Zero knowledge verification for frontier AI training is possible

Pierre Peigné - Lefebvre
General-Purpose AI Policy Lab Paris
ppeigne@gpaipolicy-lab.org

Ky Nguyen
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France
ky.nguyen@lip6.fr

Paul Wang
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France
pwang@phare.normalesup.org

Abstract

Frontier AI governance frameworks increasingly use cumulative training compute as the primary criterion for designating high-impact models, but enforcement rests on self-reporting because no technical verification primitive for training exists. Any future international agreement on frontier AI faces the same problem at higher stakes: coordinated regulation of technologies with significant externalities has historically rested on technical verification, without which agreements are declaratory. Recent governance analyses judge zero-knowledge proofs a promising candidate but currently impractical at frontier scale [26, 4]. We argue the impracticality is paradigm-bound rather than fundamental, and propose a verification architecture for frontier dense pre-training combining a pre-committed training specification, inter-node network observations, and on-the-fly Merkle commitments of intermediate computation, verified through a *zero-knowledge Virtual Machine (zkVM)* with native BF16/FP32 precompiles. The proof checks the *actual floating-point computation the GPU performed* rather than a fixed-point approximation, and preserves model-architecture confidentiality through a private training specification. The protocol produces three proof types: a *genesis proof* at initialisation, *in-training step proofs* across the run, and *ex-ante attestations* enforcing policy-relevant claims as running invariants, turning the training record into a governance-enforceable artefact. We estimate a deployable proof of concept within approximately 36 months at single-digit-percent training-side overhead, against a six-to-ten-year cycle for verification-grade custom silicon. Thirteen open research and engineering problems are catalogued as a research agenda for external contribution.

1 Introduction

Training runs for frontier AI systems are increasingly at the centre of emerging governance frameworks that rely on claims the trainer makes about its run: what data was used, what procedure was followed, what compute was consumed. No technical verification primitive exists to check such claims today, and enforcement rests on trainer self-reporting. Any future international agreement that would govern the development of frontier AI faces the same problem at higher stakes [4, 25]. Past international agreements on technologies with significant externalities show that the credibility of

Preprint.

substantive commitments usually rests on the compliance architecture negotiated with them [1]. For the option of such an agreement to remain available to states on frontier AI, the verification primitive must exist before negotiations begin, not after. This paper proposes such a primitive.

A *verification primitive* for training must produce proof of faithful execution of the committed procedure, verifiable claims about the run (compute consumed, training regime, data-content filters), and privacy for the trainer, within a training-overhead budget low enough to be economically rational at frontier cost. Single-digit-percent overhead, not factor-of-two. Prior zero-knowledge (ZK) machine-learning systems (Section 2) are *monolithic*[2], and are confined to inference or LoRA fine-tuning, operate over finite-field proxies rather than native BF16/FP32 GPU execution, and plateau at $\sim$13B parameters. Alternative verification approaches based on dedicated hardware, such as TEE$^\diamond$-attested accelerators, HBM$^\diamond$-monitoring coprocessors [22], and NICs retrofitted with verification chiplets [23], face two structural obstacles independent of cryptographic feasibility. Their silicon is proprietary to a handful of vendors, a liability for international verification where trust cannot rest on any single firm's supply chain. The research-and-development cycle for verification-grade custom silicon (specification, tape-out, formal RTL verification, manufacturing, certification, and cluster-wide deployment) is realistically six to ten years before the primitive becomes available to regulators[3].

Our introductory discussion thus begs for answering the following research question: *Is verification for frontier AI training, in a zero-knowledge and efficient non-monolithic manner, possible?*

Our contribution. This paper answers the above question positively. We argue that a verification architecture is feasible at frontier scale, deployable within approximately 36 months: *(i)* A blueprint for zero-knowledge verification of frontier dense pre-training at single-digit-percent training-side overhead, roughly five orders of magnitude below the prevailing estimate [4], verifying actual BF16/FP32 hardware execution and preserving model-architecture confidentiality via a private `arch_spec`; *(ii)* A characterisation of the ex-ante attestation as a governance-enforceable primitive, with compute-threshold enforcement as the immediate use case[4]; *(iii)* A structured roadmap of thirteen independently-attackable open problems delineating the remaining work to reach a fielded verification primitive.

Our main perspectives and suggested open problems. From an *architectural* point of view, we circumvent the finite-field arithmetic constraint of prior zero-knowledge ML systems by anchoring the proof in three complementary objects: a *pre-committed training specification*, *inter-node network* observations by an auditor-aligned anchor (physical TAP$^\diamond$ or attested SmartNIC$^\diamond$), and *on-the-fly Merkle commitments* of intermediate computation. The anchors are verified through a generic zkVM equipped with native BF16/FP32 precompiles, so the proof checks the actual floating-point computation the GPU performed rather than a finite-field approximation of it. Our AI verification protocol employs a *genesis proof* at initialisation, *in-training step proofs* across the run, and *ex-ante attestations* enforcing policy-relevant claims as running invariants during training. We highlight that the phenomenom of random local-verifiability of "snapshots" implies global-verifiability for NP relations (a faithful execution in our case) is a long standing research subject in theoretical science, reflected via the celebrated *PCP theorem* (from the seminal [3] to recent results [1]). We extend this framework to AI training verification by architecturing the specifications, network observations, random sampling verification checkpoints/indices during the model training (Sections 3.1 and 3.2) that are proved via zkVM with precompiles and compatible with steps composition. The architecture is developed for frontier *dense pre-training* as a first target, because it is architecturally the simplest setting, accounts for the largest single block of frontier training compute, and is the root of trust for every post-trained or fine-tuned derivative. The architecture does not yet cover sparse mixture-

[1] Examples include IAEA safeguards under the NPT (material accountancy, on-site inspections, and environmental sampling since the 1997 Additional Protocol) and the OPCW regime under the CWC (industry declarations, routine inspections of declared facilities; the Article IX challenge mechanism has never been invoked). The Montreal Protocol is a partial case: state-level reporting under Article 7 from 1987, with the Non-Compliance Procedure added in 1992 and atmospheric science as a parallel independent input. The Biological Weapons Convention is the counterexample: in force since 1975 with no verification regime.

[2] That is, these prior ZK systems can be performed only *per sample* or *per step*, and not efficiently for a full run.

[3] Open-hardware alternatives such as flexible hardware-enabled guarantees [23] address the vendor-concentration problem but push the deployment timeline further by requiring international coordination on the hardware design itself.

[4] The EU AI Act (Art. 51) [12] sets a cumulative training-compute threshold around 10^{25} FLOPs for general-purpose AI models with systemic risk; this remains in force. The United States executive order 14110 [30] used 10^{26} FLOPs on similar lines before its revocation in 2025. Both regimes rely on trainer self-reporting.

of-experts, reinforcement-learning post-training, multi-datacentre training, or intra-node NVLink$^\diamond$ traffic. Each is an additive extension rather than a redesign. Relevant open research and engineering problems are catalogued in Section A.

2 The prior ZK-ML paradigm and its limits

ZK for inference. Recent work has applied zero-knowledge proofs to verify ML model inference. ZKML [8] builds an optimizing compiler from TensorFlow to halo2 ZK-SNARK circuits, proving inference for models up to GPT-2 (81M parameters) in approximately one hour. zkLLM [28] introduces specialized protocols for LLM inference (`tlookup` for non-arithmetic operations, `zkAttn` for the attention mechanism) with GPU-accelerated sumcheck proving, scaling to 13B parameters in under 15 minutes. South et al. [27] propose a general-purpose framework that converts any ONNX model into verifiable evaluation attestations via ezkl, demonstrating proofs across CNNs, LSTMs, and small transformers. These systems enable a model provider to prove that a committed set of weights produces a claimed output on a given input, without revealing the weights.

ZK for training and fine-tuning. Extending ZK beyond inference to the training process is more challenging, as it requires proving backward passes and parameter updates. VeriLoRA [19] is the first framework to achieve this for LoRA fine-tuning, proving forward propagation, backward propagation, and parameter updates for a single mini-batch on models up to 13B parameters (OPT-13B, $\sim$250 s proving time per step on A100). Earlier work on full training includes zkDL [29], which flattens CNN training into a single circuit architecture (FAC4DNN) for models with tens of millions of parameters.

Structural limitations. Recent governance analyses describe zero-knowledge proofs as a promising candidate for training verification in principle, but currently impractical at frontier scale [26, 4]. Shavit [26] notes that such techniques "cannot efficiently execute computationally intensive programs, like long sequences of gradient updates on billion-parameter models". Baker et al. [4] report an estimated $\sim 5 \times 10^5$ training overhead and the loss of model-architecture confidentiality. These assessments accurately characterise the existing ZK-ML paradigm. All systems in this paradigm share two structural constraints. First, they operate over finite fields, where arithmetic is *exact* and *associative*. This is a *structural requirement of SNARKs and sumcheck protocols*, not a design choice. Every model must therefore be converted to fixed-point arithmetic before proving. The proof verifies a *different* computation than what GPUs execute (BF16/FP32 with *non-associative floating-point* arithmetic and hardware-specific rounding). For inference this gap is acceptable; for training verification it is not, because the goal is to prove a specific sequence of GPU operations was performed as claimed. Second, proving cost makes monolithic proofs of training prohibitively inefficient: the fastest existing system (zkLLM) takes 15 minutes per forward pass at 13B, far from a frontier run involving millions of forward-backward steps at hundreds of billions of parameters across FSDP$^\diamond$/TP$^\diamond$/PP$^\diamond$-distributed clusters that no existing system models. Both constraints are properties of this paradigm, not of zero-knowledge verification per se. The architecture proposed below is designed to lift them.

	ZKML [8]	zkLLM [28]	ezkl [27]	VeriLoRA [19]	**Target**
Scope	Inference	Inference	Inference	Fine-tuning	**Pre-training**
Arithmetic	Finite field	Finite field	Finite field	Finite field	**Native BF16**
Max scale	81M	13B	250K	13B	**100-1000B (est.)**
Multi-step	Per-sample	Single pass	Per-sample	1 LoRA step	**Full run**
Distributed	✗	✗	✗	✗	✓
Proves HW exec.	✗	✗	✗	✗	✓

Table 1: Comparison of ZK-ML verification systems. Existing approaches prove computations over finite-field proxies of the model. The Target column states the architectural objectives of this proposal, justified in Section 3.1 and Section B; values are not yet measured.

3 Proposed solution

Scope. The architecture developed in this paper targets *dense pre-training*. Today's frontier spans dense models at several hundred billion parameters (e.g. Llama 3.1 at 405B) and sparse mixture-of-experts architectures approaching one trillion total parameters with a small fraction active per token

(e.g. DeepSeek-V3 at 671B, Kimi K2 at roughly 1T with ∼32B active per token). We begin with dense pre-training because it is architecturally the simplest setting, because every post-trained model is rooted in a pre-training run (if pre-training is not verifiable, nothing downstream is), and because pre-training remains the largest single block of training compute[5].

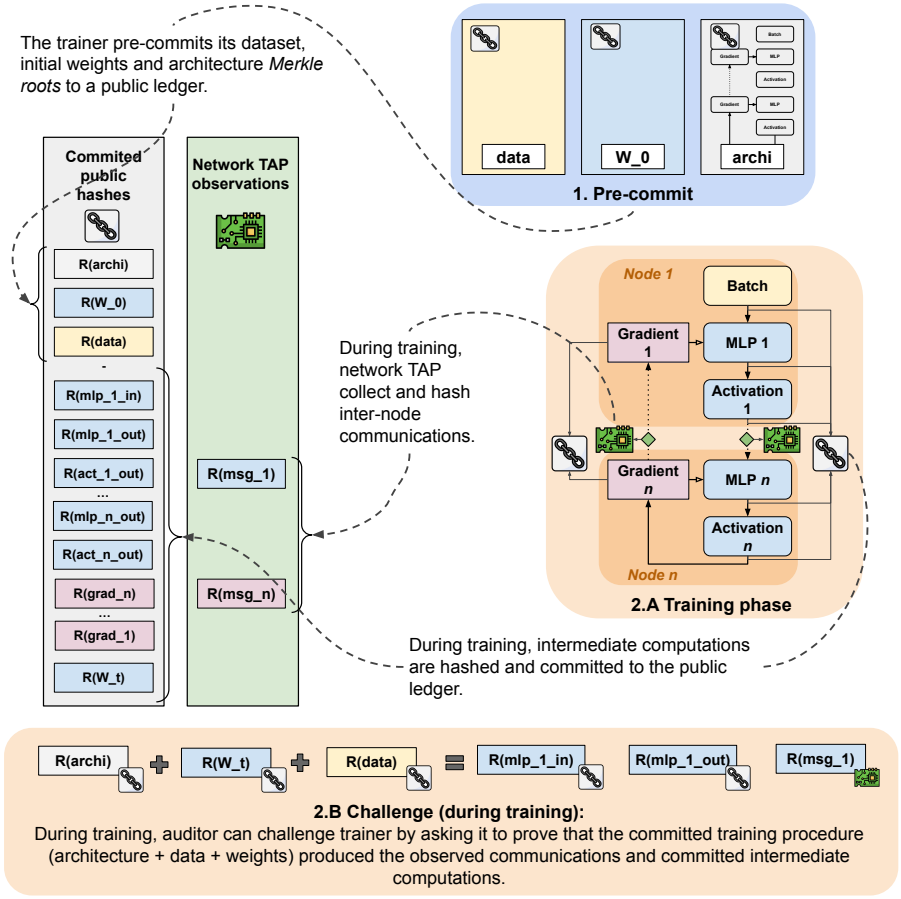

Figure 1: Overview of the verification protocol.

Overview. Figure 1 presents an overview of our verification architecture by combining three complementary trust anchors: *(I)* A pre-committed training procedure, model weights and dataset (Item **MOD. 2**), bound to the physical cluster via a challenge-response genesis proof (Protocol Item **P. 1**); *(II)* External network TAPs providing public hashes of ongoing inter-node communications, anchoring the training to independently observable data (Item **MOD. 4**); *(III)* Merkle roots of selected intermediate computations, committed on-the-fly (Item **MOD. 3**) and available for retrospective sampling challenges (Protocol Item **P. 2**).

Modification of current AI training procedures. We start by detailing modifications that we believe necessary for integration of our AI verification architecture.

MOD. 1 **(Enforcing determinism)** Verification requires bit-exact determinism so the proof checker can reproduce committed values. Standard GPU training is non-deterministic because floating-point reductions in cuBLAS$^\diamond$, NCCL$^\diamond$ collectives, and attention kernels depend on thread scheduling. At **hardware-level determinism**, per-operator kernel selection (FlashAttention-2 deterministic mode, pinned cuBLAS algorithms, NCCL NVLS$^\diamond$ reductions) achieves bit-exact determinism with 1.6–8.2% end-to-end overhead on Llama 7B FSDP (8×H100), dominated by the attention backward pass. Recent deterministic FlashAttention-3 scheduling [24] reduces this further; details in Section C. At **network-level determinism beyond FSDP**, FSDP's collectives are deterministic under NVLS. PP

[5]Two further regimes, (Section A, OP-10 for MoE and OP-11 for RL), are increasingly load-bearing for frontier capability: *sparse mixture-of-experts (MoE)*, and *reinforcement-learning (RL) post-training* (RLHF, RLAIF, rule-based RL for reasoning).

and EP$^\diamond$ reduce to per-layer and NVLS-deterministic all-to-all guarantees respectively. TP is the hard case: its intra-node allreduce over NVSwitch$^\diamond$ is *not* deterministic under NVLS above 128 MB, and deterministic NCCL configurations incur 64–89% bandwidth loss. A custom kernel exploiting TP's fixed topology is the proposed path; design in Section C.

MOD. 2 **(Pre-committing public Merkle roots)** Before training, the trainer publishes: $h_{\text{commit}} = H\big(H(\texttt{arch\_spec}) \parallel \text{MerkleRoot}(\text{dataset}) \parallel \text{MerkleRoot}(w_0)\big)$ Merkle paths enable membership proofs for individual batches or layer weights at logarithmic cost without revealing the full dataset or model. The genesis proof checks all three components; a continuity proof needs only paths for the relevant batch and layer weights.

MOD. 3 **(On-the-fly Merkle root computation)** A GPU concurrent stream computes Merkle roots of six tensors per layer at every step (layer input/output, Q, K, V, attention output) using the zkVM-native hash (Poseidon, precompile 3), published in a *per-step hash chain* for retrospective verification. When challenged, the trainer must produce values consistent with the published roots, which become immutable anchors once the chain is frozen by an auditor-signed terminal anchor before the audit seed is opened. Pure-HBM-read overhead is 0.2%; a 2-pass reduction kernel adds 1.6–3.3% (SM contention). Estimated production overhead with an optimised kernel: 0.5–1.5%. The stream can alternatively be offloaded to a SmartNIC/DPU$^\diamond$ or CPU (Section B).

MOD. 4 **(Network anchoring)** Without an independent witness, all commitments are trainer-produced, leaving open fabrication of internally consistent commitments for phantom computations. Both tiers commit a Merkle tree over inter-node collective payloads via SHA-256 (precompile 8, Table 3), enabling $\log n$-depth path verification at challenge time. At **Tier 1 (auditor-owned physical TAP)**, a passive auditor-installed device captures frames at wire speed, canonicalises payloads, and hashes them into the Merkle tree. Under pinned NCCL the wire sequence is a deterministic function of step inputs, so the trainer reconstructs the tree from a replay (Item **MOD. 1**); trust root is physical. At **Tier 2 (attested SmartNIC)**, signed DPU firmware exposes its root over an attested channel rooted in secure boot and device identity; trust root is cryptographic. Weaker than Tier 1 against firmware or supply-chain adversaries, but deployable on existing cluster infrastructure. **Tier 2** serves early adopters while **Tier 1** infrastructure is standardised; the long-term goal is uniform Tier 1, required by any regime holding against well-resourced adversaries. Both tiers observe inter-node traffic only; intra-node NVLink is invisible (hybrid FSDP+TP covered at the FSDP layer only). Intra-node coverage via GPU-resident trusted execution is discussed in Section A. The canonical tensor-to-wire mapping is the central open engineering problem; see Section A.1 and Section D.

MOD. 5 **(Weight storage rolling window)** The trainer stores a rolling window of weights at every step on NVMe/Lustre/GPFS arrays. At 405B parameters in BF16, each snapshot is $\approx$810 GB; 100 steps require 81 TB. When challenged on step N, the trainer loads weights directly and runs a single forward pass. Whole-run sampling requires either full retention or deterministic replay from durable checkpoints with committed replay metadata (optimizer, data-loader, and stochastic-component state; kernel/reduction-tree and precision metadata; checkpoint interval commitments; committed seed schedule). At frontier scale, 81 TB is modest relative to existing cluster storage.

Verification costs (genesis proof, sampling challenges) are borne by the proving system and detailed in Section 3.2. Total trainer-side overhead at Llama 3.1 405B scale on a \$100M budget[6] is in Table 2.

3.1 Architecture of the proving system - Necessary components

The proving system has four components: *a training specification* (private, known only to the trainer), *a zkVM* (run by the trainer that provides private hints from specification), *a proof checker* (compiled and given to the zkVM), and a set of *precompiles* (given access to the zkVM). Its central principle is that the zkVM never re-executes the training computation. When challenged on a step, the trainer loads stored weights, re-executes forward and backward on GPU, and provides intermediate tensors as hints; the zkVM only verifies hints are consistent with the Merkle roots committed during training.

[6]Published estimates for Llama 3.1 405B-class runs fall in the \$60M–\$120M range. The Llama 3 report [11] gives $\sim 3.8 \times 10^{25}$ FLOPs on 16,384 H100s; Epoch AI [9] and SemiAnalysis [21] place total costs in the same range.

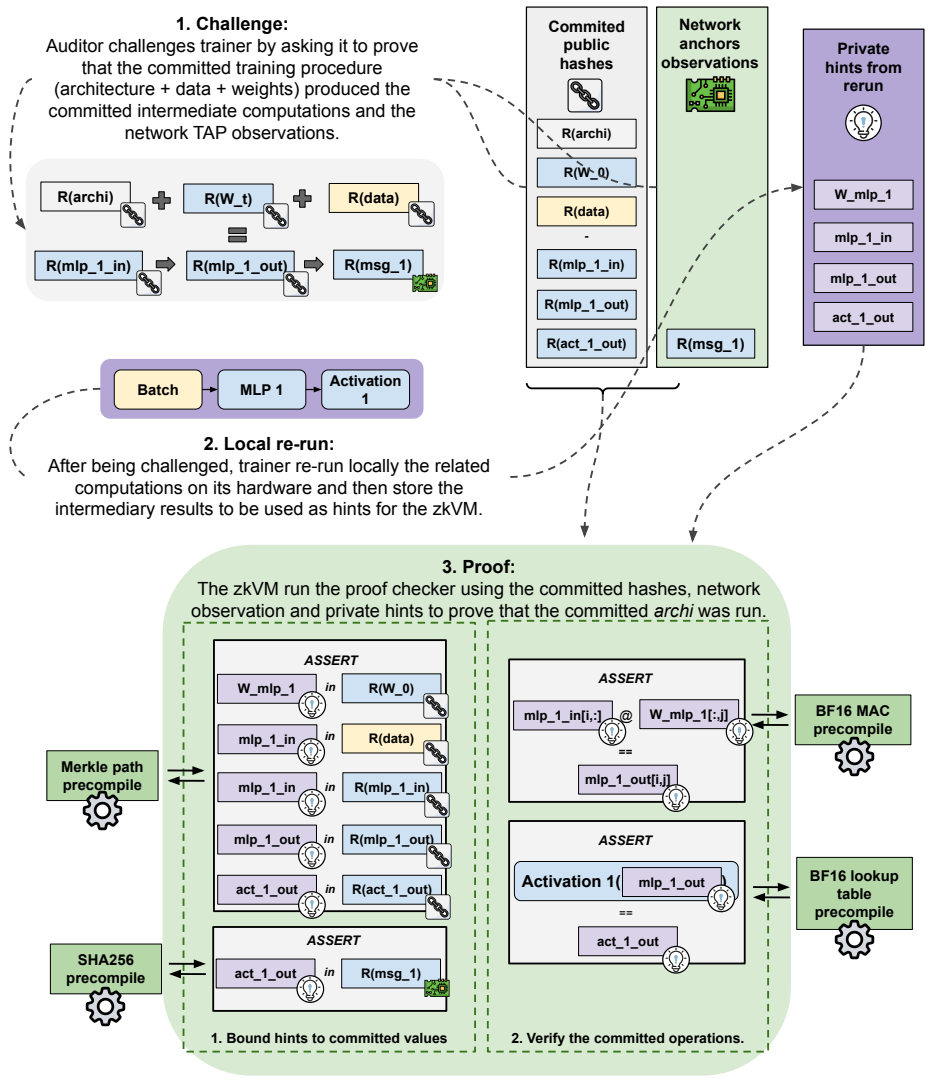

Figure 2: Proof generation from an in-training challenge. *(Top)* The auditor challenges the trainer to prove the committed procedure (architecture, weights, data) produced the committed intermediates and observed network messages. *(Middle)* The trainer re-runs the challenged operations locally and supplies the intermediate tensors as private hints to the zkVM. *(Bottom)* The zkVM verifies in two phases: **1.** Merkle path verification for each private hint, then **2.** random sampling to verify the bound hints.

Component	Cost	% of training
Determinism (**MOD. 1**, 1.6–8.2%)	\$1.6–8.2M	1.6–8.2%
Merkle computation (**MOD. 3**, ∼0.5–1.5%)	\$0.5–1.5M	0.5–1.5%
Weight storage (**MOD. 5**, 81 TB rolling)	negligible	<0.1%
Total (FSDP)	**\$2–10M**	**∼2–10%**

Table 2: Estimated training-side overhead at Llama 3.1 405B scale. The dominant cost is determinism (attention backward pass). FSDP+TP setups incur additional overhead until a custom TP allreduce kernel is deployed.

Two data categories enter the proof: **Public inputs** (Merkle roots) are committed during training on a concurrent GPU stream and linked into a per-step hash chain; they become immutable trust anchors when the chain is frozen by a auditor-signed terminal anchor before the audit seed is opened; **Private hints** (full tensor values) are produced by the host at challenge time by re-executing from stored weights and must be verified against the public roots. Verification proceeds in two layers: each hinted value is bound to its committed root via a *path check* (preventing fabricated hints), then the computation on bound values is verified using the appropriate *precompile* (preventing incorrect commitments).

The **training specification** `arch_spec` is a structured, machine-readable description of one complete training step. It declares: model architecture as typed operations per layer (GEMM$^\circ$s with dimensions, activations, normalisations); numerical precision (BF16 parameters, FP32 accumulators, IEEE 754 RNE$^\circ$); the distributed communication pattern (allgather, reduce-scatter); data loading (batch size, shuffling seed, master RNG seed[7], registered stochastic components); and tensor boundaries at which Merkle roots are committed. The `arch_spec` fully describes the procedure—no separate code commitment is needed; the genesis proof (Item **P. 1**) validates that GPU execution is consistent with it. The concrete structure of `arch_spec` is given in Section F. Commitment structure is in Item **MOD. 2**.

The **zk Virtual Machine** (zkVM) produces a succinct, non-interactive proof of correct execution. We require: application-defined precompiles (native constraint circuits callable from the guest), GPU-accelerated proof generation, and continuations with recursive composition[8]. The zkVM operates on a *hint-and-verify* model: the host (trainer's GPU) runs the full computation outside the proof and provides intermediate values as hints; the guest (proof checker inside the zkVM) does not re-execute, but verifies hints by calling precompiles against Merkle-committed values. Host computation contributes zero constraints; only the guest's verification logic does. This reduces cost by roughly five orders of magnitude versus full re-execution.[9] The pattern is standard in ZK-ML [18, 8, 28] and extends to training verification [13]. The same pattern handles the backward pass: transposed GEMMs via the MAC precompile, activation derivatives via the BF16 lookup (BF16 $\mathrm{GELU}'(x)$ is also a 2^{16}-entry table), and Adam's sqrt and division via FP32 nonlinear precompiles. This is what makes verifying *training*, not just inference, possible. We provide a complete walkthrough in Section E. The **proof checker** is a public, auditable program compiled to the zkVM's instruction set. It reads `arch_spec`, iterates over declared operations, and dispatches each to the appropriate precompile. The same binary works for any model—only `arch_spec` differs between a 10M-parameter prototype and a frontier trillion-parameter model. The `arch_spec` is a *private* input: the auditor sees only $H(\texttt{arch\_spec})$ as part of h_{commit}, preserving the trainer's IP. From *public metadata* only the number of layers (chain length) and approximate scale (storage footprint or regulatory filings) can be inferred—both already disclosed under the regulatory regimes this work serves. The auditor's trust surface reduces to a single public binary, the precompile specifications, and the zkVM implemented proof system's cryptographic soundness.

3.1.1 The precompiles

Precompiles are native constraint circuits hardwired into the zkVM's proof system, implementing specific numerical operations as compact constraint sets over the zkVM's native field $\mathbb{F}_p$. This differs

[7] Training RNG material is distinct from the auditor's audit seed: the former determines stochastic execution; the latter selects committed positions to open after the stream segment is frozen.

[8] We use `RISC Zero` as our reference implementation; the architecture is not coupled to this choice.

[9] Reduction factor $(m \cdot n/k) \cdot (c_{\text{sw}}/c_{\text{pre}})$, with $c_{\text{sw}} \approx 1{,}000$ RISC-V cycles per software-emulated MAC and $c_{\text{pre}} \approx 90$ constraints per native MAC; typically 10^4–10^6.

subtly from prior ZK-ML (cf. Table 1): prior work converts the *model's computation* to finite-field arithmetic, executing GEMMs over $\mathbb{F}_p$ in fixed-point and proving a different computation than the GPU performed. Our precompiles take the opposite approach: the GPU executes actual BF16/FP32, and the proof system verifies *individual operations* by encoding IEEE 754 semantics as $\mathbb{F}_p$ constraints. Each floating-point value is decomposed into sign, exponent, and mantissa (integers in $\mathbb{F}_p$), and constraints enforce that the integer relationships among these match the IEEE 754 specification for the operation (multiplication, accumulation, rounding). The proof system runs over $\mathbb{F}_p$ internally but proves a statement about the floating-point computation that actually ran on the GPU. Eight precompiles cover a Transformer training step:

#	Precompile	Constraints/op	Used for
1	BF16/FP32 MAC chain	$\sim$90	GEMM dot products
2	BF16 lookup table	$\sim$15	GELU, SiLU (2^{16} entries)
3	Merkle path (zkVM-native hash)[10]	$\sim$1,500	Binding hints to commitments
4	FP32 exp	$\sim$250	Softmax, cross-entropy
5	FP32 sqrt/rsqrt	$\sim$200	RMSNorm, Adam
6	FP32 div	$\sim$200	Normalisation, Adam
7	FP32 log	$\sim$250	Cross-entropy loss
8	SHA-256 compression	$\sim$7,500	Network-anchor reconciliation

Table 3: Precompile catalogue. The MAC chain (1) dominates: $> 99\%$ of constraints at Llama 3.1 405B scale. Merkle path (3) is invoked per sampled value, $< 0.1\%$ of budget. SHA-256 (8) is used only when opening paths against the network anchor (Item **MOD. 4**). Full breakdown in Section B.

The dominant precompile is **BF16/FP32 MAC chain (precompile 1)**, accounting $> 99\%$ of constraints at 405B scale. It verifies a single multiply-accumulate: BF16 multiply producing an FP32 intermediate, accumulated into an FP32 running sum. The circuit decomposes each BF16 input into sign/exponent/mantissa, verifies the multiplication via integer arithmetic on the mantissa, and checks FP32 accumulation under IEEE 754 round-to-nearest-even. Each MAC costs $\sim$90 constraints; a dot product of dimension d requires d sequential MACs. **Precompile 1** is used for all GEMM verification in forward and backward passes. We use *interactive per-entry sampling* for GEMM verification, involving **precompile 1**[11]: the host hints the full result, the auditor selects k random output entries, and the proof checker recomputes each via the MAC precompile, binding inputs and output to their committed roots. With $k = 4{,}605$, deviations affecting $\geq 1\%$ of entries are detected with probability $1 - 10^{-20}$. Approximate sumcheck [7] suggests a path to recovering Freivalds-like efficiency for FP GEMMs (Section A.1). **Precompile 2** uses a precomputed 2^{16}-entry exhaustive table for BF16 activations (GELU, SiLU, derivatives) at $\sim$15 constraints per element. **Precompile 3** verifies Merkle-path membership (Poseidon, $\sim$1,500 constraints per depth-30 path); every hinted value passes a path check to bind it to the committed root. **Precompiles 4–7** encode FP32 nonlinearities (exp, log, sqrt/rsqrt, div) used sparsely in normalisation, softmax, optimiser, and loss, at 200–250 constraints per call ($< 1\%$ of total). **Precompile 8** (SHA-256, $\sim$7,500 constraints/block) is used exclusively for network-anchor reconciliation (Item **MOD. 4**). The rationale for the Poseidon + SHA-256 dual-hash design is in Section B and Section D. In the end, each layer is *decomposed and proved independently*; cross-layer soundness via root chaining ($R(\text{layer}_\ell\text{_out}) = R(\text{layer}_{\ell+1}\text{_in})$ verified inside each layer's proof). Per-layer sub-proofs run in parallel and are folded into a $\sim$200 KB aggregate via *recursive composition* (details and cost in Section B.7). *Fused GPU kernels* (Flash Attention, fused activations) do not materialise intermediate tensors in HBM, so those tensors cannot be Merkle-committed individually; the proof treats each fused block as a unit verified end-to-end at its input/output commitments (see details of costs for these fused blocks in Section B.3.1).

3.2 Protocol for AI training verification

The protocol produces three proof types, described below by three phases Items **P. 1** to **P. 3** and overviewed in Figure 1. All three share the commitments of Items **MOD. 2** to **MOD. 4** and the precompile catalogue (Section 3.1.1). We discuss the definitions and analyses our protocol in Appendix G: Overview of computational complexity-theoretic framework is given in Section G.2; Definitions of desired properties in Section G.3; Security analysis are given in Section G.4.

[10] Concretely Poseidon today; any zkVM-friendly hash with low per-call constraint cost (Poseidon2, Reinforced Concrete) is a drop-in substitute.

[11] Exact verificaiton for GEMM is costly, we make use of randomness and interaction to achieve efficiency, in the cost of non-perfect soundness, see Section B.2 for further discussion.

P. 1 (Genesis proof) The genesis proof binds the committed training procedure to actual GPU execution, run once before the hash chain begins. Without it, h_{commit} is a signed declaration of intent with no tie to physical work. The **genesis protocol** executes as follow. The auditor supplies *random sample indices* $\{i_1, \ldots, i_b\}$ into the committed dataset. The trainer (i) constructs $B = (\text{data}[i_1], \ldots, \text{data}[i_b])$ in the committed canonical order; (ii) executes one full training step from W_0 on B (forward, backward, optimiser update); (iii) produces a zkVM proof that each sample is a committed element of $R(\text{dataset})$ at its claimed index (one Merkle path per sample via precompile 3), the step executes faithfully per `arch_spec`, and the resulting W_1 is the first link of the public hash chain. *For soundness:* because indices are randomly chosen *after* h_{commit} is published, the trainer cannot precompute step at those indices (unless breaking the crypto commitment's binding); Merkle-path checks prevent batch substitution; full-step execution exercises the entire GPU pipeline so any divergence between claimed and executed code fails. The *cost* is one training step's worth of proof. Genesis establishes faithful execution at step 0; it says nothing about learning-theoretic properties or aggregate compute—those are the subject of the ex-ante attestation (Item **P. 3**).

P. 2 (In-training step proof) Fired many times across a run, live or post-hoc against archived snapshots, this phase gives the auditor assurance that any sampled step is consistent with the hash chain and network anchor. The **step protocol** executes as follow. The auditor selects a step $t \in \{1, \ldots, T\}$[12], a tensor identifier (layer ℓ output, gradient at node i, an all-reduce payload), and k random entry indices. The trainer hints: requested entries, Merkle paths against the committed $R(\cdot)$ for step t, and—for wire-bound tensors—the SHA-256 Merkle path against $R(\texttt{msg}_*)$. The proof checker dispatches: Merkle-path verification (precompile 3), operation verification (precompiles 1, 2, 4–7), and anchor consistency where relevant (precompile 8). At $k = 4{,}605$, deviations affecting $\geq 1\%$ of entries are detected with probability $1 - 10^{-20}$. *Soundness* across a run composes via two chaining relations, ensuring **cross-step binding**. Within a step, layer boundaries link by $R(\text{layer}_\ell\text{\_out}) = R(\text{layer}_{\ell+1}\text{\_in})$. Across steps, $R(W_{t+1})$ is a deterministic function of $R(W_t)$ and $R(\texttt{grad}_t)$ under the optimiser declared in `arch_spec`, verified inside the step proof. Inductively, this forms a directed chain from $R(W_0)$ to $R(W_T)$. Sampled-step and per-layer proofs aggregate via *recursive STARK composition* into a single $\sim$200 KB artefact.

P. 3 (Ex-ante attestation) The ex-ante attestation binds policy-relevant claims at initialisation and enforces them as running invariants throughout the online phase. Categories include: *compute* (total FLOPs, tokens, GPU-hours), *procedure* (regime, absence of a particular phase such as RL), and *data-content* (fraction flagged by a public classifier below a threshold). A trainer under an ex-ante attestation cannot exceed a committed bound unless breaking the *soundness* property of the proof instantiated by the zkVM—turning the training record into a governance-enforceable artefact rather than a mere audit trail. This phase involves a **commitment extension.** That is, the trainer publishes a structured `ex_ante_claims` containing committed bounds and references (e.g. `max_total_flops`, `max_steps`, `training_regime`, `data_filter_hash`). The genesis commitment extends to $h_{\text{commit}} = H(H(\texttt{arch\_spec}) \,\|\, R(W_0) \,\|\, R(\text{dataset}) \,\|\, H(\texttt{ex\_ante\_claims}))$. Bound values are public; only `arch_spec` stays private. Opening can be upfront or on-demand via selective disclosure against $H(\texttt{ex\_ante\_claims})$. Each step proof from **P. 2** performs **running enforcement**, by carrying an additional public running counter, e.g. $F_{\text{cumu}}(t)$ for cumulative FLOPs, computed in-circuit as $F_{\text{cumu}}(t) = F_{\text{cumu}}(t-1) + F_{\text{step}}(\texttt{arch\_spec}, t)$ with the constraint $F_{\text{cumu}}(t) \leq \texttt{max\_total\_flops}$[13]. More generally, the attestations involve a public *recursive* f over committed state and prove $f(\cdot) = y$ or $f(\cdot) \leq y$. The committed state varies with the sub-type: `arch_spec` and chain length for compute claims; `arch_spec` structure for procedure claims; sampled dataset entries plus a public classifier for data-content claims[14]. The recursive structures, together with soundness of the recursive STARK composition proof by zkVM, ensures a cumulative proof is accepted only if all intermediate steps satisfy the enforcement. The per-step overhead for this enforcement is one addition and one comparison—negligible vs. the MAC budget. We discuss *computation under-declaration* and *ex-post variants* in Section G.4.

[12] The challenge step t may depend adaptively on previously-observed public roots.

[13] The same pattern applies to step counts, per-regime tallies, and any monotone declared quantity.

[14] Data-content attestations combine Merkle-path openings into $R(\text{dataset})$ with in-circuit evaluation of the classifier and a concentration bound (Hoeffding) over the sampled fraction.

4 Conclusion

We argued that frontier dense pre-training admits a tractable verification architecture combining three complementary anchors (a committed training specification, inter-node network observations, and on-the-fly Merkle commitments of intermediate computation) verified through a generic zkVM with native BF16/FP32 precompiles. The architecture sidesteps the scale and arithmetic limits of the existing ZK-ML paradigm by verifying the actual hardware execution rather than a fixed-point approximation of it. At Llama 3.1 405B scale the training-side overhead is single-digit percent, aggregate proof size after recursive composition is approximately 200 KB, and end-to-end verification cost under a layered challenge mix is a fraction of a percent of training budget. For compute-threshold attestation (larger model, bigger data, more steps), each mode admits a direct structural detection keeping low overhead. Thirteen open problems delineate the remaining work to a fielded primitive; their statements and success criteria appear in Section A.

References

[1] Prashanth Amireddy, Amik Raj Behera, Srikanth Srinivasan, Madhu Sudan, and Sophus Valentin Willumsgaard. Ideals, macaulay bases, and pcps. In *STOC 2026*, 2026.

[2] Noga Amit, Shafi Goldwasser, Orr Paradise, and Guy N. Rothblum. Models that prove their own correctness. In *ICML 2024 Workshop on Theoretical Foundations of Foundation Models*, 2024.

[3] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998.

[4] Mauricio Baker, Gabriel Kulp, Oliver Marks, Miles Brundage, and Lennart Heim. Verifying international agreements on AI: Six layers of verification for rules on large-scale AI development and deployment. 2025.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive, Report 2018/046*, 2018.

[6] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Theory of Cryptography Conference (TCC 2016-B)*, 2016.

[7] Dor Bitan, Zachary DeStefano, Shafi Goldwasser, Yuval Ishai, Yael Tauman Kalai, and Justin Thaler. Sum-check protocol for approximate computations. *Cryptology ePrint Archive, Report 2025/2152*, 2025.

[8] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. ZKML: An optimizing system for ML inference in zero-knowledge proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, 2024.

[9] Ben Cottier, Robi Rahman, Loredana Fattorini, Nestor Maslej, Tamay Besiroglu, and David Owen. The rising costs of training frontier AI models. *arXiv preprint arXiv:2405.21015*, 2024. Published by Epoch AI.

[10] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

[11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[12] EU Council. Article 51: Classification of General-Purpose AI Models as General-Purpose AI Models with Systemic Risk. EU Artificial Intelligence Act Online Resource, `https://artificialintelligenceact.eu/article/51/`, 2024. Regulation (EU) 2024/1689, Official Journal version of 13 June 2024. Accessed: 2026-04-22.

[13] Sanjam Garg, Aarushi Guo, Omer Reingold, and Ron Roth. Experimenting with zero-knowledge proofs of training. *arXiv preprint arXiv:2310.02421*, 2023.

[14] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

[15] Shafi Goldwasser, Guy N. Rothblum, Jonathan Shafer, and Amir Yehudayoff. Interactive Proofs for Verifying Machine Learning. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference (ITCS 2021)*, volume 185 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41:1–41:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology – EUROCRYPT 2016*, 2016.

[17] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2021.

[18] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Scaling up trustless DNN inference with zero-knowledge proofs. In *arXiv preprint arXiv:2210.08674*, 2022.

[19] Guofu Liao, Taotao Wang, Shengli Zhang, Jiqun Zhang, Long Shi, and Dacheng Tao. VeriLoRA: Fine-tuning large language models with verifiable security via zero-knowledge proofs. In *Network and Distributed System Security Symposium (NDSS)*, 2026.

[20] Yehuda Lindell. An efficient transform from sigma protocols to nizk with a crs and non-programmable random oracle. In *Advances in Cryptology – EUROCRYPT 2015*, 2015.

[21] Dylan Patel and SemiAnalysis. Llama 3 training economics and cost breakdown, 2024. Industry analysis; see `https://www.semianalysis.com`.

[22] James Petrie. Guaranteeable memory: An HBM-based chiplet for verifiable AI workloads. In *Workshop on Technical AI Governance (TAIG) at ICML 2025*, Vancouver, Canada, 2025.

[23] James Petrie and Onni Aarne. Flexible hardware-enabled guarantees: Part II: Technical options for flexible hardware-enabled guarantees. Technical report, ARIA, 2025. arXiv:2506.03409.

[24] Xinwei Qiang, Hongmin Chen, Shixuan Sun, Jingwen Leng, Xin Liu, and Minyi Guo. DASH: Deterministic attention scheduling for high-throughput reproducible LLM training. In *International Conference on Learning Representations (ICLR)*, 2026.

[25] Aaron Scher, David Abecassis, Peter Barnett, and Brian Abeyta. An international agreement to prevent the premature creation of artificial superintelligence. 2025.

[26] Yonadav Shavit. What does it take to catch a Chinchilla? Verifying rules on large-scale neural network training via compute monitoring. 2023.

[27] Tobin South, Alexander Camuto, Shrey Jain, Shayla Nguyen, Robert Mahari, Christian Paquin, Jason Morton, and Alex Pentland. Verifiable evaluations of machine learning models using zkSNARKs. *arXiv preprint arXiv:2402.02675*, 2024.

[28] Haochen Sun, Jason Li, and Hongyang Zhang. zkLLM: Zero knowledge proofs for large language models. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[29] Haochen Sun and Hongyang Zhang. zkdl: Efficient zero-knowledge proofs of deep learning training. *arXiv preprint arXiv:2307.16273*, 2024.

[30] US Exec. Ord. Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence. Executive Order No. 14110, 88 Fed. Reg. 75191, `https://www.federalregister.gov/documents/2023/11/01/2023-24283/safe-secure-and-trustworthy-development-and-use-of-artificial-intelligence`, 2023. Signed October 30, 2023; published November 1, 2023. Revoked by Executive Order No. 14148 (January 20, 2025). Accessed: 2026-04-22.

A Open problems

This appendix catalogues the open research and engineering problems that remain after the architecture of Section 3. The list complements and expands the public roadmap maintained at `gpaipolicylab.org/verification`. Problems fall into four areas: proof foundations (questions that could shrink the proof budget), deterministic execution (questions that would reduce the training-side overhead), hardware trust anchors (questions that determine how independent the witness can be), and protocol extensions (questions about training regimes and deployment topologies beyond the base protocol). Each problem is phrased as a research question with a named target; they are independently attackable and invite contributions from cryptography, systems, hardware, and ML communities.

A.1 Outlook

Thirteen open problems separate this architecture from a fielded verification primitive. Their resolution is the remainder of the roadmap. The critical path passes through three items in particular: a deterministic-by-construction attention kernel (OP-4), a canonical wire-to-tensor mapping for NCCL (OP-7), and a zero-knowledge proof of backpropagation at non-trivial scale (OP-3). The first two are engineering problems with measurable success criteria. The third is the highest-risk milestone on the roadmap: a fundamental obstacle there would require rethinking the protocol, not just refining its implementation. Each item is independently scoped, so the architecture and its roadmap can advance in parallel across the cryptography, systems, hardware, and machine-learning communities. Whether the 36-month deployment envelope estimated here holds depends on coordinated progress along these fronts.

On the proof side, the MAC precompile accounts for roughly 99% of the proving budget, so constraint-level optimisations (OP-2) yield direct cost gains, while a more structural win—an algebraic reduction replacing per-element sampling (OP-1), for which approximate sumcheck [7] is a promising starting point—remains open for floating-point arithmetic; most critically, a zero-knowledge proof of backpropagation at $\geq 10^6$ parameters (OP-3) has never been demonstrated and a negative result would require redesigning the protocol. On the systems side, the determinism tax is dominated by attention backward (23–82% overhead scaling with sequence length), making a deterministic-by-construction kernel under 5% overhead (OP-4) the single most important engineering lever, alongside cross-architecture precompile portability (OP-5). The network anchor is disproportionately load-bearing for the trust model and depends on three unresolved problems: an open-hardware TAP at line rate (OP-6), a canonical wire-to-tensor mapping for NCCL (OP-7), and intra-node GPU-resident trusted execution (OP-8), with distinguishing silent data corruption from adversarial deviation (OP-9) a related concern. Finally, the base protocol must be extended to sparse MoE architectures (OP-10), reinforcement-learning post-training (OP-11), and multi-datacentre training with administratively partitioned links (OP-12); and a compiler pipeline from PyTorch/JAX to `arch_spec` (OP-13) is needed to bridge the tooling gap—a completeness concern rather than a soundness one, since converter bugs produce failing proofs for honest trainers, not passing proofs for dishonest ones.

A.2 Proof foundations

OP-1. Algebraic reductions for floating-point GEMM verification. Per-element sampling is the current verification strategy for BF16/FP32 GEMMs because Freivalds and sumcheck fail under non-associative floating-point rounding. *Question:* is per-element sampling provably optimal for floating-point matmul verification, or does an algebraic shortcut exist that exploits structure (e.g. the accumulation tree topology declared in the precompile)? Recent work on sumcheck with bounded approximation error [7] suggests an approximate Freivalds test could absorb rounding discrepancy with graceful soundness degradation; the obstacles are (i) polynomial commitments that preserve soundness under approximation and (ii) a tight error bound δ distinguishing rounding noise from adversarial deviation. A tight lower bound would close the question; an algebraic reduction would cut the dominant proof cost by one to two orders of magnitude.

OP-2. Constraint minimisation for the MAC precompile. The BF16/FP32 MAC precompile costs $\sim$90 constraints per operation over the Baby Bear field and accounts for roughly 99% of the proof budget at frontier scale. *Question:* can custom gates, lookup arguments, or alternative field

representations bring the MAC cost down to 30–50 constraints per operation? Even a factor of two would halve end-to-end proving cost at modern frontier scale.

OP-3. Zero-knowledge proof of backpropagation. No prior work has demonstrated a zero-knowledge proof of backpropagation for a full LLM in native floating-point arithmetic. Existing demonstrations cover LoRA fine-tuning on quantised models [13] or forward inference only. The backward pass reuses the same primitives (MAC, Merkle paths, lookup) but the complete forward+backward circuit has never been exercised at scale in a zkVM. *Question:* produce either (a) a functional proof for a transformer of at least 10^6 parameters with bit-exact gradient match to GPU execution, or (b) an identified fundamental obstacle. This is the highest-risk milestone on the roadmap: a negative result would require redesigning the verification protocol.

A.3 Deterministic execution on hardware

OP-4. Deterministic attention backward kernels with $<5\%$ overhead. FlashAttention-2/3 backward incurs 23–82% overhead in deterministic mode, scaling with sequence length (Section C). DASH [24] achieves 1.28× over baseline deterministic mode. *Question:* can attention backward be made deterministic *by construction* (fixed warp-level reduction trees, algebraic reformulation) with under 5% overhead for sequence lengths $\geq$ 8K? This is the dominant source of the determinism tax.

OP-5. Multi-architecture precompile specification. Each GPU (H100, MI300X, Gaudi3) has distinct accumulation tree topology, rounding mode, and subnormal handling. Each currently requires a hand-engineered precompile. *Question:* can a parameterised precompile family be defined (instantiated automatically from a compact hardware descriptor) and validated against silicon? Sub-problems include a formal specification language for vendor numerics, automated constraint generation, and an empirical validation methodology (how many test vectors suffice to confirm a specification matches hardware?).

A.4 Hardware trust anchors

OP-6. Open-hardware network TAP at line rate. Tier 1 of Item **MOD. 4** requires a passive, auditable device that hashes inter-node traffic at 400+ Gbps. Commercial TAPs exist but an open-hardware, formally verified device would establish a new trust primitive for AI governance. *Question:* can streaming SHA-256 hashing at 400+ Gbps be implemented in FPGA fabric with formally verified RTL and tamper-evident physical packaging? Integration with commodity switching hardware is part of the target.

OP-7. Wire-to-tensor mapping for the network anchor. The network anchor (Item **MOD. 4**) is useful only if wire observations can be reconciled against the trainer's tensor commitments. This requires a canonical mapping between logical tensors and NCCL's on-wire byte sequence, which is not a stable public interface today: NCCL's byte layout depends on runtime configuration (`NCCL_BUFFSIZE`, `NCCL_NTHREADS`, algorithm selection), ring-allreduce intermediate states on the wire are partial reductions rather than tensor slices, and the anchor must parse InfiniBand or RoCE framing to strip volatile fields (PSN, ICRC) before hashing. *Core question:* which resolution path delivers the best trade between verification assurance and ecosystem adoption? Three candidates are available:

1. **Patched NCCL with a documented canonical format.** A forked NCCL exposes a version-locked on-wire layout and enumerates which logical tensor bytes map to which wire offsets. The cleanest path, compatible with open-source verification, but requires ongoing maintenance as NCCL evolves.
2. **Canonicalisation shim at the NCCL-to-transport boundary.** A signed, auditable library interposed between NCCL and the NIC emits a canonicalised copy of each outgoing payload for the anchor to hash. Trades a small amount of additional memory traffic for independence from NCCL's internals.
3. **Vendor-supported "verifiable NCCL" mode.** A first-class configuration flag with documented wire semantics. The only path that does not require the verification ecosystem to maintain its own fork or shim; depends on vendor cooperation.

All three yield equivalent verification guarantees and differ in deployment effort, maintenance burden, and institutional leverage required.

Three subsidiary questions attach to this problem:

- **Queue-pair mapping auditability.** The NCCL-to-QP mapping is established at init by trainer-host software. How does the auditor audit this mapping without trusting the trainer's host? One candidate is to require the anchor (Tier 1) or the SmartNIC firmware (Tier 2) to publish its own independent view, which must match the trainer's or the run is rejected.
- **Tier 2 attestation protocol.** BlueField DOCA exposes a signing and attestation stack rooted in secure boot and a device identity key. The end-to-end protocol for continuous commitment streaming, key rotation, and revocation is unspecified and would benefit from a standardised reference design.
- **Chunk-boundary alignment.** Tensor elements (2 bytes in BF16) are small relative to Merkle leaves (tens of kilobytes). If the canonical wire format allows elements to straddle leaf boundaries, verifying a single element requires two path openings instead of one. Alignment guarantees must be specified in whichever resolution path is chosen.

OP-8. Intra-node physical witness. Neither tier of the network anchor observes intra-node traffic over NVLink, so hybrid FSDP+TP training has no physical witness for the tensor-parallel layer. *Question:* can GPU-resident trusted execution (NVIDIA Confidential Compute or a successor mechanism) provide continuous, attested hashing of intra-node collective payloads at the bandwidth required? Current Confidential Compute does not expose the interface nor the throughput; identifying the minimal hardware-attestation primitives that would close this gap is an open design question.

OP-9. Distinguishing SDC from adversarial deviation. Silent data corruption on frontier hardware produces bit-flips that are indistinguishable from intentional modification at the level of an individual event. Empirical SDC rates at hyperscale are on the order of one event per 10^4–10^5 GPU-hours [10, 17]; for a 405B-class training run of $\sim 10^7$ GPU-hours this yields on the order of 10^2–10^3 SDC events (an optimistic anchor, $\sim$120 events at 1 per 10^5 GPU-hours). A sophisticated adversary could mask structured deviations as statistically natural SDC. *Question:* can statistical tests be designed to distinguish SDC patterns (random, memoryless, correlated with known failure modes such as DRAM row faults or thermal events) from structured adversarial injection? Empirical characterisation of SDC distributions on current silicon is a prerequisite.

A.5 Protocol extensions

OP-10. Mixture-of-experts verification. The base protocol assumes the computation graph is determined by the `arch_spec` ahead of time. MoE relaxes this: routing decisions are data-dependent, and which expert processes which token is decided at runtime by the router network. Extending the protocol to MoE introduces several distinct sub-problems:

- **Routing commitment.** A per-step Merkle root $R(\texttt{routing}_t)$ must join the hash chain, committing to which top-K experts were selected for each token. Without this, routing decisions are not bound to the proof.
- **Top-K selection verification.** Top-K over BF16 scores is a non-smooth, data-dependent operation. Either a dedicated precompile or a decomposition into pairwise comparisons is required; the latter is cheap per element but scales with $N \log K$ per token.
- **Deterministic tie-breaking.** Ties between expert scores (rare with BF16 but possible) must be resolved by a documented rule (e.g. expert ID) declared in `arch_spec`.
- **All-to-all canonicalisation.** Expert parallelism uses all-to-all to dispatch tokens to their selected experts. The wire byte layout depends on the committed routing, making canonicalisation a harder instance of OP-7: the mapping is data-dependent rather than configuration-dependent.
- **Load-balancing auxiliary losses.** Importance and load losses must be declared in `arch_spec` so they are bound by the proof; softmax, log, and division precompiles already cover the operations.

Framing for cost arguments. Per-step proof cost for MoE scales with per-token active compute, not with total parameters. For an MoE with K active experts per token, the per-step cost is approximately K times the cost of a dense model whose dimensions match an individual expert, plus the routing overhead listed above, because each sampled output entry requires verifying all K active experts. A 1T MoE with 32B active per token (e.g. Kimi K2, $K{=}8$) therefore costs substantially less per step than a hypothetical 1T dense model, but more than a 32B dense model.

OP-11. Reinforcement-learning post-training. RL post-training (PPO, GRPO, DPO, RLHF, RLAIF, rule-based RL for reasoning) relaxes a different assumption of the base protocol: rollouts are stochastic and model-dependent, so the inputs on which the model is trained are themselves outputs of the model. Extensions required:

- **Sampling seed commitment.** Per-rollout RNG seeds (and sampling hyperparameters) enter `arch_spec` so that rollouts are reproducible.
- **Rollout verification.** Given the committed policy and the committed seed, sampled completions are a deterministic function. The proof verifies that the stored rollout data matches this deterministic output.
- **Reference-model binding.** PPO/RLHF uses a frozen reference policy for KL regularisation. Its commitment (typically the pre-RL checkpoint, verifiable through the pre-training chain) is referenced by `arch_spec`.
- **Reward commitment.** If the reward is rule-based (deterministic function such as test-case pass/fail for code, or formal verification for math), it is specified in `arch_spec` directly. If the reward is a trained model, its training run has its own h_{commit}, referenced by the policy's `arch_spec`.
- **PPO clipping.** The clipped objective introduces data-dependent gradient masking (clipping depends on per-token probability ratios). Verifying the clipping condition per sampled token is an additional per-sample check, structurally similar to verifying routing.

Recommended first target. Rule-based RL for reasoning (e.g. math and code in the style of DeepSeek-R1) eliminates the reward-model chain and collapses to deterministic reward verification. This is the cleanest first RL target and should be where the extension is demonstrated before RLHF-style settings.

OP-12. Multi-site training verification. Frontier training increasingly spans multiple datacentres connected over WAN links. The anchor model assumes an auditor-controlled observation point on every inter-node link; realistic deployments split link control across administrative boundaries. *Question:* how does the protocol adapt when some network segments are controlled by different parties? Sub-problems include consistency of Merkle commitments across multiple anchors, handling clock skew and latency heterogeneity, Byzantine-robust aggregation of per-site hashes, and incentive alignment when organisations share a training run.

A.6 Protocol tooling

OP-13. Converter from training code to `arch_spec`. The protocol commits to `arch_spec` as the canonical description of the training computation, but real trainers write PyTorch (or JAX) code, not `arch_spec`. Bridging the two requires a converter pipeline: forward-graph capture (`torch.export`, `torch.fx`), backward-graph capture (`functorch`), mapping from ATen ops to declared `OpSpec` variants, distributed-annotation capture, optimiser-step capture, and serialisation. *Question:* can a converter be built that (a) covers the op set used by frontier training codebases, (b) produces a deterministic `arch_spec` from semantically equivalent PyTorch code, and (c) fails loudly on unsupported ops rather than silently dropping structure? The converter is a completeness concern (does every valid PyTorch training script produce a correct `arch_spec`?), not a soundness concern: the auditor only checks the proof against $\mathrm{Hash}(\texttt{arch\_spec})$, so a converter bug produces a failing proof for an honest trainer, never a passing proof for a dishonest one. ZKML [8] is a precedent for inference-only ONNX conversion; the training case adds backward capture and distributed annotations.

B Proof cost and overhead estimation

This appendix supports the quantitative claims in Section 3 with detailed derivations. It covers (i) the zkVM cost model and per-operation constraint counts, (ii) why exact GEMM verification is infeasible over floats and why Freivalds' algorithm does not directly apply, (iii) the per-layer and per-step proof budget at Llama 3.1 405B scale under the commitment and fusion assumptions used in Section 3.1, (iv) the statistical properties of interactive sampling, (v) the MoE cost story, (vi) the three proof flows (genesis, in-training, ex-ante) with concrete per-flow costs, and (vii) a training-side overhead summary. Caveats and uncertainties are at the end.

B.1 Cost model

B.1.1 Mechanics: hint-and-verify

The zkVM operates as a verification machine rather than an execution machine. For each operation in the training step, the host (trainer's GPU) computes the result and provides it as an unchecked hint. The guest (proof checker running inside the zkVM) verifies the hint against committed Merkle roots and against declared operations via native constraint circuits called precompiles. Host execution is free from the proof's perspective; proving cost is determined entirely by the number of constraints the precompiles generate.

B.1.2 Per-operation constraint counts

Table 4 expands the precompile catalogue of Table 3 with the cost model at the primitive level.

#	Primitive cost	Per-invocation	Used for
1	BF16/FP32 MAC	∼90 constraints	GEMM dot products (one per sampled entry times inner dim)
2	BF16 table lookup	∼15 constraints	Activation functions (GELU, SiLU, their derivatives)
3	zkVM-native hash compression	∼75 constraints	Merkle-path verification (∼1,500 / path at depth 30)
4	FP32 exp	∼250 constraints	Softmax, cross-entropy loss
5	FP32 sqrt/rsqrt	∼200 constraints	RMSNorm, Adam
6	FP32 div	∼200 constraints	Normalisation, Adam
7	FP32 log	∼250 constraints	Cross-entropy loss
8	SHA-256 compression	∼7,500 constraints	Anchor-path verification (∼150,000 / path at depth 20)

Table 4: Per-invocation constraint counts for the eight precompiles. The dominant primitive is the BF16/FP32 MAC; Merkle-path verifications are an additive per-sample cost; SHA-256 is invoked only for wire-bound sample reconciliation. Per-path totals assume a tree depth matched to the payload size (zkVM-native Merkle for tensors up to $\sim 10^9$ elements at depth 30; SHA-256 anchor paths at depth 20 for 1 GB collective payloads with 1 MB leaves).

The per-MAC count of ∼90 is an estimate based on the IEEE 754 integer decomposition: BF16 multiply (∼30–50 constraints for mantissa product and exponent addition), FP32 accumulation with RNE rounding (∼40–60 constraints for exponent alignment, mantissa addition, guard/round/sticky bits). The 50–150 range this spans affects absolute proof-time numbers by up to ∼2× but not comparative analysis between approaches; a direct measurement of the actual count is part of the planned empirical validation work.

B.1.3 Proving throughput

The cluster-side proving throughput assumed throughout is $\sim 10^6$ constraints per second per GPU (RISC Zero-class benchmark on A100/H100 for constraint-heavy guests), i.e. $\sim 10^9$ constraints per second aggregate on a 1,024-GPU cluster. Segment size is 2^{20} cycles, segment-level proving is embarrassingly parallel, and recursive composition folds per-segment proofs into a constant-size aggregate (∼200 KB) independent of the underlying circuit size.

B.2 Exact GEMM verification is O(n^3); Freivalds does not apply

Exact verification of an $m \times d \times n$ GEMM at ∼90 constraints/MAC costs $O(mnd)$, exceeding current proof systems by several orders of magnitude at 405B scale. Freivalds' algorithm reduces this to $O(mn + md + nd)$ but does not apply to BF16/FP32 since floating-point addition is *not* associative (Section B.2.2).

B.2.1 Why the naïve verification is cubic

A GEMM of dimensions $m \times d \times n$ computes $C = A \cdot B$ where each entry C_{ij} is a dot product of length d of BF16 inputs accumulated in FP32 with IEEE 754 RNE rounding. Verifying every entry exactly requires mn dot products, each consisting of d MACs, yielding mnd MAC verifications. At $\sim$90 constraints per MAC this is $\sim 90mnd$ constraints. For a single Q-projection GEMM of Llama 3.1 405B (taking $m = 2{,}048$, $d = n = 16{,}384$), the unsampled cost is $\sim 5 \times 10^{13}$ constraints, which exceeds the capacity of current proof systems.

B.2.2 Why Freivalds' algorithm does not directly reduce this for floats

Freivalds' algorithm verifies $C = A \cdot B$ by drawing a random vector $r \in \mathbb{F}_p^n$ and checking $A(Br) = Cr$. Over an exact field this reduces verification from $O(n^3)$ to $O(n^2)$ with error probability $\leq 1/|\mathbb{F}_p|$. For native BF16/FP32 arithmetic the algorithm fails, because floating-point addition is not associative. The two computation paths $A(Br)$ and Cr involve different accumulation orders and therefore different rounding sequences, producing different bit-exact outputs even when C is the correct GPU output. The check rejects correct computations.

Converting floats to exact integers (for example, by scaling BF16 mantissa-exponent pairs to a common exponent and padding with zeros to a fixed bit-width) and applying Freivalds over a large field is theoretically available, but introduces three obstacles: (i) the scaled integers exceed the trainer's native Baby Bear field ($\sim 2^{31}$), requiring multi-limb arithmetic with $\sim 121\times$ overhead per operation; (ii) the exact integer product does not equal the GPU's rounded output, so an $O(n^2)$ rounding-verification pass must be added on top; (iii) the interaction between exact-integer Freivalds and the FP32 accumulation rounding chain that determines the GPU's actual output is an open research question. Recent theoretical work on sumcheck protocols that tolerate bounded approximation error [7] points to a path for recovering Freivalds-like asymptotics over floats, discussed as an open problem (OP-1).

In the remainder of this appendix the baseline assumption is interactive per-element sampling; any future reduction is an optimisation on top.

B.3 Per-step cost at Llama 3.1 405B

B.3.1 Commitments and fusion assumptions

Per Item **MOD. 3**, the trainer commits six Merkle roots per layer during training: layer input, Q, K, V, attention output, and layer output. What is *not* committed during training has cost consequences, so we state this explicitly.

- **Flash Attention internals are truly fused.** The attention score matrix ($B \times \text{heads} \times s \times s$) and the softmax output never materialise in GPU memory; they are streamed through shared memory inside the Flash Attention kernel. These are uncommittable by kernel construction, not by choice. When the auditor samples attention-output entries, the host recomputes each sampled entry by running the full per-query attention over the sequence dimension. Per-sample cost is $\Theta(s \cdot d_{\text{head}})$ MACs, which is materially larger than a single GEMM dot product. To keep the attention contribution bounded, we use a reduced sample count $k_{\text{attn}} = 500$ for attention outputs, relying on the committed boundary roots (Q, K, V, attention output) for the primary consistency guarantee.

- **FFN-block internals are not truly fused.** In Llama-style SwiGLU blocks the gate and up projections materialise in HBM as separate tensors before the elementwise activation and multiply. We exploit this by committing $\mathrm{gate\_out}$ and $\mathrm{up\_out}$ Merkle roots *at challenge time* (rather than during training) when the trainer re-runs the step. The training-time hashing overhead therefore remains at six roots per layer, while the per-challenge proof samples each FFN GEMM independently at the cheaper per-GEMM rate. This is a deliberate design trade: training-time hashing stays cheap, challenge-time response grows by $\sim$1 second per layer of hashing, and the per-sample cost of FFN verification drops by a factor of $\sim d_{\text{ff}}$.

B.3.2 Per-layer constraint budget

We compute the per-layer forward + backward cost at Llama 3.1 405B scale ($d_{\text{model}} = 16{,}384$, $d_{\text{ff}} = 53{,}248$, $n_{\text{heads}} = 128$, $n_{\text{kv\_heads}} = 8$ with GQA, $d_{\text{head}} = 128$, 126 layers, sequence length $s = 2{,}048$ per microbatch, $k = 4{,}605$ samples per committed GEMM, $k_{\text{attn}} = 500$ for attention outputs).

Each forward GEMM of shape $(m \times d) \times (d \times n)$ induces two backward GEMMs (input gradient and weight gradient) whose inner dimensions are n and m respectively. Per sampled output entry, the proof cost is inner-dimension $\times 90$ constraints.

Per-layer component	Inner-dim sum (fwd + 2 bwd)	Sampled cost (k = 4,605)	Layer contribution
Q projection	34,816	$90 \cdot k \cdot 34{,}816$	1.44×10^{10}
K projection (GQA, smaller n)	19,456	$90 \cdot k \cdot 19{,}456$	8.06×10^{9}
V projection (GQA)	19,456	$90 \cdot k \cdot 19{,}456$	8.06×10^{9}
O projection	34,816	$90 \cdot k \cdot 34{,}816$	1.44×10^{10}
FFN gate projection	71,680	$90 \cdot k \cdot 71{,}680$	2.97×10^{10}
FFN up projection	71,680	$90 \cdot k \cdot 71{,}680$	2.97×10^{10}
FFN down projection	71,680	$90 \cdot k \cdot 71{,}680$	2.97×10^{10}
Flash Attention (fused, $k_{\text{attn}} = 500$)	$\Theta(s \cdot d_{\text{head}})$ per sample	end-to-end recompute	$\sim 5 \times 10^{10}$
Elementwise SiLU + multiply (per FFN sample)	d_{ff} ops $\times$ (15 + 90)	lookup + BF16 mult	$\sim 2.6 \times 10^{10}$
Merkle paths (6 trainer-side per sample)	depth 30, zkVM-native hash	1,500 constraints each	$\sim 4 \times 10^{7}$
Per-layer total (fwd + bwd)			$\sim 2.1 \times 10^{11}$

Table 5: Per-layer forward + backward constraint budget at Llama 3.1 405B scale under the commitment and fusion assumptions of Section B.3.1. Q/K/V/O and FFN GEMMs are sampled per-GEMM (trainer-side Merkle roots at layer boundaries and lazy intermediates bind the hints). Flash Attention uses reduced-sample recomputation; elementwise SiLU and multiply contribute per-FFN-sample. SHA-256 anchor-path verification (precompile 8) is zero here because no wire-bound tensors are verified per layer on an intra-node path; it appears only on sampled wire-bound tensors (Section B.4.2).

B.3.3 Per-step total

Summing 126 layers contributes $126 \cdot 2.1 \times 10^{11} \approx 2.6 \times 10^{13}$ constraints. The LM-head contribution is bounded above by a single GEMM with inner-dim sum $d_{\text{model}} + \text{vocab} + m \approx 146{,}688$, giving $\sim 6 \times 10^{10}$ constraints (less than 0.3% of the total).

Per-step proof budget: $\sim 2.6 \times 10^{13}$ constraints. The MAC precompile accounts for over 99% of this budget; Merkle-path verification (precompile 3) contributes well under 0.1%. SHA-256 anchor-path verification (precompile 8) is invoked only for wire-bound samples, adding a bounded amount per challenged wire-bound tensor (see Section B.4.2).

B.3.4 Proving time

At a cluster-wide throughput of $\sim 10^9$ constraints per second (1,024 GPUs at $\sim 10^6$ constraints per second each), a full-step proof of Llama 3.1 405B completes in $\sim$7 hours. Linear scaling of compute yields $\sim$100 minutes on 4,096 GPUs, or $\sim$25 minutes on 16,384 GPUs. Per-layer proofs are independent and compose via recursive STARK folding, so the critical path is the most expensive individual layer proof rather than the serial sum, which becomes relevant when proving is parallelised across dedicated proof clusters.

B.3.5 Comparison with the O(n^3) baseline

Summed over all per-layer GEMMs, the unsampled baseline at Llama 3.1 405B yields $\sim 5.5 \times 10^{18}$ constraints per step ($\sim$one-hundred-thousand-fold the sampled cost). The $\sim$82,000$\times$ reduction from interactive sampling at $k = 4{,}605$ is what makes proof generation feasible at all at this scale.

B.4 Interactive sampling: statistical properties

B.4.1 Sample count for target security

For auditor-chosen samples drawn after Merkle roots have been committed (no grinding), the probability of missing a deviation affecting fraction f of entries is $P_{\text{miss}} = (1 - f)^k$. Solving for

target miss probability τ:

$$k \geq \frac{\ln(1/\tau)}{-\ln(1-f)} \approx \frac{\ln(1/\tau)}{f} \quad \text{(for small } f\text{)}.$$

Target miss probability	$f = 10\%$	$f = 1\%$	$f = 0.1\%$
10^{-20}	$k = 437$	$k = 4{,}605$	$k = 46{,}052$
2^{-128} (match ZK soundness)	$k = 842$	$k = 8{,}840$	$k = 88{,}530$

Table 6: Sample counts for interactive sampling. The recommended default $k = 4{,}605$ achieves 10^{-20} miss probability per committed tensor at a 1% deviation fraction.

The default $f = 1\%$ is a tuning choice rather than a fundamental threshold. Sampling detects deviations affecting a fraction f of entries with probability $1-(1-f)^k$, so any target miss probability is reachable by scaling k: smaller f simply requires more samples ($k \propto -\ln(\text{miss})/f$). The default $k = 4{,}605$ corresponds to 10^{-20} per-tensor miss probability at $f = 1\%$; the same guarantee at $f = 0.1\%$ requires $k = 46{,}052$. For wire-bound tensors, the network anchor (Item **MOD. 4**) provides an independent commitment that the proof binds against at every sampled entry; a mismatch between the trainer's tensor commitment and the anchor's wire commitment surfaces as a failed anchor-consistency check inside the zkVM proof. Formal sensitivity analysis bounding the smallest f a rational adversary can exploit is an open problem (OP-1).

B.4.2 Network-anchor bindings

Wire-bound tensors are committed twice: the trainer holds a zkVM-native Merkle root of the tensor value, and the network anchor holds an SHA-256 Merkle root of the wire payload that transported that tensor. When the auditor samples a wire-bound entry, the proof opens both paths and verifies that the element value is consistent with both commitments. Per sampled wire-bound element, the anchor consistency check costs $\text{depth} \cdot \text{SHA-256}_{\text{compress}} + \lceil c/64 \rceil \cdot \text{SHA-256}_{\text{compress}}$ for a chunk of c bytes (∼2–8 million constraints at $c = 16$–64 KB leaves per Section D). Anchor checks are invoked only for sampled wire-bound tensors, so their contribution to per-step budget is bounded: typically under 5% of the budget even when every inter-node tensor is sampled.

B.4.3 Defense-in-depth

The general-purpose security argument against an adaptive adversary composes three layers:

1. **Network anchor.** Catches any deviation that changes wire-bound content; deterministic (not statistical) detection on any sampled wire-bound tensor.
2. **On-the-fly Merkle commitments.** Each layer's output is committed during training. Any error in any intermediate that propagates to the layer output changes the committed root; the proof catches this at the layer boundary.
3. **Interactive sampling.** Adds 10^{-20} statistical assurance on intermediate GEMMs for deviations affecting $\geq 1\%$ of entries.

Each layer is necessary: (1) alone cannot see intra-node computation; (2) alone does not rule out an adversary who produces internally consistent fake commitments; (3) alone has a detection floor at the deviation fraction. Together they provide complementary coverage.

Compute-threshold attestation: case-by-case analysis. The three-layer argument above is the general case against an adversary cheating on arbitrary aspects of the training procedure. For the specific target of compute-threshold attestation (ex-ante binding of total FLOPs, Item **P. 3**), the adversary's options are narrower. The rational adversary under this target wants to *under*-declare compute: to run more than was claimed in order to stay below a regulatory threshold. Three cheating modes are available, each with a cheap and direct detection.

- **Model larger than declared (wider).** The adversary inflates d_{model}, d_{ff}, n_{heads}, or another dimension while publishing `arch_spec` for smaller values. The commitment $R(W_0)$ is a Merkle tree whose leaves and depth are determined by the actual weight shapes. An opening

at any challenged weight position in the genesis proof must produce a path matching the `arch_spec`-declared shape. A wider actual weight matrix has leaves at different offsets; the path opening fails. Detection requires a single genesis weight-opening.

- **Model larger than declared (deeper).** The adversary runs $L_{\text{actual}} > L_{\text{declared}}$ layers. Three consequences: (i) the committed per-layer chain terminates at L_{declared}, so the input to the committed LM head is the declared-depth activation, not the actually-deeper activation from which the loss was computed; (ii) in the backward pass, the gradient committed at declared layer L_{declared} does not equal $\partial L/\partial \text{layer}_{L_{\text{declared}}}$ under `arch_spec` because the real backprop chain is longer; (iii) the network anchor observes inter-node communication volume proportional to L_{actual}. Any of these three discrepancies is caught by a single in-training or genesis challenge.

- **Data larger than declared.** The adversary trains on α-fold more data than committed in $R(\texttt{dataset})$. The `arch_spec`-declared shuffling seed determines which dataset indices appear in batch t. A fraction $1 - 1/\alpha$ of steps use indices absent from $R(\texttt{dataset})$. Opening the layer-0 input commitment $R(\texttt{mlp}_0\texttt{\_in})$ at committed positions fails for those steps. Detection probability with Q uniform random input-openings:

$$P(\text{detect}) = 1 - \alpha^{-Q}.$$

  Hiding $\alpha = 10$ data needs $Q = 2$ for 99% detection; $\alpha = 2$ needs $Q = 7$; $\alpha = 1.1$ needs $Q = 49$. Large-scale data smuggling is trivially caught; the detection budget scales with the inverse of the attempted inflation factor.

- **More training steps than declared.** The network anchor (Item **MOD. 4**) observes inter-node collectives continuously for the duration of the run. For training that requires inter-node communication at every step (FSDP, TP across nodes, PP), the anchor's observed collective count matches the actual step count. The trainer's published chain has one entry per declared step. Discrepancy is a single integer comparison and requires no cryptographic proof.

The four cases together imply a compute-threshold attestation needs far fewer challenges than general-purpose `arch_spec` verification. A conservative budget (genesis + a handful of input-data challenges at $Q \sim 10$ to rule out $\geq 2\times$ data smuggling + anchor-count check) costs on the order of \$15–25K, rather than the \$7–14M required for $Q \in \{500, 1{,}000\}$ random challenges in the generic worst case.

The overall cost-analysis tables retain the larger sampling budget as the default for two reasons. Formal sensitivity analysis (OP-1) could yet reveal corner cases that rescale the argument for the three detections above. Attestations beyond compute (for instance data-content filters targeting *semantic* smuggling within the committed dataset, or procedural attestations on fine-grained regime compliance) lack the structural detection surface of compute and operate closer to the 500-challenge regime against adaptive adversaries.

B.5 MoE verification cost

Per OP-10, per-step proof cost for MoE scales with per-token active compute, not with total parameters. For an MoE with K active experts per token of per-expert dimensions ($d_{\text{model}} \times d_{\text{expert}}$), the per-sample FFN cost is approximately K times the cost of a dense model with the same per-expert dimensions, plus a small routing overhead. Kimi K2 ($K = 8$, $d_{\text{model}} = 7{,}168$, $d_{\text{expert}} = 2{,}048$) costs substantially less per step than a hypothetical 1T dense model, but more than a 32B dense model because each sampled output requires verifying all K active experts.

Router verification adds a small per-token GEMM of inner dimension d_{model} plus a top-K selection step (either a dedicated precompile or a decomposition into pairwise comparisons, scaling as $N \log K$ for N total experts). Load-balancing auxiliary losses (importance, load) introduce per-token softmax and division, covered by existing precompiles 4 and 6 at negligible cost relative to the expert GEMMs. All-to-all canonicalisation for expert parallelism is a separate open problem (OP-7 and OP-10).

Concrete cost figures for MoE verification depend on sampling-scheme design decisions that are themselves open (OP-10); detailed per-architecture numbers are left to future work once OP-10 is resolved.

B.6 Proof-flow costs

The protocol produces three proof types (Section 3.2). Their costs differ.

B.6.1 Genesis proof

Executed once at initialisation on an auditor-chosen batch of b samples drawn from the committed dataset. The proof covers one full training step (forward + backward + optimiser update). Its cost is the per-step budget of Section B.3.2 for the chosen b, plus b dataset-membership Merkle-path openings (one per sampled index). At $b \leq$ typical microbatch size this is dominated by the per-step cost and completes in the $\sim$7-hour envelope on 1,024 GPUs. A reduced-b genesis (e.g. $b = 128$) lowers proof cost proportionally while still exercising all code paths, which may be appropriate when genesis latency is operationally important.

B.6.2 In-training step proof

Executed many times across a training run, each invocation on a sampled step t. Four challenge granularities are meaningful in practice; the auditor can mix them according to the threat model.

- **Single-commit challenge** (verify one committed root against an adjacent committed root via a single declared operation, for instance a GEMM verified by the MAC precompile, an activation verified by the BF16 lookup, or a LayerNorm verified by the FP32 nonlinear precompiles): up to $\sim 2 \times 10^{10}$ constraints, $\sim$20 s on 1,024 GPUs, $\sim$\$12 at \$2/GPU-hour. GEMM-backed challenges set this headline cost; challenges that exercise cheaper precompiles (table lookups, FP32 nonlinear ops) are essentially free by comparison.
- **Layer-partial challenge** (verify one layer's forward pass against its input and output commitments, or one layer's backward pass against its gradient-boundary commitments; the two are structurally independent sub-proofs): $\sim 1 \times 10^{11}$ constraints each, $\sim$1.8 min on 1,024 GPUs, $\sim$\$60.
- **Depth-N partial chain** (N consecutive forward layers, or N consecutive backward layers, chained via their committed boundary roots): N times the layer-partial cost. Useful when the auditor wants one proof that cross-checks multiple intermediate roots and their boundary consistency, for example an entire pipeline-parallel stage.
- **Full-step challenge** (all 126 layers of a step, forward plus backward; used for genesis and periodic comprehensive audits): $\sim 2.6 \times 10^{13}$ constraints, $\sim$7 h on 1,024 GPUs, $\sim$\$14K. Proving is embarrassingly parallel across segments, so on 16,384 GPUs the same work completes in $\sim$25 min for the same total GPU-hours.

A realistic challenge mix over a 100k-step run at Llama 3.1 405B scale might be 10,000 single-commit challenges for broad coverage ($\sim$\$120K), 500 layer-partial challenges for layer-boundary integrity ($\sim$\$30K), 100 depth-6 partial chains for cross-checking PP-stage boundaries ($\sim$\$36K), and 10 full-step challenges for comprehensive audits ($\sim$\$140K). Total: $\sim$\$326K, or $\sim$0.33% of a \$100M training budget. This "many cheap, few expensive" mix is driven by threat-model coverage rather than by any single security bound; auditors targeting only compute-threshold attestation can run a much smaller budget (see Section B.4.3).

Per-challenge response time exclusive of proof generation is dominated by the host-side re-run: loading the stored weights from the rolling window (seconds on NVMe) and re-executing the forward + backward pass for the challenged step on a partial replica (minutes to hours depending on challenge depth).

B.6.3 Ex-ante attestation running invariant

Ex-ante attestations enforce invariants (e.g. $F_{\text{cum}}(t) \leq$ `max_total_flops`) at every step proof. The invariant check is a single addition and a single comparison per step, adding a constant number of constraints (under 100) to each step proof. For any realistic invariant list the contribution is negligible relative to the per-step MAC budget.

B.7 Deeper proofs and proof composition

A depth-1 step proof trusts its input Merkle roots: if the trainer committed fake input roots, the proof still passes (the auditor catches the inconsistency downstream when the fake root does not match the preceding step's output). A depth-N proof starts from N layers earlier and verifies each intermediate commitment along the chain in-proof, at roughly N times the single-layer cost.

Challenge	Proving time (1,024 GPUs)	What it cross-checks
Single-commit (one declared op)	∼20 s	One committed root against its adjacent committed root
Layer-partial (forward *or* backward of one layer)	∼1.8 min	That layer's input/output (or gradient) boundary
Depth-6 partial chain (forward *or* backward)	∼11 min	One PP stage: intra-stage roots + one PP boundary
Full step (all 126 layers, forward + backward)	∼7 h	Every intermediate root, every anchor observation

Table 7: Proving time at Llama 3.1 405B scale on 1,024 GPUs. Within a proof, layer-level sub-proofs run in parallel on independent GPU subsets and are folded via recursive composition into a single ∼200 KB aggregate. Linear scaling in cluster size: a full-step proof on 16,384 GPUs completes in ∼25 minutes for the same total GPU-hours.

A realistic challenge mix combines shallow challenges for broad coverage (many, cheap) with occasional deep challenges that cross-check boundary commitments (few, expensive). The exact mix is an operational decision set by the auditor's risk budget.

B.8 Training-side hashing overhead

B.8.1 What must be hashed per step

Six tensors per layer are committed during training (Item **MOD. 3**): layer input, Q, K, V, attention output, layer output. At Llama 3.1 405B with $s = 2{,}048$, $m = 2{,}048$, activations are $m \cdot d_{\text{model}} \cdot 2\,\text{B} = 67\,\text{MB}$ each for the layer input/output and $m \cdot n_{\text{heads}} \cdot d_{\text{head}} \cdot 2 = 67\,\text{MB}$ for Q/O, $m \cdot n_{\text{kv\_heads}} \cdot d_{\text{head}} \cdot 2 = 4\,\text{MB}$ for K/V. Per layer, aggregate tensor bytes to hash are ∼275 MB. Over 126 layers per step the total is ∼35 GB.

The concurrent GPU stream delivers ∼80–120 GB/s effective throughput (CUDA hashing kernel, H100), putting per-step concurrent hashing time at ∼300–450 ms. At typical step times of ∼1 s this is within the overlap window with ∼1–3% memory-bandwidth contention on the primary compute stream.

Wire-bound collective payloads are hashed by the network anchor (Tier 1 physical TAP or Tier 2 attested SmartNIC). Anchor hashing happens outside the GPU budget and does not contribute to trainer-side overhead.

B.8.2 DPU/SmartNIC hashing capabilities (Tier 2 context)

For Tier 2 deployments the anchor lives in the SmartNIC. Relevant hashing throughput figures:

Device	SHA-256 throughput	Mechanism	Notes
NVIDIA BF3	∼10 GB/s	Software on 16× ARM A78 cores	Hardware SHA-256 removed vs BF2
NVIDIA BF4 (expected)	∼40 GB/s	64× Neoverse V2 cores	Announced; throughput TBD
Marvell OCTEON 10	∼12–25 GB/s	Hardware NITROX V	Commercially available
Dedicated FPGA (Tier 1)	∼100+ GB/s	Pipelined SHA-256 IP	Per OP-6, design target

Table 8: Observed or projected SHA-256 throughput for network-anchor-side hashing hardware. BF3 software-only throughput is marginal for cluster-scale inter-node traffic; hardware-accelerated silicon (Marvell OCTEON, BF4, or a dedicated Tier-1 FPGA) is required to sustain line rate at frontier cluster fabrics.

B.8.3 Total training-side overhead

Combining determinism and hashing costs, the total training-side overhead at Llama 3.1 405B is Table 9.

Component	Overhead	Confidence
Determinism (attention backward-dominated)	1.6–8.2%	Measured on Llama 7B H100; seq-len-dependent
Concurrent Merkle hashing (6 tensors/layer, GPU stream)	0.5–1.5%	Estimate from 2-pass kernel + optimised target
Weight rolling-window storage (81 TB)	$< 0.1\%$	Small vs cluster's existing checkpointing storage
Anchor-side hashing	$\sim$0% (trainer)	Moves to anchor hardware
Total trainer-side overhead	$\sim$2–10%	Sequence-length-dependent

Table 9: Training-side overhead at Llama 3.1 405B. Determinism is the dominant cost and is attention-backward-driven, scaling with sequence length. For a \$100M training budget the end-to-end trainer-side verification overhead is $\sim$\$2–10M.

B.9 Cost summary at Llama 3.1 405B

Component	Cost at \$100M training budget	% of training
Determinism tax	\$1.6–8.2M	1.6–8.2%
Concurrent Merkle hashing	\$0.5–1.5M	0.5–1.5%
Genesis proof (one-time, reduced-batch)	$\sim$\$1K	<0.01%
In-training challenges (layered mix: 10k single-commit + 500 layer-partial + 100 depth-6 + 10 full-step)	$\sim$\$326K	$\sim$0.33%
Weight storage (81 TB rolling)	$\sim$\$50K/year	<0.1%
Ex-ante attestation running cost	negligible	<0.001%
Total	**$\sim$\$2–10M**	**$\sim$2–10%**

Table 10: Verification overhead for a Llama 3.1 405B-scale pre-training run on a \$100M training budget. The dominant cost is the determinism tax; zkVM proving, hashing, storage, and challenges combined add $\sim$1–3%.

B.10 Caveats and open questions

1. **Proving throughput ($\sim 10^6$ constraints/s/GPU)** is approximate and may be optimistic for constraint-heavy FP precompiles. A direct measurement is part of the planned empirical validation work. Relative comparisons between approaches are robust to this assumption.
2. **Per-MAC constraint count ($\sim$90)** has a range of 50–150 depending on implementation. Absolute proof times scale linearly with this; the relative structure of the breakdown does not.
3. **Non-linear lookup tables** (BF16 GELU, SiLU) must match the GPU kernel output at every input. Generation and validation of these tables is routine engineering work.
4. **Sampling security at $f < 1\%$** is a defense-in-depth argument, not a hard theorem. Formal sensitivity analysis (OP-1) would close this.
5. **GPU-concurrent Merkle hashing overhead ($\sim$1–3%)** needs empirical validation on H100 with realistic training workloads.
6. **Fusion assumptions** (Flash Attention truly fused, FFN GEMMs separable with lazy commitment) are kernel-dependent. A training stack that fuses FFN end-to-end (e.g. a memory-bound variant) materially increases per-sample FFN verification cost, and the sample count for fused blocks must be reduced correspondingly.
7. **MoE cost numbers** depend on sampling-scheme design decisions open under OP-10. The order-of-magnitude framing ($K\times$ per-expert-dense) holds across reasonable choices; concrete per-architecture figures require OP-10 resolution.
8. **INT8 training with INT32 accumulation** (supported by H100/B200 Tensor Cores) would eliminate float non-associativity and make Freivalds directly applicable. No frontier model currently uses full INT8 pretraining, but the industry trend toward FP8 and FP4 is adjacent; verification cost at low-precision regimes is a worthwhile future study.

C Determinism enforcement: detailed benchmarks and analysis

This appendix provides detailed measurements and analysis supporting the determinism discussion in Item **MOD. 1**.

C.1 Kernel-level determinism configuration

We enforce determinism through optimised kernel selection rather than PyTorch's blanket `torch.use_deterministic_algorithms(True)` flag. The configuration uses: FlashAttention-2 with its deterministic backward mode (`FLASH_ATTENTION_DETERMINISTIC=1`), which serializes CTA reductions at a measured cost; cuBLAS with pinned algorithm selection (`CUBLAS_WORKSPACE_CONFIG`); and NCCL with NVLS-based reduction (which provides fixed reduction order at near-zero overhead for FSDP allgather and reduce-scatter).

We measured the end-to-end overhead of this configuration on 8× H100 HBM3 with NVSwitch, training Llama 7B with FSDP:

Seq. length	Non-det (ms/step)	Det (ms/step)	Overhead
2048	531.8	540.3	+1.6%
4096	999.1	1038.0	+3.9%
8192	1129.9	1222.0	+8.2%

Table 11: Measured full-determinism overhead (Llama 7B, FSDP, 8×H100 HBM3). The overhead is attention-dominated: cuBLAS GEMMs contribute ∼0.8% and NCCL communication (NVLS) contributes ∼0%.

C.2 Parallelism dimensions beyond FSDP

The measurements above apply to FSDP, which uses reduce-scatter and allgather (both deterministic under NVLS at near-full bandwidth). Frontier training runs typically combine multiple parallelism dimensions: tensor parallelism (TP) within a node, pipeline parallelism (PP) and FSDP across nodes, and expert parallelism (EP) for mixture-of-experts models.

Tensor parallelism. TP is the most challenging dimension. It uses allreduce after each column-parallel and row-parallel GEMM. Our NCCL benchmarks show that intra-node NVSwitch allreduce is *not* deterministic under NVLS above 128 MB BF16. The only deterministic configurations are Tree+Simple (64% bandwidth loss) and Ring+1CTA (89% bandwidth loss, unusable). This is a significant penalty: TP allreduce occurs at every layer, so even Tree+Simple could add 6–10% to total step time depending on the ratio of TP communication to compute.

Interestingly, inter-node allreduce (over TCP/RoCE) is deterministic under default NCCL configuration at all tested sizes (32–1024 MB BF16, 2-node setup). The determinism problem is specific to the intra-node NVSwitch code path.

Pipeline and expert parallelism. PP is straightforward: it sends activations between pipeline stages in a fixed schedule, and determinism reduces to the same per-layer guarantees described above. EP introduces all-to-all communication for expert routing. Our benchmarks show NVLS all-to-all is deterministic at all tested sizes, and the routing decisions themselves are deterministic given a fixed gating function.

C.3 Towards a custom TP allreduce

The overhead of Tree+Simple for TP can likely be eliminated by a custom deterministic allreduce kernel optimised for the TP use case. TP operates under narrow, exploitable constraints: fixed topology (8 GPUs on NVSwitch), fixed message sizes (determined by model dimensions, constant across steps), and a fixed reduction order is all that is needed for bit-exactness. A custom kernel using NVLink direct writes with a hardcoded reduction tree for the specific topology and message size could match NCCL's default bandwidth while guaranteeing determinism by construction. Projects such as MSCCL (Microsoft's custom collective compiler) demonstrate that hand-written collectives can match or exceed NCCL for fixed topologies.

Importantly, PyTorch's process group architecture supports per-dimension configuration: TP, FSDP, and PP each use separate communicators. The custom kernel would replace NCCL only for TP allreduce calls, while FSDP and PP continue to use NVLS under default configuration. This avoids any performance regression on the already-solved FSDP and PP dimensions.

Our current measured overhead (1.6–8.2%) applies to FSDP-only training. For FSDP+TP setups, the E2E impact of the TP allreduce bandwidth loss is likely small in practice: TP communication and layer computation overlap, and for typical TP=8 intra-node configurations the compute time dominates. For example, a 100 MB allreduce at Tree+Simple bandwidth (169 GB/s) takes $\sim$0.6 ms, which is well within the compute time of a single Transformer layer at frontier scale. The overhead would become significant only if TP communication becomes the bottleneck, for instance at very large TP degrees across nodes or with unusually small per-GPU compute. This remains to be validated end-to-end.

D Network anchoring: hash choice, reconciliation, and open problems

This appendix supports Item **MOD. 4** by (i) justifying the dual-hash design, (ii) describing the tree-structure and chunk-size tradeoffs, (iii) discussing composite precompile optimisations, (iv) developing the wire-to-tensor mapping problem in detail, and (v) listing the remaining open engineering problems.

D.1 Hash function choice: two regimes, two hashes

The architecture uses two distinct hash functions for its Merkle commitments: a zkVM-native hash (precompile 3, Poseidon today) for the trainer's on-the-fly tensor commitments, and a silicon-native hash (precompile 8, SHA-256) for the network anchor. This is not redundant: each hash is chosen for the bottleneck of the device that hashes at line rate.

The two hash families have opposite cost profiles. Poseidon operates in the trainer's native prime field: its round function is built from $x^5 \bmod p$ S-boxes and MDS matrix multiplications over $\mathbb{F}_p$. The same arithmetic that makes Poseidon inexpensive inside the proof ($\sim$75 constraints per compression, $\sim$1,500 constraints per path of depth 30) makes it expensive outside the proof: prime-field multiplication is not a native silicon primitive, and commercial FPGAs and DPUs achieve at best sub-GB/s throughput with significant area cost. SHA-256 inverts this. Its round function uses 32-bit bitwise operations (AND, XOR, rotation) and modular addition, which map directly to Boolean gates and integer ALUs: Intel SHA-NI sustains $\sim$1.9 cycles/byte, and FPGA IP cores pipeline at 5–20 Gbps per core and stack to hundreds of GB/s per device. The same simplicity is costly in-circuit, because bitwise operations over 32-bit words must be encoded via lookup tables or bit decomposition ($\sim$7,500 constraints per compression, $\sim$150,000 constraints per path of depth 20).

The asymmetry in cluster-scale throughput is therefore two to three orders of magnitude. Aggregate inter-node bandwidth on a modern training cluster reaches several hundred GB/s; Poseidon-family hashes cannot be computed at those rates on commodity silicon. The network anchor must use a hash with mature silicon support, which today means SHA-256. Conversely, the trainer's Merkle paths are opened thousands of times per sampled GEMM inside the zkVM; using SHA-256 for those paths would increase the proof budget for path verification by roughly $100\times$, whereas the GPU overhead saved on the trainer side is under 1%. Using one hash per regime is the only design that is efficient on both sides.

This split between an in-circuit-cheap hash and a silicon-cheap hash is a standard pattern whenever a system bridges two cost models with incompatible primitives; Ethereum rollups, for example, use Keccak at the L1 interface and zk-friendly hashes inside the rollup for the same reason.

D.2 Tree versus flat, and chunk size

Committing to wire payloads as a Merkle tree rather than a single flat hash trades a small amount of TAP-side work for a large reduction in challenge-time zkVM cost. Flat SHA-256 commits a payload with one digest; verifying consistency at challenge time then requires rehashing the entire payload inside the zkVM. For a 1 GB collective with the SHA-256 precompile this is roughly 16×10^6 blocks at $\sim$7,500 constraints each, or $\sim 10^{11}$ constraints per challenge. Tree-structured commitments reduce this to $O(\log n)$ path verification plus a local chunk rehash.

The tree structure costs almost nothing on the TAP side. For n leaves of size ℓ, the total hashing volume is $\sim 2n$ leaf-sized units, so the overhead over flat hashing is a small constant factor determined by ℓ/ℓ_{leaf}: with MB-sized leaves over a GB-sized payload it is at most $1.001\times$.

The leaf size ℓ is a tuning parameter. Smaller leaves shrink the chunk rehash but deepen the tree, inflating path verification. Larger leaves invert the trade.

Leaf size	Tree depth (1 GB)	Path cost	Chunk rehash cost	Total per opening
32 B (one element)	25	$\sim$190,000	$\sim$7,500	$\sim$200,000
16 KB	16	$\sim$120,000	$\sim$1,900,000	$\sim$2,000,000
64 KB	14	$\sim$105,000	$\sim$7,500,000	$\sim$7,600,000
1 MB	10	$\sim$75,000	$\sim$120,000,000	$\sim$120,000,000

Table 12: Constraint cost per wire-bound sample opening for a 1 GB collective as a function of Merkle-leaf size. Path and rehash costs assume the SHA-256 precompile at $\sim$7,500 constraints per compression block.

The zkVM cost is minimised at small leaves, which places the sweet spot at tens of kilobytes when balanced against TAP-side chunk-boundary bookkeeping and packet-boundary alignment (discussed below). Values in the 16–64 KB range give opening costs in the 2–8 million constraint range per sample, bounded and much smaller than the MAC-chain budget of a GEMM.

D.3 Composite Merkle-path precompile

The precompile catalogue in Table 3 reports costs assuming per-compression precompile invocations composed in software (a loop over tree levels calling the SHA-256 or Poseidon primitive at each step). A composite precompile that absorbs path-traversal logic (sibling loading, direction bits, chained compressions) into a single dedicated circuit eliminates the per-level RISC-V overhead and enables tighter constraint layouts across levels. Cairo's Merkle builtins and ongoing work in SP1 and Jolt follow this pattern. Estimated savings are 20–30% for SHA-256 paths (where the compressor is heavy and glue overhead relatively matters) and 15–25% for Poseidon paths. A further optimisation is a multi-path precompile that verifies several paths in one invocation, amortising transition cost across openings; this gives another 10–20% when the same challenge opens multiple paths (as is typical for GEMM sampling, which opens 3–5 paths per sampled entry).

These are implementation-level optimisations; they refine the proof-cost figures without changing the architectural claims.

Several open engineering problems attach to the network anchor: the hardware TAP itself, wire-to-tensor mapping (with queue-pair auditability, Tier 2 attestation protocol, and chunk-boundary alignment as subsidiary questions), and intra-node coverage. These are catalogued in Section A (OP-6 through OP-8) rather than duplicated here.

E Toy example: end-to-end verification walkthrough

This appendix walks through the complete verification protocol on a minimal example: a two-layer MLP with ReLU activation, split across two nodes by pipeline parallelism. The example exercises precompiles 1 (MAC), 2 (BF16 lookup), 3 (Merkle path with the zkVM-native hash), and 8 (SHA-256 for network-anchor reconciliation). It does not exercise precompiles 4–7 (FP32 nonlinear operations for softmax, normalisation, Adam, or loss); those are triggered by additional operations in the full Transformer block and follow the same pattern.

E.1 Setup

The model is a two-layer MLP, $f(x) = W_2 \cdot \mathrm{ReLU}(W_1 \cdot x)$, trained with pipeline parallelism over two nodes:

- **Node 1** holds the first linear layer $W_1 \in \mathbb{R}^{d_h \times d_{\text{in}}}$.
- **Node 2** holds the second linear layer $W_2 \in \mathbb{R}^{d_{\text{out}} \times d_h}$.
- Loss is $L = \frac{1}{2}\|y - \text{target}\|^2$ for simplicity of the backward pass.
- Optimiser is plain SGD ($W \leftarrow W - \eta \cdot \text{grad}$); no Adam state to track.
- Weights are BF16 with FP32 accumulators; rounding is RNE.

Each training step involves two inter-node messages: a forward activation passed from Node 1 to Node 2, and a backward gradient passed from Node 2 back to Node 1. The network anchor observes both.

Notation. We write $R(\cdot)$ for a Merkle root over a tensor or byte sequence. Trainer-side Merkle roots use a zkVM-native hash (Poseidon in the current instantiation, verified by precompile 3). Network-anchor Merkle roots use a silicon-native hash (SHA-256, verified by precompile 8). Both support $\log n$ membership proofs against the committed root; they differ only in per-path cost inside the proof. We write $\mathrm{Hash}(\cdot)$ for the outer compound commitment.

E.2 Phase 1: pre-commit

Before training begins, the trainer publishes the initial commitment:

$$h_{\text{commit}} = \mathrm{Hash}\big(\mathrm{Hash}(\texttt{arch\_spec}) \parallel R(W_0) \parallel R(\texttt{dataset})\big)$$

where `arch_spec` records the layer dimensions, activation type (ReLU), optimiser (SGD), precision (BF16/FP32, RNE), parallelism strategy (PP, world size 2), and shuffling seed. $R(W_0)$ is the Merkle root over the initial weight shards across both nodes; $R(\texttt{dataset})$ is the Merkle root over the training data.

The auditor records h_{commit} and installs a network anchor on the inter-node link between Node 1 and Node 2.

E.3 Phase 2: one training step

We annotate a single step with every commitment published and every message observed. Node 1 processes batch shard x; arrows mark inter-node traffic.

The two nodes execute sequentially along the pipeline direction. Arrows mark inter-node messages observed by the network anchor.

Node 1 (holds W_1).

```
mlp_1_in  <- x                                          # batch shard
mlp_1_out <- W_1 @ mlp_1_in
act_1_out <- ReLU(mlp_1_out)
publish R(mlp_1_in), R(mlp_1_out), R(act_1_out)
send act_1_out  ---- forward msg_1 ---->  Node 2      (anchor: R(msg_1))
```

Node 2 (holds W_2).

```
recv act_1_out  from Node 1
mlp_2_in  <- act_1_out
mlp_2_out <- W_2 @ mlp_2_in
act_2_out <- ReLU(mlp_2_out)
publish R(mlp_2_in), R(mlp_2_out), R(act_2_out)
L <- 0.5 * || act_2_out - target ||^2
grad_act_2 <- act_2_out - target
grad_mlp_2 <- grad_act_2 * (mlp_2_out > 0)
grad_W_2   <- grad_mlp_2 @ mlp_2_in^T
grad_act_1 <- W_2^T @ grad_mlp_2
publish R(grad_2)
send grad_act_1  ---- backward msg_2 ---->  Node 1    (anchor: R(msg_2))
```

Node 1 (backward + optimiser).

```
recv grad_act_1  from Node 2
grad_mlp_1 <- grad_act_1 * (mlp_1_out > 0)
grad_W_1   <- grad_mlp_1 @ mlp_1_in^T
publish R(grad_1)
```

```
W_1 <- W_1 - eta * grad_W_1            # at Node 1
W_2 <- W_2 - eta * grad_W_2            # at Node 2
publish R(W_{t+1})                     # combined root
```

After the step, the trainer's public hash chain contains $R(\texttt{archi})$, $R(W_t)$, $R(\texttt{data})$ (from pre-commit), plus per-step $R(\texttt{mlp}_i\texttt{\_in})$, $R(\texttt{mlp}_i\texttt{\_out})$, $R(\texttt{act}_i\texttt{\_out})$, $R(\texttt{grad}_i)$ for $i \in \{1, 2\}$, and $R(W_{t+1})$. The network anchor's record contains $R(\texttt{msg}_1)$ (forward activation payload) and $R(\texttt{msg}_2)$ (backward gradient payload).

E.4 Phase 3: challenge

The auditor selects a step t, a node i, and k random sample indices $(j_1, \ldots, j_k)$ for the output entries to challenge. Because the indices are chosen *after* the Merkle roots have been published to the hash chain, the trainer cannot predict which entries will be sampled.

The challenge demands a proof of the following statement, phrased at Node 1 for concreteness (Figure in Section 3):

$$R(\texttt{archi}),\ R(W_t),\ R(\texttt{data}) \vdash R(\texttt{mlp}_1\texttt{\_in}) \rightarrow R(\texttt{mlp}_1\texttt{\_out}) \rightarrow R(\texttt{msg}_1)$$

That is: executing the committed architecture with the committed weights on the committed data produces intermediate tensors whose Merkle roots match the committed $R(\texttt{mlp}_1\texttt{\_in})$ and $R(\texttt{mlp}_1\texttt{\_out})$, and the forward message serialisation is consistent with the network anchor's observation $R(\texttt{msg}_1)$.

E.5 Phase 4: verification inside the zkVM

The proof checker is a public, auditable program. It reads the `arch_spec` as a private input, verifies it matches h_{commit}, then, for each sampled output entry, runs the following checks. We illustrate one entry at Node 1: the output of the first GEMM at row r, column j.

```
# ---- bind the private arch_spec to the public commitment ----
arch_spec = read_private_input()
assert Hash(arch_spec) == h_commit.arch_spec_hash

# ---- for each sampled (r, j) at Node 1 ----
for (r, j) in challenge_indices:

    # (a) input row: open against R(mlp_1_in)
    x_row    = read_hint()                 # private: mlp_1_in[r, :]
    path_in  = read_hint()
    MerklePath(x_row, path_in, R(mlp_1_in))          # precompile 3

    # (b) weight column: open against R(W_t)
    W1_col   = read_hint()                 # private: W_1[:, j]
    path_W   = read_hint()
    MerklePath(W1_col, path_W, R(W_t))               # precompile 3

    # (c) GEMM entry: recompute via MAC chain and bind to output root
    mlp_out_rj = read_hint()               # private: mlp_1_out[r, j]
    MAC_chain(x_row, W1_col, mlp_out_rj)             # precompile 1
    path_out = read_hint()
    MerklePath(mlp_out_rj, path_out, R(mlp_1_out))   # precompile 3

    # (d) ReLU output: verify via BF16 lookup, bind to activation root
    act_rj   = read_hint()                 # private: act_1_out[r, j]
    Lookup(mlp_out_rj, ReLU_TABLE, act_rj)           # precompile 2
    path_act = read_hint()
    MerklePath(act_rj, path_act, R(act_1_out))       # precompile 3

    # (e) anchor consistency: the same act_rj must appear at the
    #     computed byte offset in the forward message
```

```
    offset   = compute_msg_offset(arch_spec, r, j)
    chunk    = read_hint()                  # bytes of the msg chunk
    path_msg = read_hint()
    MerklePath_SHA256(SHA256(chunk), path_msg, R(msg_1))  # precompile 8
    assert chunk[offset : offset + bf16_size] == act_rj

emit "PASS"
```

Step (e) is the link between the trainer's tensor commitment and the network anchor's wire commitment: the same BF16 value of $\texttt{act}_1\texttt{\_out}[r, j]$ must be present both under the trainer-committed $R(\texttt{act}_1\texttt{\_out})$ (via Poseidon path) and under the anchor-committed $R(\texttt{msg}_1)$ (via SHA-256 path at the offset dictated by the serialisation defined in `arch_spec`).

The guest emits as public outputs: h_{commit}, the committed roots $R(\texttt{mlp}_1\texttt{\_in})$, $R(\texttt{mlp}_1\texttt{\_out})$, $R(\texttt{act}_1\texttt{\_out})$, the anchor root $R(\texttt{msg}_1)$, and a "PASS" bit. The auditor STARK-verifies the proof (milliseconds on a laptop), checks that the public outputs match the published hash chain and the anchor record, and accepts.

E.6 What each party sees

	Trainer (host)	Proof checker (guest)	Auditor
`arch_spec`	full	private input, bound to h_{commit}	only $\text{Hash}(\texttt{arch\_spec})$
Weights W_1, W_2	full	hints, bound to $R(W_t)$	never
Activations, gradients	full	hints, bound via MAC + lookup	never
Merkle roots $R(\cdot)$	computes	public verification anchors	reads from hash chain
Anchor root $R(\texttt{msg}_k)$	not involved	public verification anchor	reads from anchor
STARK proof	not involved	produces it	verifies it

E.7 Proof cost

For illustration, take $d_{\text{in}} = d_h = d_{\text{out}} = 1024$ and $k = 100$ sampled output entries per committed tensor at each of the two nodes (a small value chosen for readability; the main-text analysis uses $k \approx 4{,}605$ for 10^{-20} soundness error at 1% deviation). The approximate constraint budget for one step:

Verification work	Precompile	Constraints (order of magnitude)
First GEMM ($d_h \cdot 90$ per sample $\times k$)	MAC (1)	$\sim 9 \times 10^6$
ReLU lookup (k per committed element)	BF16 lookup (2)	$\sim 1.5 \times 10^3$
Second GEMM (same magnitude)	MAC (1)	$\sim 9 \times 10^6$
Merkle paths, trainer side ($\sim$6 per sample $\times k$)	Poseidon path (3)	$\sim 9 \times 10^5$
Anchor consistency (1 path + 1 chunk rehash per sample)	SHA-256 (8)	$\sim 6 \times 10^6$

Total: roughly 2.5×10^7 constraints per step, which is well inside the capacity of current zkVMs (STARK proof size $\sim$200 KB after recursive composition). At production scale the MAC chain dominates because d and k both grow by two to three orders of magnitude; see Section B for the full breakdown.

F Training specification (`arch_spec`) schema

This appendix gives the concrete structure of the `arch_spec` committed during pre-training (Item **MOD. 2**) and verified by the proof checker (Section 3.1). The `arch_spec` is a private input to the proof; the auditor sees only $\text{Hash}(\texttt{arch\_spec})$ as part of h_{commit}. The schema below is the reference definition against which implementations are written. It is presented in Rust-style record syntax for concreteness; any structured serialisation with an equivalent set of fields is acceptable.

F.1 Top-level `arch_spec`

```
struct ArchSpec {
    // Model architecture
    layers:       Vec<LayerSpec>,
```

```
    embedding:    EmbeddingSpec,
    output_head:  OutputHeadSpec,

    // Training configuration
    optimizer:    OptimizerSpec,         // Adam, AdamW, SGD, ...
    data_loading: DataLoadingSpec,       // batch derivation from committed dataset
    precision:    PrecisionSpec,         // BF16/FP32, rounding, hardware

    // Distributed training
    communication: CommSpec,             // allreduce, allgather, send/recv
    parallelism:   ParallelismSpec,      // TP, PP, DP, FSDP configuration

    // Verification
    merkle_points: Vec<MerklePoint>,     // tensors committed on-the-fly
    rng_spec:      RNGSpec,              // data-shuffling and dropout seeds
}
```

Every operation that takes place during a training step must be expressible as a composition of the elements declared by `arch_spec`. Any operation outside this vocabulary is rejected by the proof checker.

F.2 Layer specification

```
struct LayerSpec {
    forward_ops:   Vec<OpSpec>,
    backward_ops:  Vec<OpSpec>,          // gradient computations mirroring forward
    optimizer_ops: Vec<OpSpec>,          // parameter update for this layer
}

enum OpSpec {
    // Linear algebra
    GEMM { m: usize, k: usize, n: usize },

    // Attention (fused; internal score and softmax tensors never materialise;
    // verified end-to-end at Q/K/V/attention-output boundaries, see Sec. 3.4.5)
    FusedAttention {
        n_heads:     usize,
        n_kv_heads:  usize,              // GQA: < n_heads ; MHA: == n_heads
        d_head:      usize,
        seq_len:     usize,
        causal:      bool,
        positional:  PositionalEncoding, // RoPE, Learned, ALiBi, None
    },

    // Non-linear activations (BF16, verified via precompile 2)
    Activation { kind: ActivationType },  // GELU, SiLU, SwiGLU, ReLU

    // Normalisation (RMSNorm used by Llama family)
    LayerNorm { dim: usize },
    RMSNorm   { dim: usize },

    // Structural
    Residual,

    // Mixture-of-experts block (out of scope for base protocol; see OP-10)
    MoEBlock {
        router:   RouterSpec,
        experts:  Vec<Vec<OpSpec>>,
        combine:  CombineSpec,
    },
}
```

A forward `GEMM` with inputs A and B produces two backward `GEMM` operations: one for the gradient with respect to the input ($\text{grad}_A = \text{grad}_C \cdot B^\top$) and one for the gradient with respect to the weight ($\text{grad}_B = A^\top \cdot \text{grad}_C$). The `backward_ops` list enumerates these explicitly.

Fused operations carry both a forward and a backward variant; both are sampled end-to-end at their input and output commitment boundaries (Section B.3.1).

F.3 Precision specification

```
struct PrecisionSpec {
    compute:           Precision,        // BF16 for parameters/activations
    accumulate:        Precision,        // FP32 for GEMM accumulators
    rounding:          RoundingMode,     // RoundToNearestEven
    accum_order:       AccumOrder,       // LinearLeftToRight, ...
    hardware:          HardwareTarget,   // H100, MI300X, ...
    cublas_algo:       Option<String>,   // pinned cuBLAS algorithm ID
    flash_attn_version: String,          // e.g. "FlashAttention-3-deterministic"
}
```

The `accum_order` field is load-bearing for determinism: given non-associative floating-point addition, two distinct orders produce two distinct bit-exact results. The committed order is what the proof checker verifies.

F.4 Communication specification

`CommSpec` governs operations that perform floating-point reductions (`AllReduce`, `ReduceScatter`). The reduction order is explicit so the proof can verify the arithmetic. Data-movement operations that do not perform reductions (pipeline-parallel send/recv, FSDP allgather) are verified through Merkle-root matching at endpoints plus network-anchor consistency, without reduction-order specification.

```
struct CommSpec {
    // Global NCCL configuration pinned for determinism (Sec. 3.3.1)
    nccl_algo:     NCCLAlgo,             // Ring, Tree, NVLS
    nccl_proto:    NCCLProto,            // Simple
    nccl_channels: usize,

    ops: Vec<CommOp>,
}

enum CommOp {
    AllReduce {
        tensor_shape:    Vec<usize>,
        dtype:           Dtype,
        reduction_order: Vec<usize>,     // explicit FP addition order
        group:           ProcessGroup,
    },
    AllGather {
        shard_shape:  Vec<usize>,
        dtype:        Dtype,
        gather_order: Vec<usize>,        // rank-order (0, 1, 2, ...)
        group:        ProcessGroup,
    },
    ReduceScatter {
        tensor_shape:    Vec<usize>,
        dtype:           Dtype,
        reduction_order: Vec<usize>,
        group:           ProcessGroup,
    },
    Send { tensor_shape: Vec<usize>, src: usize, dst: usize },
    Recv { tensor_shape: Vec<usize>, src: usize, dst: usize },
}
```

F.5 Parallelism specification

```
struct ParallelismSpec {
    tp_degree:        usize,             // tensor parallelism (typically 8)
    pp_stages:        usize,             // pipeline parallelism stages
    dp_degree:        usize,             // data parallelism replicas
    fsdp_enabled:     bool,
```

```
    fsdp_shard_degree: usize,            // ranks sharing each parameter shard

    // Expert parallelism (MoE models only; out of scope for base protocol)
    ep: Option<ExpertParallelismSpec>,

    // Process-group topology
    tp_groups: Vec<Vec<usize>>,
    pp_groups: Vec<Vec<usize>>,
    dp_groups: Vec<Vec<usize>>,
}
```

F.6 Commitment structure

The outer public commitment, re-stated here for reference, combines the `arch_spec` hash with the Merkle roots of initial weights and dataset:

$$h_{\text{commit}} = \text{Hash}\big(\text{Hash}(\texttt{arch\_spec}) \parallel R(W_0) \parallel R(\texttt{dataset})\big).$$

Any ex-ante attestation claims (Item **P. 3**) extend this structure by appending $\text{Hash}(\texttt{ex\_ante\_claims})$.

F.7 Converter pipeline

Generating the `arch_spec` from the trainer's actual PyTorch (or JAX) training code requires a converter pipeline; this is treated as its own open problem (see OP-13 in Section A).

G Protocol-aligned formalization of execution verification

Convention. In this appendix and in Section G.6.1, we revert to the cryptographic convention: the *prover* is the trainer of Section 3, and the *verifier* is the auditor.

In the scope of this paper, our protocol in Section 3.2 proves a narrower object: *relation-qualified execution-conformance* to a declared training-execution relation. The verifier never evaluates a held-out loss $L_{\mathcal{T}}(M)$. What is checked is that the public commitments, TAP tags, and zkVM proofs are jointly consistent with a witness for the committed run. The PAC-style definitions in Section G.6.1 motivate a complexity-theoretic viewpoint, for a more general *AI model* verification, encompassing learning capacity. This PAC-formalization in Section G.6.1 generalizes existing approaches [15, 2] that focus mainly on ML models.

G.1 Background notions

NP relations and arguments. An NP relation is a polynomial-time decidable predicate $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{W}$ on a public instance $x \in \mathcal{X}$ and a private witness $w \in \mathcal{W}$. Here x is the public training transcript: commitments, tags, public hyperparameters, the genesis challenge, verifier signatures, and the opened sampling seed after streaming has been frozen. The witness w is the private execution trace: architecture specification, compiled program, data, weights, batches, randomness, optimizer state, intermediate tensors, and traffic. Because the protocol relies on computational assumptions, the proof object is formally an *argument* rather than an information-theoretic proof: a polynomial-time cheating prover should not make the verifier accept a false instance except with the stated soundness error.

Commitments and Merkle binding. We use two kinds of commitments. The tensor and weight commitments are Merkle-style commitments to values that the prover later opens inside audited proofs. Their relevant property for soundness is binding: after a root has been published, the prover cannot open the same root to two different tensor values except with probability $\varepsilon_{\text{bind}}$. The verifier's seed commitment $\text{Com}(\sigma; \rho)$ has two distinct roles. Binding prevents the verifier from changing σ after the transcript is frozen, while hiding prevents the prover from learning σ during training. For soundness we use the prover-side prediction advantage $\varepsilon_{\text{com,pred}}$, not the verifier-side CZK hiding term $\varepsilon_{\text{com,hide}}$: for standard hiding commitments $\varepsilon_{\text{com,pred}}$ is bounded by the usual hiding advantage, but the games are conceptually different.

Commit-then-reveal sampling. Before training, the verifier samples a seed σ and publishes $\mathrm{Com}(\sigma;\rho)$. During training, the prover streams $(\mathrm{Com}(w_t), h_t)$ and the public anchor chain binds these values in order. Only after the terminal anchor is signed by the verifier is (σ,ρ) opened. The step set S, audited layers, and audited entries are then derived from F_σ using fixed-length encodings and separate tags

$$\texttt{SAMP/STEP}, \qquad \texttt{SAMP/LAYER}, \qquad \texttt{SAMP/ENTRY}.$$

This ordering is load-bearing: if σ is known before the stream is frozen, an adaptive prover can place deviations entirely outside the audited positions. The terminal anchor freezes only the ordered public commitment stream. It does not certify that the prover still stores the private opening material for every old commitment, so availability of sampled openings is a separate operational precondition of the spot-check protocol.

Knowledge soundness. Knowledge soundness strengthens plain soundness by requiring that an accepting prover can be converted into an extractor that outputs a witness for the accepted statement. In this appendix the statement is deliberately qualitative. For the BCS-/FRI-style compiled STARK layer, the theorem-level accounting keeps the quadratic random-oracle contribution; outer commitment-chain extraction and wrapper extraction are separate losses [6, 5]. We do not package these terms into a clean deployed-stack 2^{-128} theorem.

Auxiliary-input computational zero knowledge. Auxiliary-input CZK means that for every polynomial-time verifier with auxiliary input z, there is a simulator whose output is computationally indistinguishable from the real proof transcript, up to the public transcript and accept/reject bit [14, 20]. The claim made here is sequential and per-proof: it covers the proof objects in one audit session under the usual programmable-random-oracle modeling of Fiat–Shamir. It is not a generic concurrent-composition, GUC, or EUC claim.

G.2 Protocol-aligned formalization

We formalise and analyse security of the protocol of Section 3.2 in the same manner as for proof systems for NP relations.[15] An **NP relation** $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{W}$ pairs a public instance x (commitments, tags, hyperparameters, genesis challenge, signatures, opened seed) with a private witness w (`arch_spec`, P_{prog}, data, weights, batches, randomness, optimizer state, intermediate tensors, traffic). **Merkle commitments** bind tensor and weight values with binding error $\varepsilon_{\mathrm{bind}}$. The verifier's **seed commitment** $\mathrm{Com}(\sigma;\rho)$ is additionally hiding; for soundness we use the prover-side prediction advantage $\varepsilon_{\mathrm{com,pred}}$ (bounded by hiding) rather than the CZK term $\varepsilon_{\mathrm{com,hide}}$. In **commit-then-reveal**, the verifier publishes $\mathrm{Com}(\sigma;\rho)$ before training; the prover streams $(\mathrm{Com}(w_t), h_t)$ into the anchor chain; (σ,ρ) opens only after the verifier signs the terminal anchor; S and audited positions are then derived from F_σ under tags `SAMP/STEP`, `SAMP/LAYER`, `SAMP/ENTRY`. Required properties[16] include **(i) knowledge soundness** that allows extracting the witness from convincing provers (BCS/FRI accounting retains the quadratic random-oracle term [6, 5]); **(ii) auxiliary-input computational ZK** [14, 20] — for every polynomial-time verifier with auxiliary input z, there is a simulator whose output is computationally indistinguishable from the real proof transcript, up to the public transcript and accept/reject bit; **(iii) completeness** ensures knowing valid witness will convince the verifier. Formal definitions of **(i)–(iii)** are given in Section G.3. Security analysis in view of above errors $\varepsilon_{\mathrm{bind}}$, $\varepsilon_{\mathrm{com,pred}}$, and $\varepsilon_{\mathrm{com,hide}}$, are given in Section G.4.

G.3 Universal target relation and sampled verifier predicate

The paper should keep separate the relation the protocol is trying to certify from the predicate actually opened by the current sparse-auditing protocol. The target relation $\mathcal{R}_{\mathrm{train}}$ is the universal execution relation over the full training run. The current spot-check verifier checks a sampled restriction, denoted $\mathcal{R}^{S,Q}_{\mathrm{spot}}$, and soundness for $\mathcal{R}_{\mathrm{train}}$ pays the M2/M3 sampling terms below.

[15] PAC-style generalisation is given in Section G.6, which has the potential to go beyond NP-reation and capture learnability, as previously done for ML models [2, 15].

[16] Sequential, per-proof, under programmable-RO modelling of Fiat–Shamir; no concurrent-composition, GUC, or EUC claim.

Definition 1 (Universal training-execution relation)**.** *Let x_{run} be the public run instance containing: (i) the pre-commitment*

$$R = H(H(\texttt{arch\_spec}) \,\|\, \mathrm{MerkleRoot}(\mathrm{dataset}) \,\|\, \mathrm{MerkleRoot}(w_0)),$$

(ii) public hyperparameters θ, including T and the training schedule, (iii) the verifier public key vk_V, (iv) the genesis challenge batch and TAP tag, (v) streamed commitments $\{\mathrm{Com}(w_t), h_t\}_{t=1}^{T}$, and (vi) the terminal anchor anchor_T and verifier signature σ_{chain}.

The witness w contains the private execution trace: `arch_spec`, the compiled zkVM program P_{prog}, the dataset, the weight trajectory $\{w_t\}_{t=0}^{T}$, the batches $\{B_t\}_{t=1}^{T}$, prover hints, intermediate tensors, optimizer state, registered stochastic-component states, and the raw network traffic $\{\mathrm{traffic}_t\}_{t=1}^{T}$. In the base TAP model, the keyed-hash key K is a fixed run secret in the prover/TAP trust base and hidden from the verifier; the proof either treats K as a private witness value shared with the TAP or treats TAP-tag correctness as an external functionality assumption.

We write $(x_{\mathrm{run}}, w) \in \mathcal{R}_{\mathrm{train}}$ if all of the following hold:

(U1) ***Commitment and anchor consistency.*** *The pre-commitment equation defining R holds; each $\mathrm{Com}(w_t)$ is the Merkle apex of the declared committed tensors and optimizer state for step t; and the streamed values form the anchor chain*

$$\begin{aligned}\mathrm{anchor}_0 &= H(\text{“ANCHOR/INIT”} \,\|\, R),\\ \mathrm{anchor}_t &= H\big(\text{“ANCHOR/LINK”} \,\|\, \mathrm{anchor}_{t-1}\\ &\qquad \|\, \mathrm{Com}(w_t) \,\|\, h_t \,\|\, t\big).\end{aligned}$$

The terminal signature verifies:

$$\mathsf{Verify}(\mathsf{vk}_V, \text{“ANCHOR/LINK”} \,\|\, R \,\|\, T \,\|\, \mathrm{anchor}_T, \sigma_{\mathrm{chain}}) = 1.$$

(U2) ***Program/spec binding.*** *The zkVM program is the publicly specified deterministic compilation of the committed architecture:*

$$P_{\mathrm{prog}} = f_{\mathrm{compile}}(\texttt{arch\_spec}).$$

(U3) ***Batch and RNG derivation.*** *The batch schedule and every stochastic component output are derived from the committed dataset root, the master RNG seed, and the registered stochastic-component list in `arch_spec`. Stateless randomness uses domain-separated keyed derivations such as `RNG/STEP` and `RNG/<COMP>` with fixed-length encodings. Stateful stochastic rules, such as dynamic loss scaling, are updated by their declared deterministic transition functions.*

(U4) ***Genesis.*** *The genesis execution on the verifier's challenge batch is consistent with P_{prog}, w_0, and the TAP tag*

$$h_{\mathrm{gen}} = F_K(\mathrm{traffic}_{\mathrm{gen}}).$$

(U5) ***Universal step consistency.*** *For every $t \in [T]$, the committed transition from (B_t, w_{t-1}) to w_t is the transition prescribed by P_{prog}, and the produced traffic matches the TAP tag:*

$$\mathsf{Exec}(P_{\mathrm{prog}}, B_t, w_{t-1}; \mathbf{h}_t) = (w_t, \mathrm{traffic}_t), \qquad h_t = F_K(\mathrm{traffic}_t).$$

(U6) ***Operation-level consistency.*** *Every committed operation value used in the transition is consistent with the relevant operation semantics. For GEMMs this means bit-exact MAC-chain consistency against the committed operands; for nonlinear and lookup-heavy operations this means the corresponding precompile semantics for exp, sqrt/rsqrt, division, log, BF16 lookup, attention blocks, and normalization paths.*

(U7) ***Layer-boundary consistency.*** *Layer boundaries chain correctly: a layer's committed output root is the next layer's committed input root wherever the architecture declares such a boundary.*

Definition 2 (Sampled spot-check predicate)**.** *Let the spot-check transcript be*

$$x_{\mathrm{spot}} = (x_{\mathrm{run}}, \mathrm{Com}(\sigma), \sigma, \rho, S, Q).$$

Here Q denotes all sampled layer and entry positions. The predicate $\mathcal{R}_{\mathrm{spot}}^{S,Q}$ first checks

$$\mathsf{Open}(\mathrm{Com}(\sigma), \sigma, \rho) = 1,$$

and then checks that S, the audited layers, and the audited entries are exactly the outputs of F_σ under the fixed tags `SAMP/STEP`, `SAMP/LAYER`, and `SAMP/ENTRY`, using the protocol's fixed-length encoding convention. It checks (U1)–(U4) as public consistency conditions, and it checks (U5)–(U7) only at the sampled steps, layers, and entries specified by (S, Q).

Thus the current proof objects are arguments for $\mathcal{R}_{\text{spot}}^{S,Q}$. They are arguments for the universal target relation $\mathcal{R}_{\text{train}}$ only up to the sampling miss terms in the soundness bound. In a V5-IVC architecture, the step restriction $t \in S$ is replaced by $t \in [T]$; this removes the M3 branch but replaces it with recursive-proof, commitment, and extractor costs. It does not by itself remove all operation-level sampling costs unless the per-step verifier is also strengthened.

G.4 Security analysis for Section 3.2

For ex-ante atttestations (P. 3 of Section 3.2) - Attack surface for compute under-declaration. Compute-threshold enforcement is our first target, and the adversary of interest under-declares FLOPs to stay below a regulatory tier. Three cheating modes are available: a larger model than declared, more training data than declared, or more training steps than declared. Each has a cheap detection path beyond the generic sampling argument. A wider or deeper model fails the first weight Merkle-path opening because commitment shapes do not match `arch_spec`. Extra training data fails input-Merkle openings at steps that use uncommitted samples: hiding a factor α of extra data forces $1 - 1/\alpha$ of openings to fail, which two challenges catch at 99% confidence for $\alpha = 10$. Extra training steps are visible to the network anchor, whose observed collective count exceeds the committed chain length by a single integer comparison. A compute-threshold attestation under these three detection mechanisms is cheaper to enforce than generic `arch_spec` verification against an adaptive adversary. Other attestation classes (for instance data-content filters targeting semantic smuggling, or procedural attestations on fine-grained regime compliance) lack this structural detection surface and require denser sampling. A case-by-case analysis is already done in Section B.4.3.

For ex-ante atttestations (P. 3 of Section 3.2) - Ex-post variant. Emergent properties of the completed run that are not knowable at initialisation (final-weight norms, benchmark performance under a public evaluator) admit a natural ex-post variant of the same $f(X) = y$ structure, executed once against the completed hash chain and final weights rather than maintained as a running invariant. Ex-post attestations do not modify the online-phase protocol; they reuse the precompile toolkit on final commitments.

Completeness. Suppose $(x_{\text{run}}, w) \in \mathcal{R}_{\text{train}}$, the prover follows the committed procedure honestly, the implementation satisfies the bit-exact determinism requirements of Item MOD. 1, and the verifier derives (S, Q) from a correctly opened σ. Then the verifier accepts the spot-check protocol with probability $1 - \text{negl}(\lambda)$.

Proof sketch. All public commitments and anchor links verify by (U1). The sampled predicate only restricts universal conditions (U5)–(U7), so every sampled transition, layer boundary, GEMM entry, and nonlinear/precompile check is correct. The TAP tags match by (U4)–(U5), and zkVM completeness gives accepting genesis and per-step proofs except with negligible proof-system completeness error.

Five-branch soundness. Let α_{step} be the fraction of corrupted steps in a transcript that is not in $\mathcal{R}_{\text{train}}$, α_{layer} the fraction of corrupted audited layers inside a corrupted sampled step, and α_{entry} the fraction of corrupted output entries inside an audited operation. Let $k_{\text{step}}, k_{\text{layer}}, k_{\text{entry}}$ be the corresponding query budgets, and let $N_{\text{gemm,layer}}$ denote the number of audited GEMM-like operations per layer. For the current spot-check protocol,

$$\begin{aligned}\varepsilon_{\text{total}}^{\text{spot}} \le\ & \underbrace{T \cdot \varepsilon_{\text{bind}}}_{\textbf{M1}\text{: commitment binding}} + \underbrace{k_{\text{step}}\big((1-\alpha_{\text{layer}})^{k_{\text{layer}}} + k_{\text{layer}} N_{\text{gemm,layer}} (1-\alpha_{\text{entry}})^{k_{\text{entry}}}\big)}_{\textbf{M2}\text{: intra-step auditing}} \\ & + \underbrace{(1-\alpha_{\text{step}})^{k_{\text{step}}}}_{\textbf{M3}\text{: step sampling}} + \underbrace{(1+k_{\text{step}})\varepsilon_{\text{zkVM}}}_{\textbf{M4}\text{: proof soundness}} + \underbrace{\varepsilon_{\text{com,pred}} + \varepsilon_{\text{com,bind}}}_{\textbf{M5}\text{: commit-then-reveal}} \\ & + \varepsilon_{\text{PRF}} + \varepsilon_{\text{sig}}.\end{aligned}$$

The five protocol mechanisms are M1–M5. The PRF and signature terms are auxiliary cryptographic correction terms: $\varepsilon_{\mathrm{PRF}}$ accounts for replacing the domain-separated sampler outputs by independent uniform samples, and $\varepsilon_{\mathrm{sig}}$ accounts for forging the verifier's terminal-anchor signature.

Proof sketch. First condition on the commit-then-reveal event: the seed commitment hides σ from the prover until the stream is frozen and binds the verifier to the opened seed. The complement of this event contributes Sound-M5, $\varepsilon_{\mathrm{com,pred}} + \varepsilon_{\mathrm{com,bind}}$. Under this event and the PRF hybrid, S, layers, and entries are uniform samples from their domains. If the transcript is not in $\mathcal{R}_{\mathrm{train}}$, then one of the following must occur. The prover equivocates on a committed tensor or anchor value, giving M1. The sampled steps miss all corrupted steps, giving M3; the displayed binomial term is a convenient upper bound on the exact hypergeometric miss probability. If a corrupted step is sampled, the sampled layers or entries miss the corrupted local operation values, giving the two M2 branches. If the verifier accepts despite a locally false zkVM statement, the proof-system soundness event M4 occurs. Finally, if the terminal chain can be changed after σ is known, the verifier signature has been forged or the protocol flow has been violated; the cryptographic part is $\varepsilon_{\mathrm{sig}}$. A union bound over these events gives the display.

At the paper's working point $k_{\mathrm{step}} = 10^3$ and $\alpha_{\mathrm{step}} = 10^{-2}$, the dominant term is still

$$(1 - \alpha_{\mathrm{step}})^{k_{\mathrm{step}}} \approx 4.3 \times 10^{-5}.$$

The current mainline is therefore sparse-auditing, detection-grade execution verification rather than a clean full-stack 2^{-128} claim.

Knowledge soundness. Assume the inner zkVM proof system is a proof of knowledge in the programmable random-oracle model, with the BCS-/FRI-style extractor accounting just noted, and assume the commitment openings recorded in the public transcript are extractable except with their binding losses. Then an accepting prover for the spot-check protocol can be converted into an extractor that outputs a witness w_{spot} for $\mathcal{R}_{\mathrm{spot}}^{S,Q}$, except with the spot-check soundness error above and the inner-proof and outer-commitment extraction losses. When the sampling events do not miss the actual deviations, this extracted sampled witness is consistent with the universal witness required for $\mathcal{R}_{\mathrm{train}}$.

Proof sketch. Run the proof-of-knowledge extractor on the accepted genesis proof and each accepted audited-step proof. Oracle recording fixes the Fiat–Shamir challenges used by the proofs, and commitment binding fixes the Merkle openings and anchor-chain values to unique tensors except with the stated binding probabilities. The extracted local witnesses agree on adjacent committed roots by the boundary checks and agree with the frozen transcript by the terminal signature. This stitches the extracted local executions into a sampled trace witness. The step from sampled trace to universal trace is exactly where M2 and M3 enter; without IVC or a stronger per-step verifier, knowledge soundness remains a qualified sampled-argument statement. This appendix does not claim a referee-grade deployed-stack extractor theorem with a concrete 2^{-128} end-to-end loss.

Sequential auxiliary-input CZK. For any non-uniform polynomial-time verifier with auxiliary input z, the intended privacy target is computational zero knowledge for the proof objects viewed sequentially within one audit session. The public leakage is the public transcript itself: R, θ, training length, commitment roots, TAP tags, proof sizes, sampled positions after reveal, and the accept/reject outcome. Under the usual honest-verifier-ZK predecessor plus Fiat–Shamir-preserves-ZK package, together with the keyed TAP channel and a CZK outer wrapper when used [16], the verifier learns no additional witness information from the proof transcripts in this sequential/per-proof model.

Proof sketch. The simulator samples the public coins and programmed random-oracle answers needed by the Fiat–Shamir proofs, simulates each accepted proof object in sequence, and uses the hiding of the private commitments and the secrecy of the TAP key K to avoid exposing tensors, weights, data, optimizer state, or raw traffic. Sequential composition allows the simulator to carry forward the public transcript state proof by proof. The exact STARK/FRI setting still contains folklore edges in the bounded-query lift and the transplant of auxiliary-input random-oracle tables into the BCS simulator, and the illustrative concrete inner statistical-ZK residue is not a clean 2^{-128} number. Accordingly, this appendix uses only the qualitative sequential/per-proof consequence. It makes no generic concurrent-composition claim and no shared-setup GUC/EUC closure claim.

G.5 Operational caveats and upgrade boundary

Sparse auditing is detection-grade, not universal. Bulk cheating is detected according to the M2/M3 branches above, and the M3 term is structural for a protocol that opens only k_{step} sampled steps from a length-T stream. A single carefully targeted bad step is caught with probability roughly k_{step}/T, not with cryptographic certainty. Adaptive weighting or PAC-style interpretations, when added, remain overlays with extra assumptions rather than replacements for $\mathcal{R}_{\mathrm{train}}$-conformance.

Domain-separated sampling is part of the statement. The step, layer, and entry samplers must be derived from the opened σ by a domain-separated PRF with fixed-length encodings. The tags `SAMP/STEP`, `SAMP/LAYER`, and `SAMP/ENTRY` must not be reused for training RNG, anchor hashing, TAP MACs, or any other protocol role. Without this discipline the M2/M3 probabilities in the theorem are not the probabilities of the implemented verifier.

Batch and RNG derivation are load-bearing. The relation requires the batch B_t and every registered stochastic component to be derived from the committed dataset root, the master RNG seed, and the deterministic transition rules in `arch_spec`. This is what rules out a wrong-batch or RNG-divergence witness. The verifier audit seed σ is separate from the master RNG seed in `arch_spec`; it only drives the sampled set (S, Q) and is opened after the relevant commitment stream has been frozen. If a component's randomness, dynamic state update, dtype conversion, or data-ordering convention is left outside the registered schedule, the appendix does not certify it.

Terminal anchors and rolling storage. Post-training spot checks remain valid only when the prover can open every sampled commitment. Arbitrary whole-run post-training sampling therefore needs full retention, or deterministic replay from durable checkpoints together with committed replay metadata: optimizer state, data-loader state, stochastic-component state, kernel/reduction-tree and precision metadata, checkpoint interval commitments, and the committed seed schedule declared in `arch_spec`. A strictly rolling deployment is therefore a separate windowed variant: each window needs its own challenge schedule, with the audit seed for that window committed before the window is frozen and opened before the required opening material is evicted. If a sampled point cannot be opened, the verifier rejects rather than resampling.

Windowed audit variant. For proof-before-eviction storage, fix public windows $I_j = \{a_j, \ldots, b_j\}$ and require either a per-window audit commitment $c_j = \mathrm{Com}(\sigma_j; \rho_j)$ posted publicly before step a_j, or a pre-registered public-beacon rule fixed before the window starts. At window close, a signed or timestamped freeze certificate binds the interval, previous anchor, frozen window endpoint A_j (for example $A_j = \mathrm{anchor}_{b_j}$), the seed commitment or beacon rule, sampler parameters, and opening deadlines; this freeze must be public before (σ_j, ρ_j) is opened or the beacon output is used. The opened seed or beacon output derives $S_j \subseteq I_j$, and every sampled proof/opening, or exact committed replay producing that opening, must arrive before the window proof deadline and before private material for I_j is evicted. Missing seed openings, missing sampled openings, failed replay, late proofs, or abort after challenge reveal are rejection/non-compliance events, never occasions to resample or shrink S_j. In such a theorem, M3 and M5 are window-indexed; a full statement needs per-window soundness accounting together with an explicit timing/bulletin-board model, and does not inherit the theorem statement or numerical bound of the single terminal-anchor protocol.

TAP key custody is an assumption in the base model. The keyed tag $h_t = F_K(\mathrm{traffic}_t)$ protects against bounded-entropy traffic-guessing by a verifier who knows plausible traffic patterns but not K. In the base model, K lives in the prover/TAP trust base for the run and is hidden from the verifier. This appendix does not close TAP compromise, prover–TAP collusion, key-rotation ambiguity, side channels, or vendor-root questions. TAP-TEE changes the custody and attestation story for K, but it is an upgrade path with its own attestation, side-channel, and manufacturer-trust assumptions, not a mainline theorem here.

Concrete cryptographic residues remain. The knowledge-soundness and CZK paragraphs are intentionally scoped. The inner BCS/FRI knowledge extractor keeps the quadratic random-oracle term; the outer commitment-layer extraction story is separate. The sequential CZK story is theorem-composed only after keeping the bounded-query and auxiliary-input simulator-transplant edges

explicit. The concrete ZK residue at illustrative parameters blocks any clean deployed-stack 2^{-128} slogan until the parameter story is tightened.

V8, TAP-TEE, and V5-IVC target different gaps. Exact V8 targets genesis-weight provenance, but a paper-ready version still needs public initialization metadata, registry-grade initializer semantics, canonical serialization, dtype/cast rules, and normal-family sampler specifications. TAP-TEE targets telemetry custody and provenance, not the dominant statistical M3 branch. V5-IVC targets step coverage and can remove M3 only by replacing sparse spot-checking with universal per-step proof coverage; it then inherits recursive-verifier, outer-PCS, binding, and extraction costs. None of these upgrades is treated here as deployed-stack ready, and a Groth16 outer wrapper does not retroactively tighten the inner IVC or sampled-verifier soundness terms.

Composition boundary. This appendix makes no generic concurrent-composition, shared-random-oracle, shared-CRS, GUC, or EUC closure claim. The safe paper posture is the sequential/per-proof one stated above, with stronger shared-setup composability left as future work.

G.6 PAC-style Threat model and formalisation.

Generalizing the formalisation of Section G.2, we formalize AI verification beyond execution relation and encompass PAC-style learnability. Then we state precisely under this formalisation the relations we want to verify. All our modelisation and formalisation is done with respect to the framework of *computational complexity*. We recall that, because the architecture in Items **MOD. 2** to **MOD. 4** ultimately verifies faithful execution of a committed training procedure rather than low loss on an external distribution, Section G.2 states the protocol-aligned execution-verification relation together with its soundness, knowledge-soundness, and zero-knowledge guarantees.

G.6.1 Definitions

In this part, we take a *computational complexity theoretic* approach to the problem of AI training verification. This is inspired by recent works in among the learning theory community on the subject of verification, e.g., see [15] for a foundation of *PAC verification* where a *verifier* verifies if a learning algorithm as a *prover* has produced a near-optimal hypothesis[17], or see [2] for a foundation of *self-proving models* in which learning algorithms as provers sucessfully convince a verifier on their answers' correctness[18]. These existing works draw inspiration in their theoretical modeling from a computational complexity perspective. In the seminal work of [15] it is stated that *"The P vs. NP problem asks whether finding a solution ourselves is harder than verifying a solution supplied by someone else. It is natural to ask a similar question in learning theory: Are there machine learning problems for which learning a good hypothesis is harder than verifying one proposed by someone else? We find this theoretical motivation compelling in and of itself."*

We propose a similar conceptual approach in our formalisation, i.e., basing on computational complexity, for a similar perspective: because it is widely believed to be *hard* to train an AI model for some specific task so that it can we *safely* deployed at a national/international level, is there a difficulty gap between such training process and verifying one proposed by some third-party? And it is computational complexity that provides us with the better lens to look into these questions of hardness and verification. The PAC-style definitions below are a learning-theoretic motivation and comparison point. They are not the cryptographic theorem proved by the protocol in this paper: Section G states the narrower execution-conformance claim, namely consistency with a declared training-execution relation.

Our setting of training verification. Let $\mathcal{M} \subseteq \{0,1\}^{\mathcal{C}}$ be a set of models, as a subset of some family of Boolean circuits $\mathcal{C} \stackrel{def}{=} \{C_{n,d}\}_{n,d\in\mathbb{N}}$ indexed by size n and depth d. Let $\mathcal{T}$ be a distribution over $\mathcal{C} \times \{0,1\}$. This represents the desirable results of the training process, the circuits $C_{n,m}$ such that $\mathcal{T}(C_{n,m}) = 1$ is what obtained after training. The prover P and verifier V have access to their own *oracles* $\mathcal{O}_P$ and $\mathcal{O}_V$ respectively. In our prototype later, $\mathcal{O}_P$ represents a zkVM, for instance. The prover P and verifier V have access to private source of random coins as well, denoted $r_P \in \{0,1\}^*$ and $r_V \in \{0,1\}^*$ in that order.

[17] As examples one can think of *machine learning* algorithms for these PAC verification applications.
[18] As examples one can think of *large language models* for these self-proving applications.

A *training verification protocol* can consist of *offline phase* and *online phase*. During the *offline phase*, P publishes some *common reference string* crs. During the *online phase*, P and $V[\mathsf{crs}]$, where $V[\mathsf{crs}]$ indicates that V has access to the published crs, take turn to send each other messages $w_1, w_2, \ldots$, and at the end of the online phase $V[\mathsf{crs}]$ outputs a model $\widetilde{\mathcal{M}} \in \mathcal{M}$ or "reject". One can think of a scenario where P send to $V[\mathsf{crs}]$ a trained model $\widetilde{\mathcal{M}}$ during the online phase and tries to convince $V[\mathsf{crs}]$ of its properties with respect to $\mathcal{T}$. In the end, if $\widetilde{\mathcal{M}}$ is "good", i.e. $\mathcal{T}(\widetilde{\mathcal{M}}) = 1$, $V[\mathsf{crs}]$ outputs $\widetilde{\mathcal{M}}$, otherwise it rejects.

Remark 3. First of all, we treat our models for verification as Boolean circuits, which is nonuniform as later we take into account the possibility of precompiled algorithms with hints as parts of these models. This leads to nonuniformity as different hints on different input lengths produce different behaviors, Our distribution $\mathcal{T}$ encompasses the properties/relations that we would like to check on a given model $C_{n,d}$. This generalizes the notion of relations as one sees usually in the context of NP problems. For example, one can paramterize $\mathcal{T}$ with threshold constraints, and consider the "good" model $C_{n,m}$ that satisfy $\mathcal{T}(C_{n,m}) = 1$.

Our definitions. We start the formal definitions.

Definition 4 (Agnostically trainability)**.** *A class of models $\mathcal{M}$ is* α-agnostically trainable *if there exists a circuit T such that for every distribution $\mathcal{T}$ over $\mathcal{C} \times \{0,1\}$ and every $\epsilon, \delta > 0$, T outputs with probability $1 - \delta$ over its random coins M that satisfies*

$$L_{\mathcal{T}}(M) \leq \alpha \cdot L_{\mathcal{T}}(\mathcal{M}) + \epsilon$$

where $L_{\mathcal{T}} : \{0,1\}^{\mathcal{C}} \to \{0,1\}$ is a loss function *that is defined as $L_{\mathcal{T}}(M) \stackrel{def}{=} \Pr_{(C,y)\leftarrow\mathcal{T}}[M(C) \neq y]$ and $L_{\mathcal{T}}(\mathcal{M}) \stackrel{def}{=} \inf_{M\in\mathcal{M}}(L_{\mathcal{T}}(M))$.*

Definition 4 is inspired by PAC-learnability, that is the circuit T outputs with high probablity some trained model that is ϵ close to the best trained model with respect to the distribution $\mathcal{T}$, while being "agnostic" allows a multiplicative factor α to the best loss with respect to $\mathcal{T}$. The agnostic aspect follows and generalizes existing verification for ML (Definition 4 of [15]) to AI models in general.

We now proceed to define what is *training verifiability*.

Definition 5 (Training verifiability)**.** *We say that $\mathcal{M}$ is* α-verifiable *with respect to a family of distribution $\mathfrak{T}$ containing distributions $\mathcal{T}$ over $\mathcal{C} \times \{0,1\}$ using oracles $\mathcal{O}_P, \mathcal{O}_V$ if there exists PPT Turing machines (P, V) such that for all $\delta, \epsilon > 0$*

Completeness *For any $\mathcal{T} \in \mathfrak{T}$, the random variable $m \stackrel{def}{=} [P^{\mathcal{O}_P}(\delta,\epsilon), V^{\mathcal{O}_V}(\delta,\epsilon)]$ satisfies*

$$\Pr[m \neq \text{ "reject"} \wedge L_{\mathcal{T}}(M) \leq \alpha \cdot L_{\mathcal{T}}(\mathcal{M}) + \epsilon] \geq 1 - \delta \ .$$

Soundness *For any $\mathcal{T} \in \mathfrak{T}$, for any (possibly unbounded) PPT P', the random variable $m \stackrel{def}{=} [P'^{\mathcal{O}_P}(\delta,\epsilon), V^{\mathcal{O}_V}(\delta,\epsilon)]$ satisfies*

$$\Pr[m \neq \text{ "reject"} \wedge L_{\mathcal{T}}(M) > \alpha \cdot L_{\mathcal{T}}(\mathcal{M}) + \epsilon] \leq \delta \ .$$

In the above the random variable $m \stackrel{def}{=} [P^{\mathcal{O}_P}(\delta,\epsilon), V^{\mathcal{O}_V}(\delta,\epsilon)]$ denotes the output of the verifier after the completion of the protocol.

Our models capture prior works. We argue the robustness of our modelisation below. In other words, our setting with respect to Definitions 4 and 5 captures the PAC-verification [15] and self-proving [2] settings. This essentially puts more interests in our new proposed modelisation, as it can englobe existing verification setting.

H Glossary

H.1 Hardware and distributed training

cuBLAS NVIDIA's GPU-accelerated linear algebra library, used here for GEMM-heavy training kernels.

cuDNN	NVIDIA's deep learning primitives library, providing optimised GPU implementations of neural-network operations such as convolutions, normalization, and pooling.
NCCL	NVIDIA Collective Communications Library, the standard library used for multi-GPU and multi-node collectives such as allreduce, allgather, and reduce-scatter.
NVLink	High-bandwidth GPU-to-GPU interconnect used for in-node transfers.
NVSwitch	In-node switching fabric that connects GPUs over NVLink and supports high-bandwidth any-to-any communication.
NVLS	NVLink SHARP, an NCCL path that can offload reductions to NVSwitch hardware and impose a fixed reduction order for supported collectives.
Allreduce	Collective operation that combines tensors across participants and returns the combined tensor to each participant. Floating-point allreduce can be order-sensitive.
Allgather	Collective operation that gathers tensor shards from all participants so that each participant receives the concatenated result.
Reduce-scatter	Collective operation that reduces tensors across participants and scatters disjoint output shards back to them.
All-to-all	Collective communication pattern in which each participant sends a distinct shard to every other participant.
FSDP	Fully Sharded Data Parallelism: a data-parallel training scheme that shards model state across devices and uses collectives to materialize and reduce shards.
Tensor parallelism	Parallelism that splits individual tensor operations, typically matrix multiplications, across devices and often requires allreduce within a layer.
Pipeline parallelism	Parallelism that assigns different layers or layer blocks to different devices and streams microbatches through the resulting pipeline.
Expert parallelism	Parallelism for mixture-of-experts models in which experts are distributed across devices and token routing induces all-to-all traffic.
HBM	High-Bandwidth Memory: stacked DRAM packaged on the GPU module, providing the bulk of training-tensor storage and the highest-bandwidth path between compute and memory.
SmartNIC	Programmable network interface card with on-card compute that can perform packet inspection, telemetry, hashing, or attestation in line with traffic, used here as a tier of network anchoring.
DPU	Data Processing Unit: a programmable accelerator (often combined with a SmartNIC) that offloads networking, storage, and security functions from the host CPU.
TEE	Trusted Execution Environment: an isolated processor mode (e.g. Intel TDX, NVIDIA Confidential Computing) providing remote attestation and integrity guarantees for code and data executed inside it.

H.2 ZK and proof systems

PAC	Probably approximately correct: a learning-theoretic guarantee over a distribution. In this paper PAC-style claims are separate from the cryptographic execution-conformance statement.
NP relation	Polynomial-time decidable relation between a public instance and a private witness; the proof system argues that such a witness exists.
ZK	Zero knowledge: a privacy property saying that a proof reveals no witness information beyond the public statement and accepted transcript.

zkVM Zero-knowledge virtual machine: a proof environment that proves correct execution of a guest program. In this paper the guest is the proof checker, not the full training run.

SNARK Succinct non-interactive argument of knowledge: a proof system with short proofs and efficient verification for NP statements, under system-specific assumptions and setup.

STARK Scalable transparent argument of knowledge, typically a hash-based proof system using polynomial commitment and low-degree testing machinery such as FRI.

FRI Fast Reed–Solomon interactive oracle proof of proximity, the low-degree testing component used in many STARK constructions.

BCS transform Ben-Sasson–Chiesa–Spooner compilation pattern that turns an interactive oracle proof into a non-interactive argument using commitments and Fiat–Shamir challenges.

PCS Polynomial commitment scheme: a commitment primitive for polynomials with succinct opening proofs at selected points.

Groth16 Pairing-based SNARK with very small proofs and fast verification, requiring a structured common reference string for the proved circuit.

Fiat–Shamir Transformation that makes a public-coin interactive proof non-interactive by deriving verifier challenges from a hash of the transcript.

Random oracle Idealized hash-function model in which each new query receives an independent random response, with repeated queries answered consistently.

Knowledge soundness Property that an accepting prover can be converted, except with stated losses, into an extractor that outputs a witness for the accepted statement.

Computational zero knowledge (CZK) Privacy property requiring that proof transcripts can be efficiently simulated up to computational indistinguishability, revealing no witness information beyond the public transcript.

Auxiliary input Extra side information supplied to an adversarial verifier or distinguisher; auxiliary-input security asks that the guarantee survive such side information.

IVC Incrementally verifiable computation: recursive proof composition for a long computation, maintaining a compact proof as steps are appended.

CRS Common reference string: public proof-system parameters, sometimes generated by a trusted or structured setup depending on the proof system.

Public instance Public statement and inputs checked by the verifier, such as commitment roots, tags, parameters, and proof-system public inputs.

Witness Private data that makes the public instance true; here it may include tensors, weights, batches, randomness, optimizer state, and traffic.

Precompile Specialized proof-system circuit or primitive for a fixed operation, such as a MAC chain or Merkle path check, called by the proof checker instead of emulating the operation generically.

Extractor Algorithm used in a knowledge-soundness argument to recover a witness from a prover that convinces the verifier.

GUC/EUC Generalized or externalized universal-composability frameworks for protocols with shared setup or global state. The appendix explicitly does not claim these composition guarantees.

H.3 Protocol and commitments

TAP Traffic Access Point: a passive observation point on training-cluster links. In the base protocol it records or tags traffic rather than deciding whether the run is valid.

TAP tag	Keyed digest of observed traffic used as a public binding value for later proof checks.
TAP-TEE	Upgrade path in which TAP key custody or telemetry attestation is moved into a trusted execution environment (TEE) or similar hardware root. It changes trust assumptions rather than the sampled-auditing theorem.
PRF	Pseudorandom function: a deterministic keyed function whose outputs are computationally indistinguishable from random to parties without the key.
Keyed PRF	A PRF evaluated under a secret key K, used here for domain-separated tags or sampler outputs depending on the protocol role.
Commit-then-reveal	Pattern in which a value, such as a sampling seed, is first committed and only later opened after the relevant transcript has been fixed.
Audit seed	Verifier or public randomness used to derive sampled steps, layers, or entries after the relevant commitment stream has been frozen. It is distinct from the training RNG seed.
Merkle root	Root hash of a Merkle tree, serving as a compact public digest of many leaves.
Merkle commitment	Commitment to a vector or tensor represented by a Merkle tree; individual entries can be opened with authentication paths to the public root.
Anchor chain	Hash chain linking per-step commitments, TAP tags, and run metadata in order, so later entries bind to the earlier transcript.
Terminal anchor	Final value of the anchor chain, used as the point at which the streamed transcript is frozen before challenge material is opened.
Windowed audit	Rolling-storage variant that partitions the run into public windows, freezes each window before revealing its audit seed, and requires sampled openings before private material is evicted.
Window root	Commitment root, digest, or anchor for the public step records in one window. It freezes the public record stream, not the private witness material.
Freeze certificate	Signed or timestamped public message binding a window interval, previous anchor, current window root or anchor, seed commitment, sampler parameters, deadlines, and protocol context before seed reveal.
Bulletin board	Append-only public log or ordering service for protocol events such as seed commitments, freeze certificates, seed openings, proof submissions, aborts, and timeouts.
Logical clock	Monotone counter or timestamp used by a bulletin board to compare event order. It is a public ordering abstraction, not a claim about synchronized hardware clocks.
Public beacon	External randomness source whose output is unpredictable before a specified time and publicly verifiable after release; it can replace a hidden verifier seed in some windowed variants.
Quorum	Threshold set of verifiers or authorities whose joint action authorizes a freeze, randomness contribution, or audit decision under an explicitly stated threshold policy.
Domain separation	Use of distinct labels or encodings so that hashes or PRF outputs for one protocol role cannot be reused as another role.
Poseidon	Hash function designed to be efficient inside algebraic proof systems; used here for Merkle-path checks in the proof circuit.
Spot-checking	Sampled verification of selected steps, layers, or entries rather than exhaustive checking; soundness then includes the probability that sampling misses a deviation.
M1–M5 branches	Shorthand for the five soundness mechanisms tracked in the appendix: commitment binding, intra-step auditing, step sampling, proof soundness, and commit-then-reveal.

Opening material Private data needed to open and prove a sampled statement, including committed values, Merkle paths, commitment randomness when relevant, hints, stochastic state, and proof witnesses.

Full retention Storage mode in which private opening material for every challengeable step is kept until the audit seed has been opened and all sampled proofs have been accepted or rejected.

Deterministic replay Reconstruction of sampled opening material from committed checkpoints and metadata as a single-valued, bit-exact function of the frozen transcript. Failed replay is a rejection event, not a reason to resample.

Checkpoint Durable committed snapshot used for recovery or replay; for training verification it must include enough model, optimizer, RNG, data-loader, kernel, and metadata state to reproduce relevant commitments.

Proof-before-eviction Rolling-window discipline in which a window is frozen, sampled, and proved before the prover deletes the private material needed to answer challenges about that window.

Acceptance watermark Operational status saying that a window's required sampled proofs have accepted, or the window has publicly failed; eviction policies should be keyed to this status rather than to age alone.

V5-IVC Roadmap shorthand for an IVC-based upgrade path that aims at universal per-step coverage rather than sampled step coverage. It is referenced as a target, not as a deployed theorem here.

V8 Roadmap shorthand for an upgrade path targeting genesis-weight provenance and initialization semantics. It is outside the appendix's main theorem.

H.4 ML training and numerical terms

`arch_spec` Architecture specification consumed by the proof checker, declaring model operations, precision, communication pattern, data-loading parameters, and commitment boundaries.

BF16 Bfloat16 floating-point format with a compact mantissa and an FP32-sized exponent range; training commonly uses BF16 values with FP32 accumulation.

FP32 IEEE 754 single-precision floating point, used here for accumulators and selected nonlinear operations.

RNE Round-to-nearest-even, the IEEE 754 rounding mode assumed by the numerical checks unless otherwise stated.

GEMM General matrix-matrix multiplication, the dense linear-algebra primitive underlying most Transformer linear layers and their gradients.

MAC chain Ordered multiply-accumulate computation for a dot product, including the rounding behavior of each multiplication and accumulation step.

Training RNG seed Random-number-generator seed, usually committed through `arch_spec`, that determines batches and stochastic training components. It is separate from the verifier's audit seed.

Stochastic component Declared source of training randomness or stateful stochastic behaviour, such as dropout, data augmentation, stochastic rounding, dynamic loss scaling, or router noise.

Hint Private value supplied by the host to the zkVM guest; it is not trusted until checked against commitments and operation-specific constraints.

Host The trainer-side process or GPU environment that performs the native computation and supplies private hints to the proof checker.

Guest The program executed inside the zkVM, here the proof checker that validates committed hints and sampled operations.

Fused operation	GPU kernel or operation block whose internal intermediates are not separately materialized; the proof treats it through declared inputs, outputs, and operation semantics.