [18]	model	prior Freiv.[a]	n/e[d]	–	native FP	1.42–1.50× Slalom runtime
ARGUS (this work)	–	**Freiv.**	✓	**concurrent**	native FP32	18–74% overhead in balanced workloads vs unverified GPU

[a] ArrowCloak reuses prior TSLP Freivalds-style integrity checks; its main contribution is stronger weight obfuscation rather than a new integrity protocol.
[b] ShadowNet and TSQP are reported here using the real-device comparison in the ArrowCloak paper, which normalizes both against Slalom.
[c] KV-Shield protects user KV-cache privacy rather than outsourced model weights.
[d] ArrowCloak's evaluation covers single-forward-pass classification; autoregressive decode with KV-cache management under obfuscation is not evaluated.

Scale. Our evaluation uses a single GPU, batch size 1, and models up to 8B parameters; larger configurations exceed the memory of our test platform, so we have not measured these regimes empirically. The verification cost does not scale uniformly with model size. Model depth affects only the projection registry, which stores one precomputed Wr per matmul, so deeper models extend the per-layer pipeline by a constant per-layer cost rather than changing its structure. The hidden dimension is the more consequential parameter, since it sets the size of each transferred frame and therefore the per-layer D2H and trusted-copy cost. Requests can be checked separately, so multi-request batches admit the same per-layer pipeline; we expect amortizing per-frame overhead across a batch to reduce relative overhead further, and leave the empirical study to future work.

Multi-GPU inference. The prototype does not support multi-GPU inference. Extending it would require checking the cross-device transfers and collective operations used by model parallelism, binding intermediate values to the same request and model partition, and coordinating verification across devices.

VIII. RELATED WORK

Prior approaches to verifying GPU-offloaded inference each fail at least one of (i) keeping the GPU outside the TCB, (ii) running the committed model in native floating point, and (iii) staying within the same order of magnitude as unverified GPU inference. This section discusses related work and summarizes a comparison to TEE-shielded and offloaded designs in Table XI.

GPU confidential computing. GPU vendors offer firmware-level encryption and attestation (e.g., NVIDIA Confidential Computing) that extends the TEE boundary to include the GPU, removing the need for CPU-side verification. However, this support is currently limited to a narrow set of datacenter GPUs (e.g., H100, H200), excluding most deployed hardware.

Verification without TEE shielding. A separate line of work checks GPU arithmetic without TEE-shielded partitioning. ABFT [34] attaches deterministic checksums to matmuls to detect single-processor faults; ALBERTA [35] extends this approach to the GEMM layers of vision transformers, augmenting it with a confidence-based floating-point threshold to handle FP non-associativity. Because the checksum structure is public, an adaptive adversary can craft a perturbation that corrupts a matmul output and its checksum jointly to evade detection. DiFR [36] takes the opposite extreme: its Token-DiFR mode re-executes the forward pass on a trusted reference implementation under a shared sampling seed, detecting systematic provider deviations (e.g., quantization, sampling bugs) by accumulating token-level divergence over hundreds to thousands of spot-checked outputs at zero provider-side overhead. ARGUS sits between the two: it exploits the per-matmul $d/2$ cost asymmetry that both Freivalds [25] and ABFT use, but keeps the random projection r secret inside the CVM for checking errors chosen independently of the secret projections, and unlike DiFR runs inline on every query rather than amortizing over batches, achieving 18–74% overhead on balanced workloads.

TEE-shielded LLM partition (TSLP) systems. Slalom [10], SOTER [11], ShadowNet [27], TransLinkGuard [31], Tempo [29], TSQP [30], CoreGuard [33], and ArrowCloak [18] protect model confidentiality by offloading obfuscated weights to an untrusted GPU while keeping non-linear operations inside a CPU TEE. The TEE de-obfuscates the inputs, applies the non-linear function, re-obfuscates, and returns the result to the GPU for the next linear layer. Several of these systems include Freivalds-style integrity checks on the GPU's linear computations. Slalom runs the check in finite-field arithmetic, so it sidesteps floating-point non-determinism but quantizes the computation. GroupCover [28] and Tempo run the check in native floating point but do not address how to separate inherent FP noise from adversarial corruption; the tolerance threshold is either left implicit or not discussed. ArrowCloak reuses prior TSLP Freivalds checks; its main contribution is a stronger weight-obfuscation scheme based on approximate LWE. Among these systems, only TransLinkGuard [31] and CoreGuard [33] address autoregressive LLM decoding, but both

require the TEE to participate synchronously on every token's critical path. ARROWMATCH [18] breaks prior lightweight obfuscation schemes, including permutation and scaling, with $>98\%$ accuracy by exploiting direction similarity to the public base model. ArrowCloak reduces this leakage toward the $1.10\times$ black-box baseline, but still requires an unmeasured TEE round trip per attention layer. These systems are complementary to ARGUS: TSLP targets confidentiality of offloaded weights, whereas ARGUS targets integrity against an adversarial GPU.

Table XI summarizes the comparison to the closest TEE-shielded/offloaded systems for LLMs and DNNs. The recent literature mixes systems that protect model or input confidentiality with systems that directly verify GPU arithmetic, so we separate those properties in the table. "Decode" refers to autoregressive LLM decoding; "Decode mode" distinguishes *synchronous* (TEE on the critical path of every decode token) from *concurrent* (TEE verifies asynchronously, off the critical path). ARGUS targets online integrity of every LLM matmul under native FP32 while keeping the GPU outside the TCB.

IX. CONCLUSION

Cloud LLM inference gives clients little request-level evidence that a GPU faithfully executed the committed model. ARGUS addresses this gap by treating the GPU as untrusted and verifying GPU-produced results inside a CVM, with practical overheads enabled by minimal transfers, per-layer pipelining, and batched decode verification. Across six FP32 models from 1B to 8B parameters, ARGUS adds 18–74% overhead on balanced workloads while detecting all token-changing corruptions in our attack suite with zero false positives.

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd for their valuable feedback and guidance. This work was supported in part by the National Science Foundation (NSF) under Grant No. OAC-2613950.

REFERENCES

[1] OpenAI, "OpenAI API reference," https://developers.openai.com/api/reference/overview, 2026, accessed 2026-05-06.
[2] Google, "Gemini API documentation," https://ai.google.dev/gemini-api/docs, 2026, accessed 2026-05-06.
[3] Runpod, "Runpod cloud GPUs and pricing," https://www.runpod.io/pricing, 2026, accessed 2026-05-06.
[4] Lambda, "Lambda GPU cloud," https://lambda.ai/service/gpu-cloud, 2026, accessed 2026-05-06.
[5] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling next-gen LLM applications via multi-agent conversations," in *First Conference on Language Modeling*, 2024.
[6] H. Sun, J. Li, and H. Zhang, "zkLLM: Zero knowledge proofs for large language models," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.
[7] T. Liu, X. Xie, and Y. Zhang, "zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2968–2985.
[8] Z. Ghodsi, T. Gu, and S. Garg, "SafetyNets: Verifiable execution of deep neural networks on an untrusted cloud," *Advances in Neural Information Processing Systems*, vol. 30, 2017.
[9] NVIDIA, "Confidential computing on NVIDIA H100 GPUs for secure and trustworthy AI," https://developer.nvidia.com/blog/confidential-computing-on-h100-gpus-for-secure-and-trustworthy-ai/, 2023, accessed 2026-05-07.
[10] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," in *International Conference on Learning Representations (ICLR)*, 2019, arXiv:1806.03287.
[11] T. Shen, J. Qi, J. Jiang, X. Wang, S. Wen, X. Chen, S. Zhao, S. Wang, L. Chen, X. Luo *et al.*, "{SOTER}: Guarding black-box inference for general neural networks at the edge," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 723–738.
[12] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song, "Occlumency: Privacy-preserving remote deep-learning inference using SGX," in *The 25th Annual international conference on mobile computing and networking*, 2019, pp. 1–17.
[13] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "DarkneTZ: towards model privacy at the edge using trusted execution environments," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 161–174.
[14] Intel, "Intel software guard extensions (Intel SGX)," https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, 2024, accessed 2026-05-07.
[15] Advanced Micro Devices, "AMD SEV-SNP: Strengthening VM isolation with integrity protection and more," AMD, White paper, 2020. [Online]. Available: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf
[16] Intel Corporation, "Intel trust domain extensions (Intel TDX) module base architecture specification," Intel, Tech. Rep., 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html
[17] Arm Limited, "Arm confidential compute architecture (CCA) documentation," https://developer.arm.com/documentation, 2026, accessed 2026-05-07.
[18] P. Wang, B. Dong, Y. Cai, Z. Zhang, J. Liu, H. Xue, Y. Wu, Y. Zhang, and Z. Zhang, "Game of arrows: On the (in-)security of weight obfuscation for on-device TEE-shielded LLM partition algorithms," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025.
[19] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," in *AMD White Paper*, 2016.
[20] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel TDX demystified: A top-down approach," in *ACM Computing Surveys*, 2024.
[21] The Linux Kernel Authors, "Software IO TLB: Linux kernel documentation," https://www.kernel.org/doc/html/latest/core-api/swiotlb.html, 2024.
[22] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th symposium on operating systems principles*, 2023, pp. 611–626.
[23] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of machine learning and systems*, vol. 5, pp. 606–624, 2023.
[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
[25] R. Freivalds, "Probabilistic machines can use less running time," in *IFIP Congress*, 1977, pp. 839–842.
[26] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995.
[27] Z. Sun, R. Sun, C. Liu, A. R. Chowdhury, L. Lu, and S. Jha, "ShadowNet: A secure and efficient on-device model inference system for convolutional neural networks," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1596–1612.
[28] Z. Zhang, N. Wang, Z. Zhang, Y. Zhang, T. Zhang, J. Liu, and Y. Wu, "GROUPCOVER: A secure, efficient and scalable inference framework for on-device model protection based on TEEs," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.
[29] R. Xu and Z. Fang, "Tempo: Confidentiality preservation in cloud-based neural network training," in *International Joint Conference on Neural Networks (IJCNN)*, 2024, pp. 1–10.
[30] Y. Sun, G. Xiong, J. Liu, Z. Liu, and J. Cui, "TSQP: Safeguarding real-time inference for quantization neural networks on edge devices," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 2114–2132.

[31] Q. Li, Z. Shen, Z. Qin, Y. Xie, X. Zhang, T. Du, S. Cheng, X. Wang, and J. Yin, "TransLinkGuard: Safeguarding transformer models against model stealing in edge deployment," in *Proceedings of the 32nd ACM International Conference on Multimedia (MM)*, 2024, pp. 3479–3488.
[32] H. Yang, D. Zhang, Y. Zhao, Y. Li, and Y. Liu, "A first look at efficient and secure on-device LLM inference against KV leakage," in *Proceedings of the 19th Workshop on Mobility in the Evolving Internet Architecture (MobiArch)*, 2024, pp. 13–18.
[33] Q. Li, Y. Xie, T. Du, Z. Shen, Z. Qin, H. Peng, X. Zhao, X. Zhu, J. Yin, and X. Zhang, "CoreGuard: Safeguarding foundational capabilities of LLMs against model stealing in edge deployment," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2025, arXiv:2410.13903.
[34] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.
[35] H. Liu, G. Singh, P. Filipiuk, and S. K. S. Hari, "ALBERTA: ALgorithm-Based error resilience in transformer architectures," *arXiv preprint arXiv:2310.03841*, 2023, nVIDIA Research.
[36] A. Karvonen, D. Reuter, R. Rinberg, L. Marks, A. Garriga-Alonso, and K. Warr, "DiFR: Inference verification despite nondeterminism," *arXiv preprint arXiv:2511.20621*, 2025.

APPENDIX A
FULL PER-PHASE BREAKDOWN

Table XII expands the compact Table I into the wall-clock components of each ARGUS run on Qwen3-4B, including configurations omitted from the in-text version. Pf. fwd is the prefill GPU forward pass with hook capture enabled. Pf. ver (Prefill Ver.) is the CPU-side Freivalds check for prefill, which runs concurrently with decode and is hidden from the wall-clock. Decode is the GPU decode pass with hooks disabled. Dc. ver (Decode Ver.) is the Decode Verify pass.

APPENDIX B
PER-MODEL OVERHEAD ACROSS CONFIGURATIONS

Table XIII reports the full sensitivity sweep visualized in Fig. 4.

USE OF GENERATIVE AI

Scope of use. Large language models were employed solely to refine the English prose of this paper, including grammar, word choice, and sentence structure. Every revision was reviewed and approved by the authors.

TABLE XII

FULL PER-PHASE BREAKDOWN ON QWEN3-4B (MS). T = PROMPT LENGTH, G = GENERATION LENGTH. PF. VER (PREFILL VER.) RUNS CONCURRENTLY WITH DECODE AND IS HIDDEN FROM THE WALL-CLOCK.

		GPU baseline		ARGUS **phases**					
T	**G**	prefill	decode	Pf. fwd	Pf. ver	decode	Dc. ver	**TEE total**	**Overhead**
64	128	30	3,125	154	178	3,492	325	4,130	+31%
64	256	29	6,220	176	201	6,886	556	7,779	+24%
64	512	30	12,393	200	230	13,576	1,295	15,285	+23%
64	1024	29	23,757	307	329	27,035	3,428	31,013	+30%
128	128	41	2,878	162	204	3,455	304	4,094	+40%
128	256	41	6,194	194	229	6,902	588	7,858	+26%
128	512	41	12,167	228	266	13,581	1,365	15,347	+26%
128	1024	42	24,867	370	424	27,547	3,638	31,887	+28%
256	128	58	3,194	296	345	3,486	325	4,284	+32%
256	256	58	6,172	345	397	6,923	642	8,079	+30%
256	512	58	12,497	401	456	13,917	1,654	16,363	+30%
256	1024	58	22,847	530	604	27,977	5,454	34,736	+52%
512	128	93	3,103	685	786	3,567	440	4,891	+53%
512	256	94	6,144	712	828	7,027	665	8,642	+39%
512	512	94	12,633	751	864	13,768	1,653	16,436	+29%
512	1024	94	24,679	836	961	27,479	4,842	33,625	+36%
1024	128	190	3,130	1,911	2,177	3,654	684	6,533	+97%
1024	256	191	5,921	1,798	2,071	7,085	965	10,115	+65%
1024	512	192	12,046	1,872	2,136	13,607	1,840	17,744	+45%
1024	1024	192	25,228	1,950	2,260	27,286	5,115	35,165	+38%

TABLE XIII

ARGUS OVERHEAD (%) VS. GPU BASELINE ACROSS MODELS AND (T, G) CONFIGURATIONS. – DENOTES CONFIGURATIONS THAT EITHER DID NOT FIT IN MEMORY (GPU OR CPU TOO SMALL) ON OUR TEST MACHINE, OR WERE NOT MEASURED FOR THAT MODEL.

T	**G**	**Qwen3-1.7B**	**Qwen3-4B**	**Qwen3-8B**	**GPT-2-XL**	**Llama-3.2-1B**	**Llama-3.2-3B**
64	128	40	31	41	51	59	36
64	256	23	24	34	46	41	30
64	512	26	23	25	38	36	31
64	1024	18	30	–	–	48	34
64	2048	24	–	–	–	65	–
128	128	28	40	36	50	73	42
128	256	21	26	32	35	56	34
128	512	20	26	30	29	47	30
128	1024	23	28	–	–	51	37
128	2048	38	–	–	–	69	–
256	128	47	32	41	38	74	38
256	256	29	30	32	48	40	28
256	512	21	30	29	34	40	28
256	1024	24	52	–	–	43	39
256	2048	34	–	–	–	56	–
512	128	49	53	54	54	85	54
512	256	30	39	42	38	43	40
512	512	19	29	–	30	45	32
512	1024	30	36	–	–	55	32
512	2048	22	–	–	–	61	–
1024	128	83	97	–	–	114	104
1024	256	45	65	–	–	83	67
1024	512	31	45	–	–	63	38
1024	1024	23	38	–	–	55	43
1024	2048	38	–	–	–	66	–

APPENDIX C
ARTIFACT APPENDIX

This appendix describes how to install and evaluate ARGUS. The artifact supports both a conventional VM with an NVIDIA GPU and a CPU confidential VM (CVM) with a GPU. Both paths exercise the same workloads, verification code, and paired-overhead protocol. AMD SEV-SNP hardware is needed only for CVM-specific and attestation experiments; contact the authors to arrange access to a validated environment.

A. Description & Requirements

The artifact includes the implementation, native CPU verifier, benchmark runners, experiment scripts, and result-generation utilities. The verifier checks GPU matrix multiplications and recomputes nonlinear operations; prefill verification overlaps GPU execution, and generated-token-chain verification follows decoding.

1) How to access: The evaluated artifact is archived at https://doi.org/10.5281/zenodo.22780367 and mirrored at https://github.com/keke1022/Argus-NDSS27-Artifact. Clone revision `ndss27-ae-v1` with its Git submodules. Model weights are obtained separately from Hugging Face.

2) Hardware dependencies: Hardware-independent tests need an x86-64 CPU. GPU experiments need a CUDA-capable NVIDIA GPU exposed as CUDA device 0 with sufficient free VRAM. Use multiple CPU cores for performance evaluation; the reference SEV-SNP CVM had 16 vCPUs, 54 GiB RAM, and an NVIDIA RTX PRO 6000. A single-core functional path is available through `artifact/run-single-core-full.sh`; its timings are not comparable to the multi-core paper results. Conventional VM measurements are expected to show comparable paired overhead, but absolute latency may differ.

3) Software dependencies: Ubuntu Linux, Python 3.12, `uv`, Git with submodules, CMake, Ninja, a C/C++ compiler, an NVIDIA driver, and a compatible CUDA toolkit are required. Python dependencies are pinned in `pyproject.toml` and `uv.lock`. The README lists validated versions and troubleshooting.

4) Benchmarks: The full suite covers GPT-2-XL, Llama-3.2-1B/3B, and Qwen3-1.7B/4B/8B in FP32. The recommended small workload uses Qwen3-1.7B with 64 prompt and 128 generated tokens. Tests cover performance, verification ablations, honest residuals, arithmetic corruption, and token substitution.

B. Artifact Installation & Configuration

From the artifact repository root, run:

```
uv sync --locked
./scripts/build_ggml.sh
uv run --locked python -m \
  argus.native_extension build \
  --cuda-arch 12.0
uv run --locked python -m \
  argus.native_extension check
```

The last command must print `Argus native verifier ready`. Replace `12.0` with the GPU's CUDA compute capability when necessary. The README covers model acquisition and CVM configuration.

C. Experiment Workflow

Run the initial functional check, then experiments E1–E3 below. Commands use the ordinary-VM configuration by default. Inside the reference CVM, enable CVM-specific checks with `CUDA_CONFIDENTIAL_COMPUTE_EMULATED=1`. Reported experiment times exclude model downloads.

D. Major Claims

The numerical results refer to the paper's reference SEV-SNP environment; other machines should reproduce the trends rather than exact absolute latency.

- **(C1) Practical verified inference.** Across six FP32 models (1B–8B), overhead is 18–74% on balanced workloads relative to unverified GPU inference. E1 checks paired baseline–Argus overhead; see Fig. 4, Table I, Table XII, and Table XIII.
- **(C2) Reduced prefill-verification overhead.** Pipelining and packed transfers reduce verification overhead. E2 checks prefill ablations; see Table V and the related decode ablation in Table VI.
- **(C3) Detection of output-changing corruption.** Calibrated thresholds accept honest executions and detect output-changing corruptions in the evaluated workloads. Token substitutions outside the verified top-k set are rejected. E3 checks these properties; see Table VIII, Table IX, and Table X.

E. Evaluation

Execute all commands from the artifact repository root.

1) Initial functional check: [1 human-minute + approximately 1 compute-minute]

[Preparation] Complete installation; no GPU or model weights are needed. *[Execution]* Run `artifact/run.shtest`. *[Results]* Expect `Result: PASSED`, confirming functional verifier logic and native components.

2) Experiment (E1): End-to-end overhead: [5 human-minutes + 2–4 compute-minutes for the representative run; several compute-hours for the multi-model campaign]

[Preparation] Complete the functional check and cache Qwen3-1.7B weights locally. *[Execution]* Run `artifact/run.shrepresentative`. It compares baseline and Argus on the same machine using FP32, $(T, G) = (64, 128)$, 10 warm-up pairs, and 10 measured pairs. For a broader campaign, run `artifact/run.shfull`; the runner selects workloads according to available GPU memory.

[Results] Raw measurements, the manifest, and paired comparisons are written to `results/ae/representative/`. The final `gate.log` line must read `Representative gate passed`: the median paired Argus/GPU latency ratio must be below 2.0, and normalized robust latency dispersion must be at most 0.15 for each mode.

A workload is *balanced* when $T/G \leq 2$. The representative run has 40% overhead on the reference machine. Across measured balanced workloads, overhead is 18–74%; prompt-heavy workloads ($T/G > 2$) report 45–114% in Table XIII.

Overhead should decrease as generation grows relative to the prompt. This scaled-down experiment exercises the full verification protocol, but gives an indicative paired overhead rather than an exact reproduction of reference-machine latency.

3) Experiment (E2): Prefill-verification ablation: [5 human-minutes + approximately 3–5 compute-minutes]

[Preparation] Cache Qwen3-1.7B and Qwen3-4B. The respective workloads are $(T, G) = (256, 512)$ and $(512, 512)$. *[Execution]* Run `experiments/tab_ablation_prefill/run.sh`. *[Results]* Expect two structured result files and corresponding ablation tables. Pipelining should reduce serial-verification latency. The ordering of CPU-only recomputation, Slalom-style, and full-pass serial verification is hardware-dependent; CPU-only may be faster. Packing has a smaller, platform-dependent effect, so the two pipelined variants may have similar latency.

4) Experiment (E3): Corruption detection: [5 human-minutes + 1–1.5 compute-hours]

[Preparation] Use the supplied calibrated thresholds and attack configurations; the arithmetic campaign uses Qwen3-1.7B at $(T, G) = (512, 256)$. *[Execution]* Run `experiments/tab_honest_noise/run.sh`, followed by `experiments/attack_detection/run.sh`.

[Results] The first runner regenerates Table VIII: each verifier role should have sufficient observations and honest residuals within its acceptance bound. Exact values depend on the GPU and software stack. The attack runner regenerates Table IX and Table X. Most token-changing corruptions should be detected, with computation-skipping attacks caught consistently. Small perturbations may evade detection within the numerical tolerance or when they do not change tokens. Substitutions outside the verified top-k set should be rejected at all tested generation positions, with no false positives on honest runs.