TRAINING WITNESSES: TRUSTING THE TRAINING WITHOUT TRUSTING THE TRAINER

Anonymous authors
Paper under double-blind review

ABSTRACT

Progress in machine learning cannot outpace our ability to verify it. With an explosion in papers today, every scientific claim rests initially on trust in the trainer, leading to uneven evaluation, baselines, and forestalling of reliable progress. Traditionally, the burden of verification falls on the reader, who must reproduce expensive training runs. This strategy is impractical due to an explosion in slop contributions, diversity of methods, and the sheer compute required. We put the burden of proof where it belongs, on the trainer, and in the process also cut the overall cost of verification significantly. We introduce **Witnesses**, a method for certifying training, data usage and evaluation in a neural network training run. Our key insight is that fast behavioral fingerprints with occasional replay challenges are sufficient for auditing neural network training. Our method is applicable at scale with minimal overhead to the trainer, is cheap for the verifier, rejects bad training runs with amplifiable probability, and allows for exact queries of both data inclusion and exclusion. We test our method on language model training runs from 100M to 2B scales, across DDP and FSDP, and demonstrate this minimal overhead. We also introduce a self-regulating leaderboard of "auto-certified" training runs that enables shared baselines and progress. We invite the community to participate in the leaderboard to improve reproducibility in machine learning.

1 INTRODUCTION

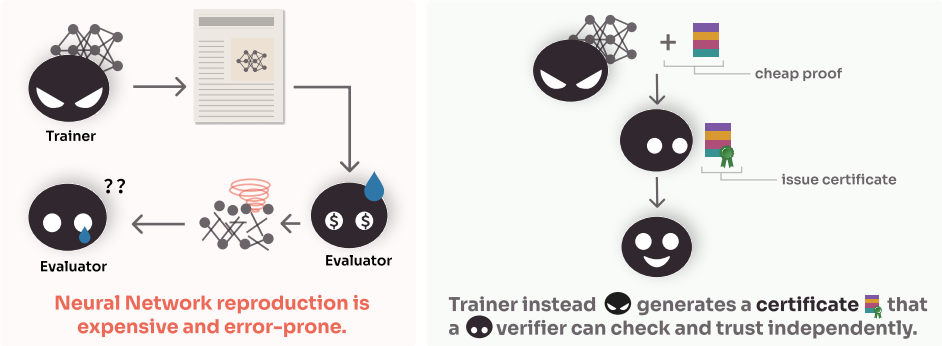


Figure 1: **Training Witnesses.** We present a system that creates privacy-preserving *witnesses* that allows a public verifier to cheaply check and trust what information is included in the training of a model without disclosing model training details, enabling trust in the artifacts without trusting the trainer.

An enormous number of papers (Su et al., 2026) now flood machine learning conferences each year, and the number is growing exponentially, including received 60,000 submissions just at ICLR this year. How much of it has moved the field forward?

There's an emerging consensus (Shah, 2026; Baumann et al., 2026; Abid, 2026; Raff et al., 2025; Schaeffer et al., 2025; Rogers & Luccioni, 2024) in the field that finds it hard to *trust* the papers being written, *reproduce* the results when trust is missing, or even *verify* specific claims made in academic papers. We believe this *progress crisis* (Hutson, 2018) in machine learning is driven by the proliferation of varied baselines, unequal comparisons, and undeclared interventions, rendering standardized comparisons difficult (Sculley et al., 2025) and thus actual progress less visible. We

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

believe a mere policy change alone cannot fix this alarming crisis. Instead, solving this crisis requires a change in the technology we use to evaluate and certify contributions.

Our goal is to mitigate this crisis by *building an algorithmic solution that provides assurance of model developer claims while reducing the cost of verification*. The hardest version of this problem involves certifying entire pretraining runs, which enables standardized comparisons with third-party verification of training without the need to retrain. Reproducibility certificates already exist in the systems research community (USENIX Association, 2026). However, unlike systems, reproduction cannot merely be the responsibility of the program committee since resources required to verify a single ML project are prohibitively higher, let alone a whole batch of projects (ACL Rolling Review, 2026).

In response, **we introduce an algorithm, a code-base agnostic system, and a self-regulating leaderboard** that solves exactly this hardest setting. We let authors prove their models were trained on the data they state, and in turn enable more confidence that their claims of generalization are indeed from algorithmic contributions. Existing methods to do this move the burden of checking onto a single third-party verifier, which is intractable at current scales especially for research areas such as pretraining, novel architecture, etc. Instead, we leverage the insight that **a progress certificate enables systematic comparison of any transitions that happened during computation**, including results; our system therefore enables the model developer to publish exactly this evidence of training progress, which can then be selectively revealed to a lower-powered verifier to certify. Our proof is privacy-preserving, meaning the model training technique will not be revealed to the user; undeclared data usage can also be kept private, other than revealing that additional undeclared data was used.

Our method, **Witnesses**, uses the fact that practical data attestation audits can be established cheaply at scale with a fast behavioral fingerprint, allowing secure rare replay challenges. This insight allows us to build a **protocol and system that reduce scalable training data attestations to be cheap at scale** incurring only $\leq 1.19\times$ speed and $\leq 1.11\times$ memory overhead for a 2.02B parameter, 2048 block size FSDP/TP-2 reference training run. We use three insights to achieve this:

1. **For model parameter provenance**, we introduce and prove that a low-rank, full-spark, orthonormal projection of weight matrices yields a behavioral fingerprint that fixes parameter degrees of freedom. This projection is fast enough to run at every gradient update, and it forces any cheating signal to be non-sparse and consistently detectable.
2. **For update function sealing**, we leverage the insight that both Jax and Torch's accelerator serializations are stateless (JAX Authors, 2026; PyTorch/XLA Authors, 2026; OpenXLA Project, 2026) to ensure all training steps can be exactly replayed without side effects.
3. **For efficient implementation**, we combine Deepspeed-style optimizer state offloading (Ren et al., 2021) with a white-box cryptographic sidecar (Chow et al., 2003) to build a signed, append-only chain, and produce a cheaply-checkable certificate of training data.

We validate this system between 100M-2B scales, including for sharded and multi-host training, as well as test the method under adversarial optimization. We additionally release **a web system**[1] that developers can use to attest their training and build flexible leaderboards to display attestations. We hope these techniques and tools can improve the reproducibility crisis in machine learning.

2 Related Work

Exact certification of training, data and evaluation is conceptually simple, but practically intractable at scale when done naively. At a high level, there are three overarching goals that such a system has to achieve for widespread deployment: the system has to be cheap to execute, trustworthy despite the low overhead, and meaningfully shift cost away from the verifier (lest it becomes just reproduction).

A first method for certifying proof of training data (PoTD) sequentially attests every training transition by building trust using Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) techniques (Sun et al., 2025; Eisenhofer et al., 2025; Garg et al., 2023). These methods are usually too slow to be practical (with Sun et al. (2025) offering "less than a second per update for 10M parameters" performance). Thus, they do not achieve our goal of running with low overhead. Second, statistical characterization methods exist to remove checkpointing cost (Choi et al., 2023),

[1] An anonymized version with user-submitted data is available at https://www.dvbench.org/

leveraging the fact that LLMs temporarily overfit to one sample of training and learn the rough training order of data (Kuditipudi et al., 2025; Maini et al., 2021; Choi et al., 2023). However, a valid sequence of overfitting to existing data does not preclude additional, undeclared poisoning, leading to low detection rates for data addition. This effect fails our second goal of adding trust.

Thus, more practical PoTD methods often simply write down the training with probabilistic chance. However, such systems still require (Madrigal-Cianci & Chordia, 2026) an expensive checkpointing and replay procedure, incurring high cost on both the trainer and verifier as shown in Figure 4.

Importantly, all approaches are specialized to particular training topologies or algorithms (Garg et al., 2023; Eisenhofer et al., 2025; Sun et al., 2025; Maini et al., 2021; Choi et al., 2023), and therefore are unwieldy and do not by themselves generalize across the diverse methods used in LLM development.

3 Setup: Training Witnesses

	Open:	Training	Weight	API
Discloses	Data (training)	✓		
	Data (evaluation)	✓	✓	
	Training procedure	✓		
	Hyperparameters	✓		
	Model params	✓	✓	
	Architecture	✓	✓	
	Inference access	✓	✓	✓
Check	d_{train}	✓		
	$d_{\text{eval}} \notin d_{\text{train}}$	✓	✓	
		Witnesses (ours) can certify		

Figure 2: **Disclosure settings and what is certified.** Our method can protect against dataset inclusion and exclusion attacks in the fully open setting, and data inclusion attacks in the open-weight setting.

We want a system that can give a verifier confidence that a model was trained by some input stream with no undeclared updates. To do this, we propose a **training witness certificate**, which is a cryptographically secure, append-only records chain that establishes facts about a training run. The trainer produces a witness so that any verifier can check for any data sample's inclusion and exclusion. We consider two settings in fig. 2. For *open training*, the verifier has access to model and training data; for *open weight*, the verifier has access to final model only. We support two queries:

Membership Queries Data *membership* queries check if a sample was used in training, defending against *inclusion attacks* where a trainer uses undeclared data (e.g., test-set contamination).

Exclusion Queries Data *exclusion* queries check if a sample was not used in training. These queries defend against *exclusion attacks*, in which a trainer falsely claims to have withheld a sample.

3.1 Defenses and Assumed Capabilities

We set up our task as a three-party game as shown in Figure 3, with the Trainer (*Alice*, A), an untrusted training program; Prover (*Bob*, B), an untrusted prover program collocated with the Trainer; and Verifier (*Victor*, V), a trusted verifier with access to the full initial training state (θ_0, s_0).

Attacker Abilities A is malicious and intends to generate a valid proof tape which contains forged information. We assume that A is a computationally bounded polytime adversary which cannot find collisions in the usual computational security of hash, signature, and commitment schemes. We consider A to have full user-level control over the training process, including the training script, data pipeline, memory, and storage. We assume no boundary between A and B, allowing collusion.

Algorithmic Guarantees Our tape must be difficult to fabricate, meaning a valid tape that's been modified should become invalid. We also guarantee perfect completeness (all honest A can produce a valid tape) and probabilistically-checkable soundness (A can cheat with probability that's an epsilon-function of the computation that B intends to spend in constructing the tape).

Practical Guarantees Our formal algorithm shifts the trust of an expensive training procedure to trusting the correctness of cheap hashing steps. To instantiate this trust efficiently, we anchor a white-box crytographic attestor as a part of B. Specifically, we ship an obfuscated virtual-machine with our system which measures the rest of the system and derives cryptographic primitives. We make the engineering assumption that A cannot forge the behavior of this VM, cannot lift the code of the VM, or modify the code of this VM without breaking the integrity of the tape.

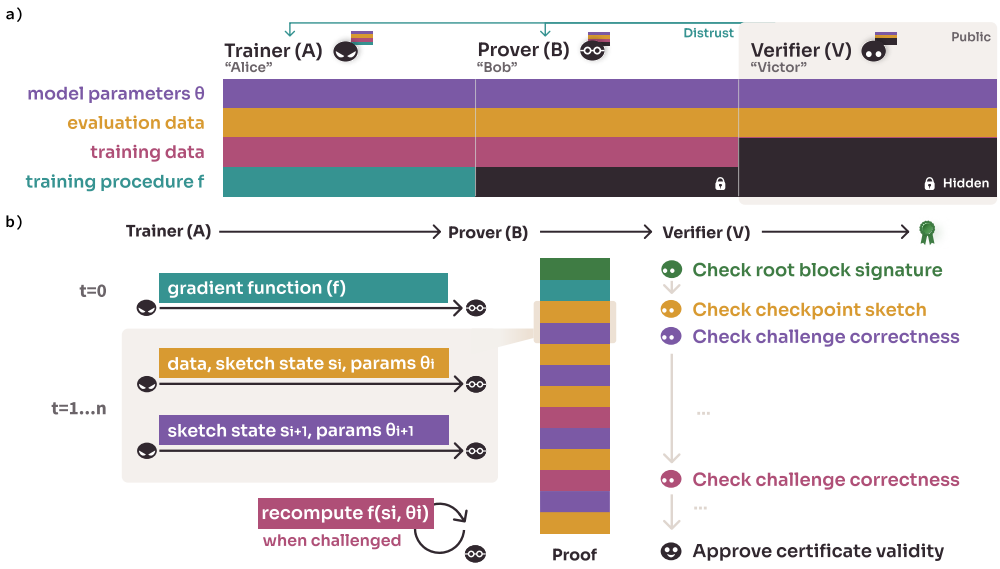


Figure 3: *(Top)* **Privacy and trust boundaries between our three interacting parties.** The prover obtains a black-box functional training procedure, and the verifier obtains only a proof of training data with minimal privacy leakage. *(Bottom)* **Interaction protocol between the parties.** The Witnesses system uses a communicating algorithm between a potentially-malicious trainer and prover to generate a privacy-preserving **public training witness** which any verifier can cheaply check the validity of.

4 Certification Protocol

Tape Structure We define $\bar{\tau} = (R, M, u_0, \ldots, u_{n-1})$. The logical tape consists of a root record R, a measurement record M ($\tau_{-1} := (R, M)$), and contains n transitions $u_0 \ldots u_{n-1}$ within. We describe the construction of the root record in section C.5 and describe the measurement record in section C.6. In order to ensure the integrity of the proof tape, we derive the final tape τ by a cryptographically signed and ratcheted hash-chain. Logically, each link in $\tau_i = f_{\text{chain}_i}(\tau_{i-1}, \bar{\tau}_i)$ contains information about the previous record already inserted, and a ratcheted cryptographic signature. We describe this system in further detail in section C.4. Applying f_{chain} throughout the tape construction gives the final tape $\tau = (\tau_0, \ldots, \tau_{n-1})$.

Construction Responsibilities A is the untrusted trainer; B_{host} is the host program of the untrusted verifier; and B_{att} is the trusted attestation engine which is bootstrapped by B_{host}. There are two ways B_{att} and B_{host} can be linked. For proofs, we assume the two modules are linked through a cryptographically secure attestation method such as a standard zkVM (RISC Zero, 2023); to speed up our practical implementation, we also introduce an engineering tradeoff which uses whitebox-cryptography (Chow et al., 2003) for attestation. Please note, importantly, these choices are consistent with our algorithmic and practical threat models, respectively.

Hashing We use a series of $\mathcal{H}_*$ hash operations throughout which are to be understood as cryptographically ratcheted, stamped hashes. The structure of these hashes is discussed in section C.1.

4.1 Update Records

A logical "update" record consists of a series of records which instrument a particular declared transition. The high-level goal is to record the executed transition function $\tilde{f}(\theta_i, s_i, d_i) = \theta_{i+1}, s_{i+1}$ over parameters θ_i, optimizer states s_i, and input data batch d_i and check it against a declared f.

Update records attest $\tilde{f}$ by committing the output of parameter, state, and data sketch functions C_θ, C_s, C_d, and check that $\tilde{f} \cong f$, the attested "true" update, by probabilistic challenges controlled by p_{check}. Specifically, each update record binds the sketch of the parameter and optimizer state before update $C_\theta(\theta_i), C_s(s_i)$, the update data itself, $C_d(d_i)$, and the sketch of the parameter and optimizer state after update $C_\theta(\theta_{i+1}), C_s(s_{i+1})$. We then sample and replay records with probability p_{check} to check that the transition follows the declared f. This record is described in detail in section C.7.

4.2 State Commitments

So far, our design entails effectively transcribing every training step, attesting the transitions cryptographically, and challenging them on occasion. A naive choice of C_θ, C_s, and C_d would make the transcript prohibitively long, slow to produce and check. We address this with three sketches:

First, we introduce in section 4.2.1 a novel *quasi-geometric* commitment for parameters C_θ which is designed both to attest the identity of the parameters between commitments and to ensure that future verifiers can easily link any checkpoint against a commuted fingerprint geometrically.

Second, we use a combination of a cryptographically secure hash and a Karp-style rolling hash for the data commitment C_d, which ensures both that downstream verifiers can check any fragment of data used in training and bind the data to the tape. We discuss this in section 4.2.2.

Furthermore, we use a *non-geometric* hash for the optimizer state C_s designed to ensure the identity between two distinct commitments of s_i (i.e., the one in τ_i and the one in τ_{i+1}) remains the same. We discuss this standard multi-round permutation non-geometric hash in section C.3.

4.2.1 Geometric Parameter Sketch

Let us first turn to our quasi-geometric parameter commitment C_θ using orthonormal projections.

Algorithm 1: SketchCommit $C_\theta(\theta_i, p_{\text{sketch}})$

Require: Parameters $\theta_i = (\theta_i^{(1)}, \ldots, \theta_i^{(m)})$
Require: Sketch fraction $p_{\text{sketch}} \in (0, 1]$
Require: Fixed orthonormal sketch matrices $\{Q_j\}_{j=1}^m$
Ensure: $C_\theta(\theta_i)$

$C^{(0)} \leftarrow \perp$ ▷ Initial commitment
for $j \leftarrow 1$ **to** m **do**
  $d_j \leftarrow \dim\left(\theta_i^{(j)}\right)$
  $l_j \leftarrow \lceil p_{\text{sketch}} \cdot d_j \rceil$
  $l_j \leftarrow \min\{\max\{l_j, 1\}, d_j\}$
  assert $Q_j \in \mathbb{R}^{d_j \times l_j}$ ▷ Domain is p_{sketch} portion of leaf
  assert $Q_j^T Q_j = I_{l_j}$ ▷ Orthonormality
  $r_i^{(j)} \leftarrow \theta_i^{(j)} Q_j$ ▷ Project leaf
  $r_i^{(j)\prime} \leftarrow \mathcal{H}_{\text{sketch}}\left(i,\, j,\, \text{vec}\left(r_i^{(j)}\right)\right)$ ▷ Hash projection
  $C^{(j)} \leftarrow \mathcal{H}_{\text{sketch}}\left(r_i^{(j)\prime},\, C^{(j-1)}\right)$ ▷ Merkle fold
end for
return $C_\theta(\theta_i) \leftarrow C^{(m)}$

Why Just a Low-Rank Projection We design here a very simple scheme to quickly bind model parameters to tape: just project the weights θ_i against fixed orthonormal vectors. Our key observation, proven in lemma E.9, is that modifications to model parameters which survive a random orthonormal matrix Q_j cannot be sparse because such random projections are almost surely full spark. Thus, an invalid update either has to touch a lot of coordinates (making it more easily detectable by the challenge in section 4.1) or has to show up on this sketch.

High-Level Design The sketch (algorithm 1) involves caching projections of orthonormal vectors Q_j against each weight matrix θ, thus "fixing" degrees of freedom each matrix can express. We then perform a Merkle fold to reduce probes to a single hash.

4.2.2 Batch Commitments

To attest the actual data being used, we also commit hashes of input data d_i during each transition. We use two hashes for this: first, a cryptographically secure hash C_d commits the data to ensure sampling is random with respect to data, and a fast Karp-style rolling hash F_d attests to the downstream verifier what data is actually being used. Details of both hashes are discussed in section C.2.

5 Protocol Guarantees

Here, we sketch out the argument that the protocol given in section 4 admits a probabilistically checkable proof of training data usage. Below we present the proof for both the *soundness* and *completeness* of the protocol. A more detailed proof of soundness can be found in section E. In particular, section E.2 gives the threat model axioms that we rely on.

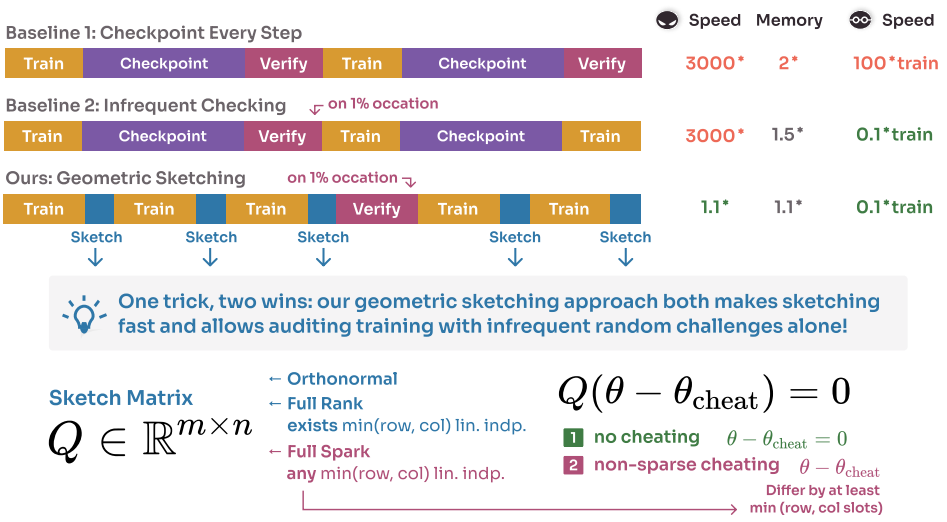


Figure 4: **Our protocol's key security insight**. By using a geometric sketch, we can achieve fast binds of our model parameters to a certificate tape while ensuring that cheating which breaks the continuity of the tape cannot be done sparsely. Our protocol is cheap, fast and has a low memory footprint.

We start by defining $I_f(\tau)$, the number of invalid transitions recorded in τ under f: $I_f(\tau) = |\{i \in \{0, \ldots, n-1\} \mid (\theta_{i+1}, s_{i+1}) \neq f(\theta_i, s_i, d_i)\}|$. $\theta_i, \theta_{i+1}, s_i, s_{i+1}, d_i$ are transitions in τ. We also generate a single-use nonce ν to be used throughout the proof.

5.1 Completeness

We want all valid training programs to produce valid tapes; i.e., we'd like our protocol to be *complete*.

Theorem 5.1 (Completeness of the Protocol)**.**

$$\Pr\left[V(\tau, c, \nu, \theta_0, s_0, \mathcal{H}_{\mathit{func}}(f), \theta_f) = 1 \mid I_f(\tau) = 0, \theta_f \in \tau\right] = 1 \tag{1}$$

Proof. For all θ, s, our sketches are all deterministic functions of their inputs, configuration c, and declared protocol randomness, including the shared boundary family, so $x = x' \implies C_s(x) = C_s(x')$ and $C_\theta(x) = C_\theta(x')$. The verifier-supplied (θ_0, s_0) gives the base case. Thus by induction over the tape updates, we have that $C_s^{\text{post}}(u_i) = C_s^{\text{pre}}(u_{i+1})$ and $C_\theta^{\text{post}}(u_i) = C_\theta^{\text{pre}}(u_{i+1})$ for all $i \in \{0, \ldots, n-2\}$. Thus $V_{\text{bind}}, V_{\text{tape}}$ will not reject. Furthermore, since every recorded transition agrees with f, V_{replay} will not reject. Finally, since $\theta_f \in \tau$, V_{bind} will not reject based on our sketch's completeness. Thus $V(\tau, c, \nu, \theta_0, s_0, \mathcal{H}_{\text{func}}(f), \theta_f) = 1$. □

5.2 Soundness

To ensure that we prevent cheating, we show that the probability of a successful cheat is bounded:

Theorem 5.2 (Soundness of the Protocol)**.**

$$\Pr\left[V(\tau, c, \nu, \theta_0, s_0, \mathcal{H}_{\mathit{func}}(f), \theta_f) = 1 \mid I_f(\tau) \geq k\right] \leq \varepsilon_1(k, c) \tag{2}$$

and

$$\Pr\left[V(\tau, c, \nu, \theta_0, s_0, \mathcal{H}_{\mathit{func}}(f), \theta_f) = 1 \mid I_f(\tau) = 0, \theta_f \notin \tau\right] \leq \varepsilon_2(c) \tag{3}$$

where $\varepsilon_1, \varepsilon_2$ are defined as in theorems E.12 and E.14, and k is the count of invalid updates.

We give formal proofs of this statement in theorems E.12 and E.14. At a high level, we demonstrate this by casework. Specifically, in lemma E.11, we show that there are only three possible ways of sneaking in invalid updates, each of which has bounded probability of success: by swapping out the optimizer state, by swapping out the sketch, or by swapping out both.

6 Efficient Implementation

In addition to the algorithm, we also contribute a low-overhead implementation of the protocol by addressing several challenges of a naive implementation.

Sealing Computation To guarantee that the attested transition function f is of bounded size, we leverage the insight that modern accelerator compilation already requires computation to be stateless and extracts a closed update program. We thus just extract and replay the lowered accelerator compilation result by either the Jax or Torch libraries. We describe this in detail at section F.1.

Sealing B_{att} Other than the whitebox attested binary, many Rust worker operations require dropping back to the Python global interpreter lock to perform an operation (e.g., traverse a PyTree). However, shared Python interpreter state may be mutable by future operations, tampering with the integrity of the attested checker program. To address this, we develop a best-effort sealing mechanism for Python global state through AST monitoring, which we expand on in section F.4.

Worker Concurrency Our Python API runs inline to user code, dispatching steps for training and generating artifacts as required by section 4.1. A Rust worker then separately coordinates the actual cryptography and waits for accelerator work to finish before writing the certificate. To ensure Python and Rust concurrent state transfers are coherent, we design a selective state offloading scheme discussed in section F.2; we further discuss our broker synchronization logic in section F.5.

7 Experiments

Certificate Integrity Under a 100M parameter model, we sweep IID random perturbations to the model parameters, and measure the resulting detection rates.

Training Data Corruption We corrupt our training sequences with IID random perturbations across different rates, and measure the probability of detection over multiple independent samples. We compare the emperical rates against our theory, statistical methods (Choi et al., 2023), and a naive full offload-and-challenge baseline (Madrigal-Cianci & Chordia, 2026). We aim to establish that we can catch any effectful sample corruption (Souly et al., 2025) at robust rates with strong performance.

Sketch Adversarial Optimization Finally, since the sketch challenge vector is visible in the certificate, a malicious attacker may produce an entirely honest training and then adversarially optimize against the sketch. To address this, we take a 100M parameter model θ and choose two sets of 64-token sequences D^{train} and D^{mal}. We use AdamW to minimize:

$$\mathcal{L}(\theta;\lambda) = D_{\text{KL}}^{\text{train}}(\theta_0\|\theta) - \lambda D_{\text{KL}}^{\text{mal}}(\theta_0\|\theta) + \text{RMS}\left(\frac{S(\theta) - S(\theta_0)}{a_{\text{tol}} + r_{\text{tol}}|S(\theta_0)|}\right), \tag{4}$$

where S is the committed parameter sketch, λ is a Pareto tradeoff parameter, $a_{\text{tol}} = 1 \times 10^{-8}$ and $r_{\text{tol}} = 1 \times 10^{-5}$ are calibration values for `float32` precision. In addition to optimizing this loss, we additionally perform the optimization in a subspace orthogonal to the sketch. That is, for Q being the orthonormal basis of the sketch, we update parameters as $\Delta_{\text{attack}} = \Delta_{\text{naive}_{\mathcal{L}}}\left(I - QQ^T\right)$.

Performance Evaluations Our primary performance evaluations involve pretraining a billion-param scale dense GPT model with a weight-tied embedding of 100,288 tokens, a significant increase over published baselines. We profile four regimes: 1) 1.03B parameter GPT model under data-parallel training (DDP) across 2 GPUs; 2) 2.02B parameter GPT model under two-way tensor-parallel training (TP-2) across 2 GPUs; 3) roofline analysis of our system under one GPU, training the same 1.03B parameter GPT model as in the DDP case; 4) two-GPU TP-2 scaling evaluation at various scales ranging from 100M parameters to 2B parameters.

8 Results

Our system achieves strong performance with theoretically-sound detection rates. In fig. 5, we show that our end-to-end system can detect all poisoning-regime data substitution attacks with near-certainty, even while checking 1% of transitions. Our system outperforms Choi et al. (2023) in detection rates, while performing dramatically faster (fig. 7) than offload-and-challenge Madrigal-Cianci & Chordia (2026). Our detection rates match theoretically estimated detection almost exactly.

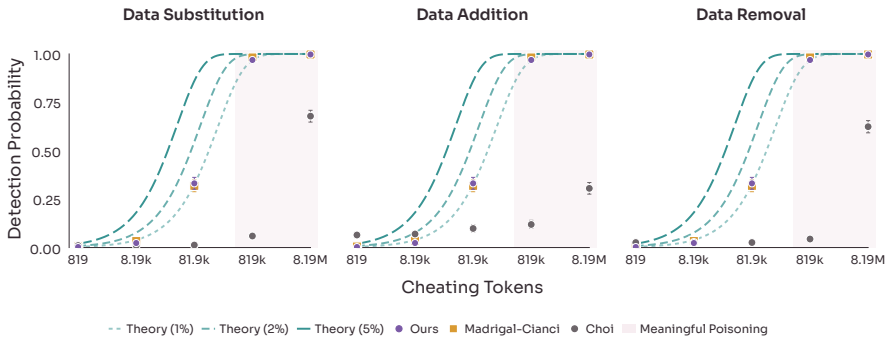


Figure 5: **Our model achieves near-certain detection rates of data inclusion, exclusion, and substitution attacks at any amounts that would constitute meaningful poisoning**. End-to-end data uinsertion attacks, sweeping across number of tokens inserted. Empirical checks performed during 1% of updates, in line with all performance evaluations. *(Left)*: attack by substituting declared and undeclared sequences. *(Middle)*: attack by adding sequences. *(Right)*: attack by subtracting sequences. Attacks distributed uniformly. Shaded region represents literature-established rates of effective poisoning attacks; error bars are 10-90 Beta posterior credible intervals.

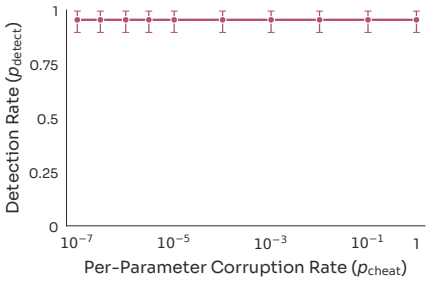


Figure 6: **Swapping out parameters uniformly is detected with near certainty**. Certificate rejection rate as a function of per-parameter invalid transition probability at 100M params. Error bars are 10-90 Beta posterior credible intervals.

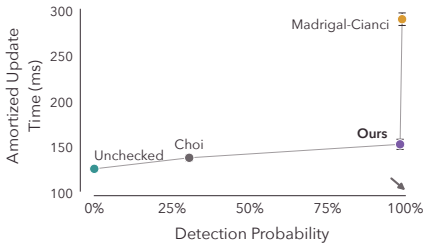


Figure 7: **Our method achieves low overheads versus baselines**. Per-step single GPU training overheads versus detection rates on 100M dense GPT-style model on single-GPU with sizes scaled for roofline saturation. Gray arrow shows direction of improvement. NVIDIA A100 PCIe.

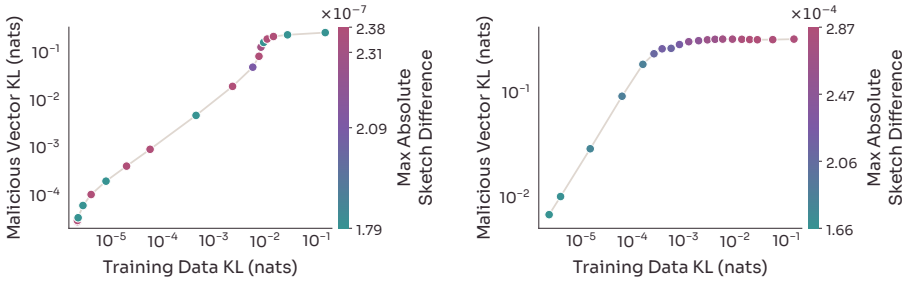


Figure 8: **Even active optimization against our sketch still cannot evade it without meaningfully shifting the model towards training distribution collapse.** We sweep across a Pareto tradeoff parameter λ to minimize eq. (4). **Left**: directly minimizing eq. (4) with sketch-nullspace projection. **Right**: attempting to modify output on malicious KL without sketch error minimization or null space projection. All plotted points detected by sketch at floating point precision $r_{\text{tol}} = 1 \times 10^{-5}$, $a_{\text{tol}} = 1 \times 10^{-8}$.

Unlike prior work, our system reliably detects both inclusion and exclusion attacks. Although Choi et al. (2023) is reasonably fast, a provenance-style statistical check cannot reliably detect data addition attacks fig. 5 (middle). Our system reliably detects both inclusion and exclusion attacks.

Parameter corruption is detected with certainty. As seen in fig. 6, our sketch is able to detect in practice any dependently distributed random parameter corruption with almost certainty. In fact, structured random parameter corruption in select subspaces is also reliably detected fig. 14.

Strong detection rate persists even through adversarial optimization. Even despite performing adversarial optimization against our publicly-disclosed sketch, fig. 8 highlights that our system is still able to detect any meaningful perturbation to the model outputs. Minimizing the sketch

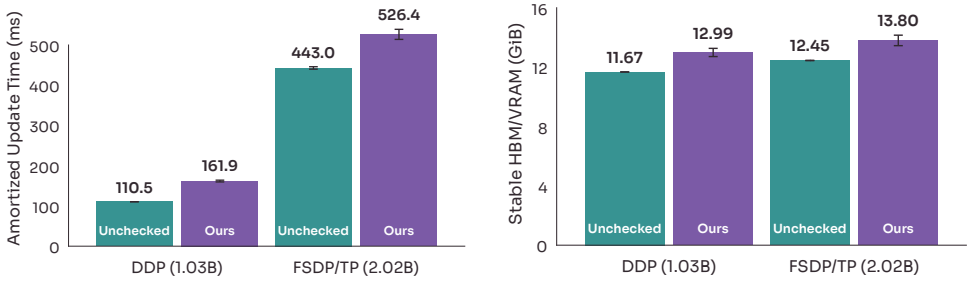


Figure 9: **Our method achieves remarkably low overhead at sharded training settings, where it is communication-bound**. Communication-bound overheads. **Left**: amortized per-step training time; **Right**: mean HBM resident size (bytes in use) per GPU. Error bars from Student's t-test confidence intervals over 100M tokens of training at 1% check probability. DDP and FSDP sharding conducted over 2 NVIDIA H100 NVL72.

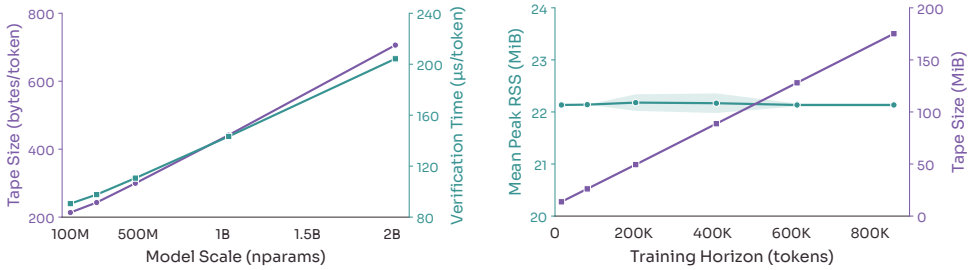


Figure 10: **Our verification system maintains constant memory (RSS) and linear tape size to model size, meaning it can be effectively streamed**. **Left**: Certificate length and verification time per token across a variety of scales. **Right**: verification procedure Resident Set Size (RSS) as a function of cert. length. Verification conducted using `wait4` syscall under a single pinned physical core of an AMD EPYC 9554 processor. Error bars from Student's t-test confidence intervals.

error meaningfully reduces the ability to achieve high KL divergence on malicious samples while maintaining low KL divergence on training samples. Our checkpoint provenance scheme, even despite whitebox access to the sketch, is therefore still secure against trivial offline nullspace attacks.

Our system performs most strongly in sharded, communication-bound settings. Our headline performance results, $\leq 1.19\times$ speed overhead, are achieved in fig. 9, where we are able to elide copying by overlapping them with communication. Saturation experiments in fig. 15 show that we achieve this without sacrificing the MFU or throughput of underlying models.

Verification is streamable with constant working set. One tradeoff of our system is that the certificate grows log-log linearly with model parameters. However, fig. 10 shows that our verification procedure only needs to keep a constant working set of the certificate in memory. A large certificate can thus be effectively verified online in a streaming fashion without ever storing the whole certificate.

9 Discussion

In this work, we introduced Witnesses, a practical system for attesting training progress of large language models which detects data attacks across 100M-2B parameter GPT-style models trained with both DDP and FSDP. In practice, Witnesses is the first attestation system of model training progress demonstrated at billion-scale LLM training with practical overheads, making it immediately applicable at academic and open-weight scales.

Although our current solution extends solely to the process of training, it is general over any closed update function of parameters, and fast enough to deploy at scale. Future work can extend this to support additional forms of (arguably less latency sensitive) workflows. Thus, we believe Witnesses opens an exciting new field of work that provides technical governance and integrity to scientific results previously not possible at modern ML-scales. To encourage the adoption, we release Witnesses as a leaderboard, described in section B, allowing submissions of training runs to be attested and verified by the community. We hope that these tools will be able to meaningfully make model training more reproducible, leading to consistent progress in our exponentially growing field.

AI USE STATEMENT

We use LLM assistance for grammatical, formatting, and bibliography support. All code, figures, text, and citations have been manually and exhaustively checked. We additionally use coding agents for the implementation of method details, along with assistance for proofreading and editing including for the arguments of proofs. However, all proofs are manually derived and checked. We take full responsibility of the work.

REPRODUCIBILITY STATEMENT

The Witnesses system is available at https://anonymous.4open.science/r/zkdv/. The README file of the repository provides setup instructions, which include a minimal setup step that generates the initial whitebox cryptography randomness core. The package is organized as a `uv` packaged standard Rust/Python library.

REFERENCES

Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. Zero-knowledge proofs of training for deep neural networks. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, pp. 4316–4330. ACM, 2024. doi: 10.1145/3658644.3670316.

Abubakar Abid. What we learned by reproducing 2,200 papers from icml. https://huggingface.co/blog/icml-2026-open-reproductions, August 2026. Hugging Face Blog.

ACL Rolling Review. An explanatory letter on the ARR may 2026 cycle, June 2026. URL https://aclrollingreview.org/may26-cycle-letter.

Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Technical report, National Institute of Standards and Technology, 2010. URL https://www.aumasson.jp/blake/blake.pdf. Submission to the NIST SHA-3 Competition.

Joachim Baumann, Jiaxin Pei, Sanmi Koyejo, and Dirk Hovy. Stop automating peer review without rigorous evaluation. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. doi: 10.48550/arXiv.2605.03202. URL https://arxiv.org/abs/2605.03202. Position paper, oral presentation.

Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pp. 90–99. IEEE Computer Society, 1991. doi: 10.1109/SFCS.1991.185352.

Siyuan Chen, Zhuofeng Wang, Zelong Guan, Yudong Liu, and Phillip B. Gibbons. Practical offloading for fine-tuning LLM on commodity GPU via learned sparse projectors. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 23614–23622, 2025. doi: 10.1609/aaai.v39i22.34531. URL https://ojs.aaai.org/index.php/AAAI/article/view/34531.

Dami Choi, Yonadav Shavit, and David K. Duvenaud. Tools for verifying neural models' training data. In *Advances in Neural Information Processing Systems*, volume 36, pp. 1154–1188. Curran Associates, Inc., 2023. doi: 10.52202/075280-0056. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/03e33e1f62e3302b47fe1d38a235921e-Abstract-Conference.html.

Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pp. 250–270. Springer, 2003. doi: 10.1007/3-540-36492-7_17.

Thorsten Eisenhofer, Doreen Riepel, Varun Chandrasekaran, Esha Ghosh, Olga Ohrimenko, and Nicolas Papernot. Verifiable and provably secure machine unlearning. In *2025 IEEE Conference on Secure and Trustworthy Machine Learning*, pp. 479–496. IEEE, 2025. doi: 10.1109/SaTML64287.2025.00033.

Rūsiņš Freivalds. Fast probabilistic algorithms. In Jiří Bečvář (ed.), *Mathematical Foundations of Computer Science 1979*, volume 74 of *Lecture Notes in Computer Science*, pp. 57–69. Springer, 1979. doi: 10.1007/3-540-09526-8_5.

Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. Experimenting with zero-knowledge proofs of training. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1880–1894. ACM, 2023. doi: 10.1145/3576915.3623202.

Sam Gunn. How to sketch a learning algorithm. *arXiv preprint arXiv:2604.07328*, 2026. doi: 10.48550/arXiv.2604.07328. URL https://arxiv.org/abs/2604.07328.

Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, 2018. doi: 10.1126/science.359.6377.725. URL https://www.science.org/doi/abs/10.1126/science.359.6377.725.

JAX Authors. Exporting and serializing staged-out computations, 2026. URL https://docs.jax.dev/en/latest/export/export.html. Accessed August 27, 2026.

Hengrui Jia, Mohammad Yaghini, Christopher A. Choquette-Choo, Natalie Dullerud, Anvith Thudi, Varun Chandrasekaran, and Nicolas Papernot. Proof-of-learning: Definitions and practice. In *2021 IEEE Symposium on Security and Privacy*, pp. 1039–1056. IEEE, 2021. doi: 10.1109/SP40001.2021.00106.

Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi: 10.1147/rd.312.0249.

Rohith Kuditipudi, Jing Huang, Sally Zhu, Diyi Yang, Chris Potts, and Percy Liang. Blackbox model provenance via palimpsestic membership inference. In *Advances in Neural Information Processing Systems*, volume 38, pp. 162337–162367. Curran Associates, Inc., 2025. doi: 10.52202/085713-5421. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/eda4f6fdf680bcd2b81c969a4090856a-Abstract-Conference.html.

Tingfeng Lan, Yusen Wu, Bin Ma, Zhaoyuan Su, Rui Yang, Tekin Bicer, Masahiro Tanaka, Olatunji Ruwase, Dong Li, and Yue Cheng. ZenFlow: Enabling stall-free offloading training via asynchronous updates. *arXiv preprint arXiv:2505.12242*, 2025. doi: 10.48550/arXiv.2505.12242. URL https://arxiv.org/abs/2505.12242.

Juan P. Madrigal-Cianci and Eshan Chordia. Probabilistic audits for verifiable training and outcome improvement in decentralized learning, 2026. URL https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6552219. SSRN Scholarly Paper No. 6552219.

Pratyush Maini, Mohammad Yaghini, and Nicolas Papernot. Dataset inference: Ownership resolution in machine learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=hvdKKV2yt7T.

OpenXLA Project. StableHLO compatibility, 2026. URL https://openxla.org/stablehlo/compatibility. Accessed August 27, 2026.

Pierre Peigné, Ky Nguyen, and Paul Wang. Zero knowledge verification for frontier AI training is possible. *arXiv preprint arXiv:2606.05433*, 2026. doi: 10.48550/arXiv.2606.05433. URL https://arxiv.org/abs/2606.05433.

PyTorch/XLA Authors. Torch Export to StableHLO, 2026. URL https://docs.pytorch.org/xla/release/r2.6/features/stablehlo.html. Accessed August 27, 2026.

Edward Raff, Michel Benaroch, Sagar Samtani, and Andrew L. Farris. What do machine learning researchers mean by "reproducible"? *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(27):28671–28683, 2025. doi: 10.1609/aaai.v39i27.35093. URL https://doi.org/10.1609/aaai.v39i27.35093.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020. doi: 10.1109/SC41405.2020.00024.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference*, pp. 551–564. USENIX Association, 2021. URL https://www.usenix.org/conference/atc21/presentation/ren-jie.

RISC Zero. RISC Zero zkVM: Scalable, transparent arguments of RISC-V integrity. Technical report, RISC Zero, 2023. URL https://dev.risczero.com/proof-system-in-detail.pdf.

Anna Rogers and Sasha Luccioni. Position: Key claims in LLM research have a long tail of footnotes. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 42647–42665. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/rogers24a.html.

Rylan Schaeffer, Joshua Kazdan, Yegor Denisov-Blanch, Brando Miranda, Matthias Gerstgrasser, Susan Zhang, Andreas Haupt, Isha Gupta, Elyas Obbad, Jesse Dodge, Jessica Zosa Forde, Francesco Orabona, Sanmi Koyejo, and David L. Donoho. Position: Machine learning conferences should establish a "refutations and critiques" track. In *The Thirty-Ninth Annual Conference on Neural Information Processing Systems Position Paper Track*, 2025. URL https://openreview.net/forum?id=DS1XSAPvKs.

D. Sculley, William Cukierski, Phil Culliton, Sohier Dane, Maggie M. Demkin, Ryan Holbrook, Addison Howard, Paul T. Mooney, Walter Reade, Meg Risdal, and Nate Keating. Position: AI competitions provide the gold standard for empirical rigor in GenAI evaluation. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 82166–82178. PMLR, 2025. URL https://proceedings.mlr.press/v267/sculley25a.html.

Nihar B. Shah. Position: Peer review in ML/AI conferences should separate publication from presentation and offer non-anonymous review tracks. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/67082.

Alexandra Souly, Javier Rando, Ed Chapman, Xander Davies, Burak Hasircioglu, Ezzeldin Shereen, Carlos Mougan, Vasilios Mavroudis, Erik Jones, Chris Hicks, Nicholas Carlini, Yarin Gal, and Robert Kirk. Poisoning attacks on LLMs require a near-constant number of poison samples. *arXiv preprint arXiv:2510.07192*, 2025. doi: 10.48550/arXiv.2510.07192. URL https://arxiv.org/abs/2510.07192.

Buxin Su, Natalie Collina, Garrett Wen, Didong Li, Kyunghyun Cho, Jianqing Fan, Bingxin Zhao, and Weijie Su. Self-rankings as a predictor of scientific impact beyond peer review. *Nature Computational Science*, 2026. doi: 10.1038/s43588-026-01039-0. URL https://doi.org/10.1038/s43588-026-01039-0.

Haochen Sun, Tonghe Bai, Jason Li, and Hongyang Zhang. zkDL: Efficient zero-knowledge proofs of deep learning training. *IEEE Transactions on Information Forensics and Security*, 20:914–927, 2025. doi: 10.1109/TIFS.2024.3520863.

USENIX Association. OSDI '26 call for artifacts, 2026. URL https://www.usenix.org/conference/osdi26/call-for-artifacts. Accessed August 27, 2026.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 1113–1120. ACM, 2009. doi: 10.1145/1553374.1553516.

Guang Yang, Ziye Geng, Yihang Chen, and Changqing Luo. Fingerprinting deep neural networks for ownership protection: An analytical approach. In *International Conference on Learning Representations*, pp. 36124–36143, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/3d0b9b7ffa25a8edd4b14a00075a4986-Abstract-Conference.html.

Sally Zhu, Ahmed M. Ahmed, Rohith Kuditipudi, and Percy Liang. Independence tests for language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 79673–79698. PMLR, 2025. URL https://proceedings.mlr.press/v267/zhu25a.html.

Appendix

Table of Contents

A Related Work **15**

B The Witnesses Leaderboard System **15**

B.1 Overall Design 15

B.2 Implementation Details 18

B.3 User Experience. 18

C Protocol Design Details **19**

C.1 Commitment Hashes 19

C.2 Batch Fingerprinting 19

C.3 Non-Geometric Optimizer State Hash 19

C.4 Record Signature Ratcheting 20

C.5 Root Record Structure 20

C.6 Measurement Record Structure 20

C.7 Update Record Structure. 20

D Verification Procedure **21**

E Complete Proof **22**

E.1 Formalism 22

E.2 Threat Model Guarantees 22

E.3 Optimizer State Binding 23

E.4 Parameter Sketching 24

E.5 Tape Integrity 25

E.6 Checkpoint Integrity 26

E.7 Additional Properties 27

F Performant Systems Design **27**

F.1 Update Program Serialization 27

F.2 Interleaved State Offloading and Recovery 28

F.3 Eliding Offload Computation 29

F.4 Invoked Python Sealing 29

F.5 Flow Control and Accelerator Memory Management 30

F.6 Additional Notes on Execution Order and Pipelining 31

G Additional Evaluations **31**

A Related Work

Statistical Training Data Provenance Our work traces its lineage to systems that check what data is being used during training. Usually, these systems can only establish positive inclusion evidence ("is data X used?"). Methods include using memorization as an inclusion heuristic (Choi et al., 2023), post-hoc checkpoint evidence (Maini et al., 2021), using training order and forgetting dynamics (Kuditipudi et al., 2025; Zhu et al., 2025). Our work follows the same goal as proving training data usage, but also introduces the ability to prove negative evidence ("is data X not used?") and provides formally guaranteed properties.

Cryptographic Proof of Training Cryptographic proof of training methods take the opposite tradeoff as statistical ones; instead of checking for evidence of inclusion, these methods certify the process of training using expensive formal proofs. Methods include training computation certificates restricted to tiny architectures (Garg et al., 2023; Eisenhofer et al., 2025; Sun et al., 2025), zero-knowledge proof systems for fully dense networks (Abbaszadeh et al., 2024), and formal sketches of training certification with dedicated hardware (Peigné et al., 2026). We use a probabilistically sampled method, most similar to Madrigal-Cianci & Chordia (2026) but with a fast provenance sketch that dramatically reduces the cost of verification while maintaining amplifiable soundness.

Fast Fingerprints of Training State A key part of our method involves fast per-step checkpoint evidence, which conventionally performs slowly and is the key drawback of methods like Jia et al. (2021). Faster sketch methods include classical rolling hashes (Karp & Rabin, 1987), randomized, feature-preserving dimensionality reduction (Weinberger et al., 2009), and newer decision boundary characterizations such as Yang et al. (2026) or data deletion sketches such as Gunn (2026). Our method uses low-rank orthonormal state projections to cheaply constrain degrees of freedom, leading the sketch to be performant enough to run during every update.

Sampled Probabilistic Replay Algorithms Our method derives its amplifier assurance using a conventional assurance technique of commit and random-check. This technique traces its lineage to classical random verification literature (Freivalds, 1979), which evolved into online partial checking systems for coherence such as (Blum et al., 1991). Jia et al. (2021) has also recently used this technique to verify machine learning training, but its reliance on fully streaming frequent chunks of parameter updates makes it impractical at scale.

Fast Parameter Offloading Finally, to make our system practically feasible, we borrow from the literature of fast parameter streaming. This line of work is most notably shown by the DeepSpeed project in the ZeRO algorithm (Rajbhandari et al., 2020). Specific methods include eliding GPU memory movement during all gathers (Ren et al., 2021), offloading only learned sparse projectors (Chen et al., 2025), and prioritizing asynchrony using gradient information (Lan et al., 2025). Our work also uses elided offloading, but makes the realization that waiting for offloading to finish is only important during sampled replay steps and thus makes no attempts at intermediate coherence.

B The Witnesses Leaderboard System

B.1 Overall Design

Our leaderboard system allows users to stream their training runs to a publicly trusted verifier website, which cheaply verifies the training run and queries any data hashes in the streamed certificate against a centralized repository of corpora. If a given sample is found, the website additionally displays the provenance of the sample, percentages of identified text and the corresponding corpus name.

Measurement results from evaluation can be attested in a similar manner, where inputs and outputs of the evaluation are serialized and attested as declared "update functions", but verification of the scoring procedure is not performed by the leaderboard. This design allows for arbitrary evaluation processes, including ones that require an environment, to be used.

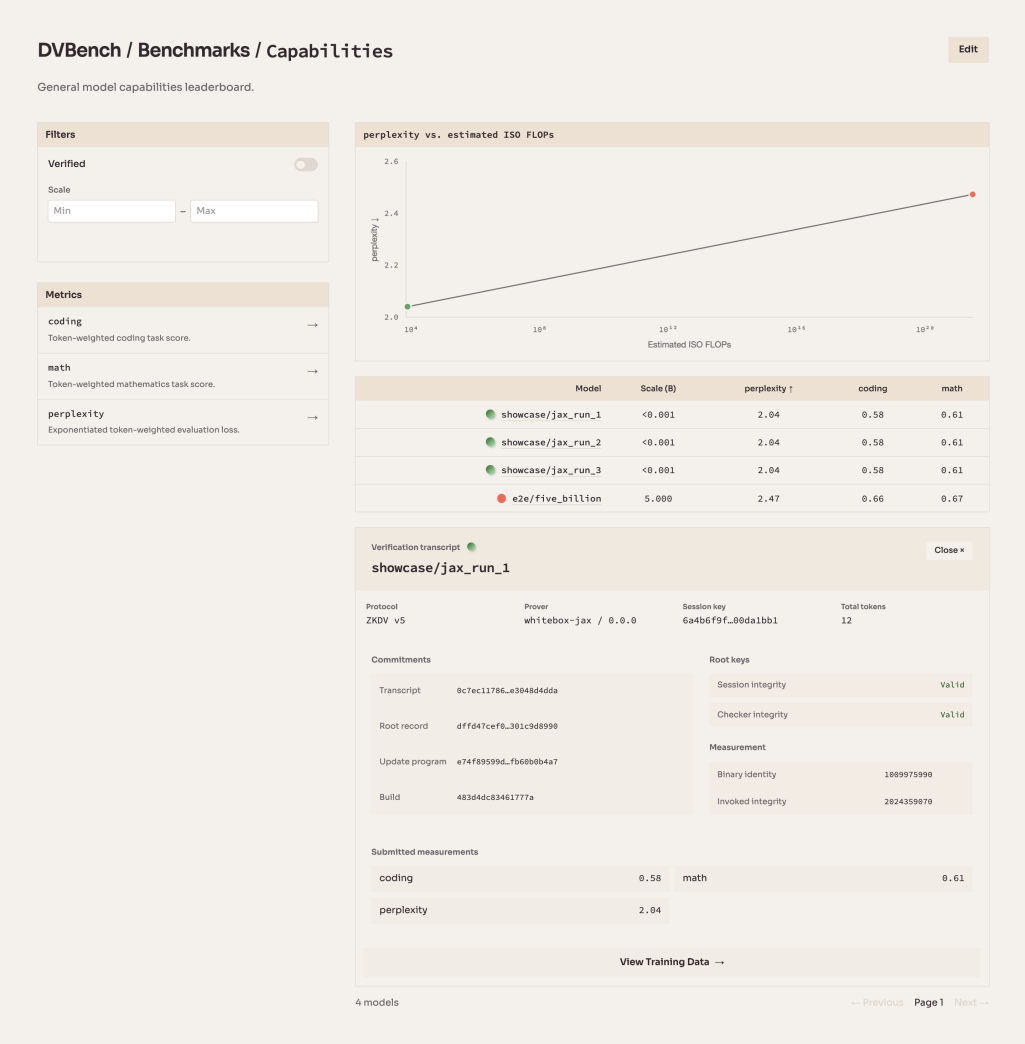


Figure 11: Verification transcript of a training run showing the metadata available to the users.

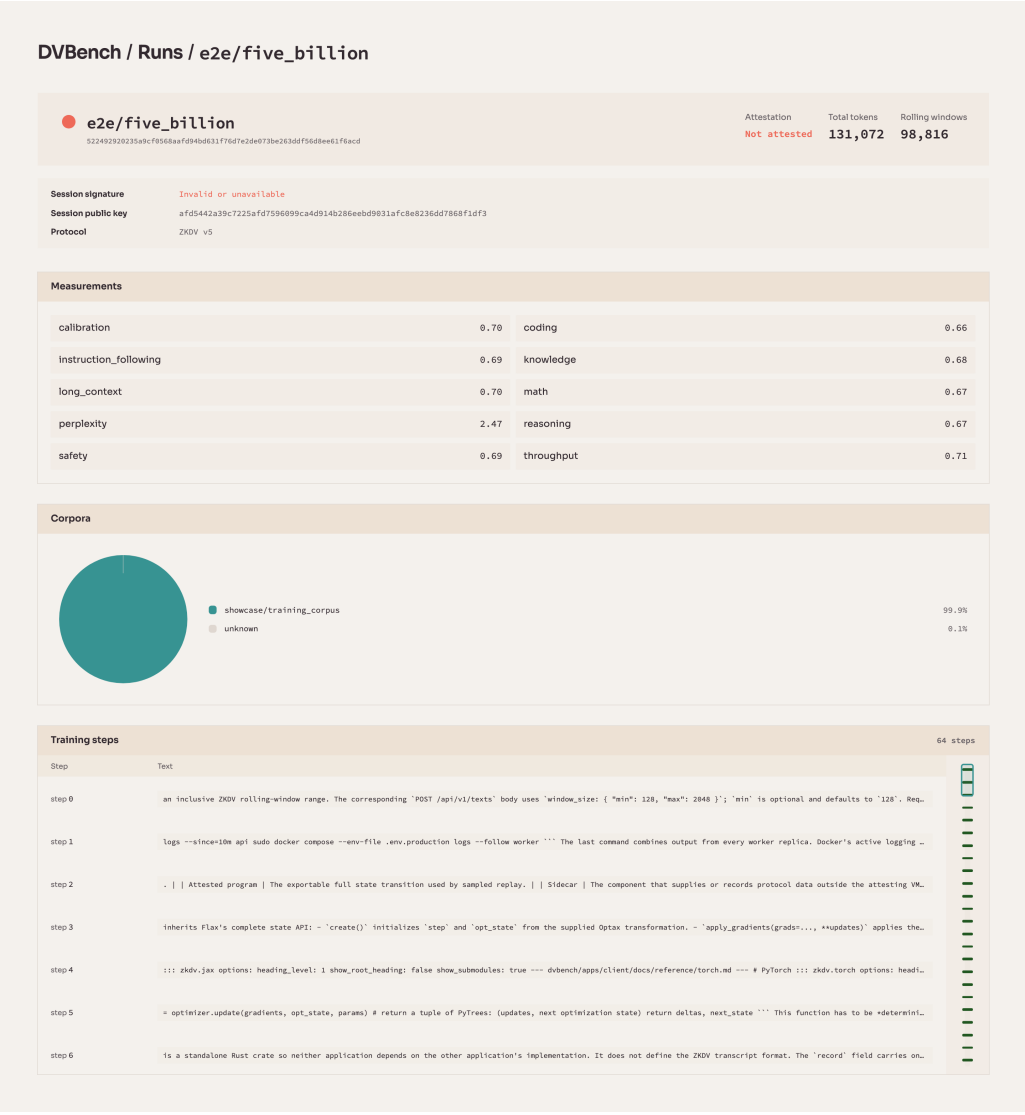


Figure 12: Detail screen of a run with training data attested and revealed; unverified runs can also be listed on our leaderboard relying on the traditional method of trusting the trainer.

B.2 Implementation Details

We leverage a three-layer architecture for our leaderboard system.

Client Streaming Client transcript information (each link in the hash chain) is streamed to the verifier using WebSockets, allowing for low-latency streaming of the certificate. The API layer receiving this stream is stateless, and will immediately inject the stream into an ephemeral NATs JetStream [2] queue for processing.

Distributed Workers Each worker holds a temporary lease on a particular transcript, instantiating a stateful verifier that pulls off of the JetStream queue and verifies the transcript. Due to our verification algorithm's small work set requirement fig. 10, we can run many verifiers in parallel and quickly serialize the verification state should trainers disconnect or workers crash.

Long-Term Database Upon verification completion, the verified state is signed and stored in a long-term database. To facilitate quickly querying dataset inclusion, we fold all of the training data hashes into a bloom filter that is stored alongside the verified transcript. The web interface then queries this database for displaying the provenance of the training run.

B.3 User Experience

Despite the relative complexity of our system, Witnesses does not actually require that many user touchpoints to use. Specifically, our algorithm only requires that the users annotate their attested function which we serialize, and wrap their training step so we can pipeline it into the attestation system. To this end, we leverage Maturin [3] to design a simple Python facade that allows easy integration of Witnesses into existing training harnesses. Below we will outline the application of this system in user code for the Torch variant of our implementation.

Connect Streaming Client Due to the relatively larger size of the proof certificate, we never store the certificate on training work and instead stream to the verifier directly. This design can also be modified to store the certificate on a local disk without changing the verification procedure. Thus, users first connect to a streaming client:

```
from dvbench import Client
client = Client(ENDPOINT)
experiment = client.experiment("owner/model")
proof = experiment.proof(backend="torch")
proof.start(rolling_window=32)
```

Wrap Model Prototypes We apply our facade to `nn.Module` prototypes by wrapping the model and optimizer with our attestation system.

```
model, optimizer = proof.prepare(model, optimizer)
```

Attest Training Step We then wrap the training step with our facade to serialize it:

```
@proof.attest
def attested_step(model, optimizer, batch):
    inputs, targets = batch
    logits = proof.forward(model, inputs)
    loss = F.cross_entropy(logits, targets)
    proof.backward(loss)
    optimizer.step()
    optimizer.zero_grad()
```

[2] https://nats.io/

[3] https://github.com/pyo3/maturin

Training Finally, users can train normally, either by invoking the attested step or using an equivalent fast path that produces numerically identical results:

```
for inputs, targets in training_batches:
    batch = (inputs, targets)
    with proof.transaction(batch=batch):
        logits = model(inputs)
        loss = F.cross_entropy(logits, targets)
        proof.backward(loss)
        optimizer.step()
        optimizer.zero_grad()
```

C Protocol Design Details

C.1 Commitment Hashes

Throughout the prose, in order to ensure protection from adaptive sampling attacks, we will use a series of $\mathcal{H}_*$ hash operations. Unless otherwise specified, $\mathcal{H}_*$ is meant to be understood as a cryptographically secure hash against $\mathcal{A}$ (in practice, BLAKE3), which contains a session-dependent stamp and secret material computed by B_{att} not revealed to $\mathcal{A}$. Inputs use an injective, domain-separated serialization ser. This per-hash stamping derives its security from the per-update ratcheted secret key in section C.4; after each commitment is stamped, the key ratchet state is advanced and cannot be re-used to evaluate another commitment.

C.2 Batch Fingerprinting

We fix a choice of L, the minimum window size for which a sequence of tokens is considered as a "sample." Using this window size, we then perform a polynomial rolling hash:

Let d_i be encoded as a sequence of tokens $d_i = \left(d_i^{(1)} \dots d_i^{(m)}\right)$. We compute two values: a secure commitment C_d and a matching hash F_d:

$$C_d(d_i) = \mathcal{H}_d(\text{ser}\,(d_i)), \qquad F_d(d_i) = \left(H_{\text{poly}}\left(d_i^{(1)}\right), \dots, H_{\text{poly}}\left(d_i^{(m-L+1)}\right)\right). \tag{5}$$

Where each sub-hash is:

$$H_{\text{poly}}\left(d_i^{(r)}\right) = d_i^{(r)} a^{L-1} + \dots + d_i^{(r+L-1)} a^0 \bmod p. \tag{6}$$

for large prime p and base a. Committing C_d allows us to ensure future sampling is a secure function of the training data, while F_d allows matching rolling hashes.

C.3 Non-Geometric Optimizer State Hash

Let's again partition the optimizer state as a series of leaves: $s_i := \left(s_i^{(1)}, \dots, s_i^{(m)}\right)$.

We define an injective view $W : \mathbb{F}_{\text{adm}} \to \left\{0, \dots, 2^{31} - 1\right\}$ which re-represents each floating-point value as an integer. To see that the representation is unique, we restrict dom W to be standard hardware-resident finite-precision floats. Then, following a T-round BLAKE-like construction (Aumasson et al., 2010), for each boundary s_i, leaf j, round t, and coordinate r, B_{att} samples independent hidden $a_{j,t,r} \sim \text{Unif}\left(\mathbb{Z}_{2^{32}}^{\times}\right)$ and $b_{j,t,r} \sim \text{Unif}\left(\mathbb{Z}_{2^{32}}\right)$. That is, we fix odd a and not-necessarily-odd b codewords as a random function of the boundary state. The same family hashes the two copies of s_i, remains unsampled until both are fixed, and is then discarded. We compute:

$$h_{j,t}(s_i) = \sum_{r=1}^{|s_i^{(j)}|} a_{j,t,r} \left[W\left(s_i^{(j)}\right)_r \oplus \left(r a_{j,t,r} + b_{j,t,r}\right)\right] \left(\text{mod } 2^{32}\right), \tag{7}$$

where $\oplus$ is the bitwise XOR operation. Finally, we define:

$$C_s(s_i) = \mathcal{H}_{\text{state}}(h_{1,1}(s_i) \dots h_{m,T}(s_i)) \pmod{2^{256}} \tag{8}$$

where $\mathcal{H}_{\text{state}}$ is any standard cryptographic Merkle hash chain.

C.4 Record Signature Ratcheting

The tape being produced establishes a signed hash chain of such record payloads, specifically that:

$$\tau_i = f_{\text{chain}_i}(\tau_{i-1}, \bar{\tau}_i) = (\bar{\tau}_i, \mathcal{H}_{\text{record}}(\tau_{i-1}), \text{vk}_{i+1}, \sigma_i) \tag{9}$$

where $\bar{\tau}_i$ is the current payload of the record (one of R, M, u_i); $\mathcal{H}_{\text{record}}$ is a cryptographic hash of records. Finally, σ_i is the signature of the current record, which is given as:

$$\sigma_i = \text{Sign}_{\text{sk}_i}\left(\mathcal{H}_{\text{sig}}(\text{ser}\left(\bar{\tau}_i, \mathcal{H}_{\text{record}}(\tau_{i-1}), \text{vk}_{i+1}\right))\right) \tag{10}$$

where $\text{Sign}_{\text{sk}_i}$ is a cryptographic signature using secret signing key sk_i, verified by public key vk_i. We expire each secret key sk_i after signing only one step.

$$(\text{sk}_{i+1}, \text{vk}_{i+1}) = \text{KeyGen}\left(\mathcal{H}_{\text{ratchet}}(\text{sk}_i)\right). \tag{11}$$

C.5 Root Record Structure

The root record R is responsible for establishing the cryptographic primitives necessary for the resulting signing and attestation. It contains a session-specific nonce, timestamp, prover build information, and protocol configuration. In order to bootstrap trust in the system, the session public key is bound by an attestation under a stable attester key from which trust is derived. Each subsequent tape record will be signed by a series of keys from the ratcheted session signing state. Logically, let:

$$R_0 = (M, \nu, c, \mathcal{H}_{\text{init}}(\text{ser}\left(\theta_0, s_0\right)), \mathcal{H}_{\text{func}}(f), \text{vk}_0), \qquad R = \left(R_0, \text{Sign}_{\text{sk}_{\text{root}}}\left(\mathcal{H}_{\text{sig}}(\text{ser}\left(R_0\right))\right)\right). \tag{12}$$

where $\text{Sign}_{\text{sk}_{\text{root}}}$ is a signature made by a stable root-of-trust key, and M is session-specific metadata.

C.6 Measurement Record Structure

In order to ensure trust in the authenticity of the recording binary B, we additionally add trusted self-measurements as a record in the binary. This involves two measurement words m_1, m_2 and a cryptographic hash $M = (\mathcal{H}_{\text{measure}}(m_1), \mathcal{H}_{\text{measure}}(m_2))$, where m_1 attests the binary portion of the system and m_2 attests any running code by both script analysis and live interpreter freezing. Details on the second step are given in section F.4.

C.7 Update Record Structure

Let $C_\theta : \mathbb{R}^{|\theta|} \to \mathbb{R}, C_s : \mathbb{R}^{|s|} \to \mathbb{R}$ be the parameter and state sketching functions described in section 4.2, and let C_d be the data commitment described in section 4.2.2. Furthermore, let $\text{PRNG} : \{0,1\}^* \to [0,1]$ be a uniform probabilistic random number generator.

The logical "update record" consists of five sub-records $u_i^{(1)} \dots u_i^{(5)}$, constructed chronologically.

Pre-State Commitment Given θ_i, s_i and the user's declared batch d_i, we compute and record parameter, state, and data sketches from before the update occurs: $u_i^{(1)} = (C_\theta(\theta_i), C_s(s_i), C_d(d_i), F_d(d_i))$.

Post-State Commitment A computes the training algorithm following $\tilde{f}(\theta_i, s_i, d_i)$, yielding θ_{i+1}, s_{i+1}. We record these outputs, as declared by A, as well: $u_i^{(2)} = (C_\theta(\theta_{i+1}), C_s(s_{i+1}))$.

Deterministic Sampling Let $\mathcal{H}_{\text{record}}$ be a cryptographically secure hash; we follow a known PRNG function and compute $u_i^{(3)} = p_{\text{challenge}} = \text{PRNG}\Big[\mathcal{H}_{\text{record}}\Big(\nu, \tau_{i-1}, u_i^{(1)}, u_i^{(2)}\Big)\Big]$. This is our probabilistic sampling decision: *we only challenge* $f \cong^? \tilde{f}$ *when* $p_{challenge} \leq p_{check}$. This ordering provides the unpredictability property of challenges: A cannot cheat on only what will not be challenged, and $\mathcal{H}_{\text{record}}$ is ratcheted upon computation to prevent continuous search.

Challenge Replay If our records are challenged in $u_i^{(3)}$, B restores the committed function f and computes $f(\theta_i, s_i, d_i) = \theta'_{i+1}, s'_{i+1}$. We compare $C_\theta\big(\theta'_{i+1}\big)$ and the full s'_{i+1} against committed values, then challenge one slot x of θ'_{i+1} and commit $u_i^{(4)} = \Big(\theta_{i+1}^{(x)}, \theta_{i+1}^{(x)'}\Big)$. Note that while the state buffer and parameter sketch are locally *checked*, only one parameter is *attested* on tape.

Checkpoint Commitment If A wants to emit a checkpoint, we transfer it to host, compute $h_i^{\text{ckpt}} = \mathcal{H}_{\text{ckpt}}\Big(\text{ser}\Big(\nu, \tau_{i-1}, u_i^{(1)}, \ldots, u_i^{(4)}, \theta_{i+1}\Big)\Big)$, and commit it. We additionally sample q positions in the model weight and commit them. That is, we sample $x_1 \ldots x_q = \text{PRNG}\Big[\mathcal{H}_{\text{record}}\Big(h_i^{\text{ckpt}}\Big)\Big]$ and commit $u_i^{(5)} = \Big(h_i^{\text{ckpt}}, \theta_{i+1}^{(x_1)} \ldots \theta_{i+1}^{(x_q)}\Big)$.

D Verification Procedure

Our verification algorithm checks a given transcript τ, initial state (θ_0, s_0), final checkpoint θ_f, predeclared training algorithm $\mathcal{H}_{\text{func}}(f)$, and nonce ν for their declared structure, and ensures that every update is structurally sound.

Attestation Checking V_{sig} We examine the binding of B_{att} to B_{host}. We check that the root session key derivation in the root record R is valid from the published B_{att} program. We **reject** if initial attestations are not signed from known builds, or if (in the zkVM case) initial session materials are not constructed from a known algorithm. Otherwise, we **continue**.

Parameter Binding V_{bind} We then check if the checkpoint we want to attest θ_f actually belongs to the tape τ. To do this, we extract $Q_1 \ldots Q_m \in \tau$, the parameter sketch described in section 4.2.1, and apply them again to θ_f. After hashing, this produces a single digest $C_\theta(\theta_f)$ which we search for in the tape. We also check that the checkpoint's full commitment h_i^{ckpt} recomputes from θ_f, that it's in the tape, and that the sampled values match. **Reject** if any check fails; **continue** otherwise.

Replay Binding V_{func} We then check that the checked transition was consistent with the declared transition. It is not important that this transition is disclosed to the verifier, as long as it is consistent. We first **reject** if $\mathcal{H}_{\text{func}}(f) \notin \tau$, and then check the attestation bound to $\mathcal{H}_{\text{func}}(f)$ in the tape; we **reject** if the attestation is invalid (e.g., it's not derived from the session key, or—in the non-whitebox binding setting—the zkVM execution tape is invalid). Otherwise, we **continue**.

Tape Integrity Checking V_{tape} We next check the integrity of the tape by monitoring each of its transitions. Specifically, we follow the ratcheting design of section C.4 to ensure the integrity of each chain link, bind the first record to (ν, θ_0, s_0), check adjacent pre/post parameter and optimizer-state commitments, and check the declared tape contents' hash against the signed hash material. We **reject** if any check fails, and **continue** otherwise.

Sampled Replay Checking V_{replay} Lastly, we check that the challenged value and PRNG sampling were done correctly for the challenged updates. Specifically, we verify the challenge derivation, replayed parameter sketch, full optimizer state, and sampled parameter value, and verify that the application of f matches the same hash as the one bound in $\mathcal{H}_{\text{func}}$. **Reject** if replays are incorrect or sampling is not followed; **accept** otherwise.

E COMPLETE PROOF

We now give the full proof of the soundness statement made in section 5.

E.1 FORMALISM

For an n-step training run, A produces parameter states $\theta_0 \dots \theta_n$ and optimizer states (and metadata, such as LR schedule) $s_0 \dots s_n$ using data samples $d_0 \dots d_{n-1}$. s_i or d_i will also include computed or supplied random seeds, respectively. A thus follows a deterministic procedure $\tilde{f} : (\Theta, S, D) \to (\Theta, S)$ from a combination of parameters Θ, state S, and data D to the corresponding next parameter and state.

Confirmation $c \in C$ includes $p_{\text{check}}, p_{\text{sketch}}, T, q$ (check probability, sketch size, hashing rounds, and number of checkpoint sketch vectors) and the cryptographic parameters. Our goal is to develop $\mathcal{P} : (\Theta, \Theta, S, S, D, C) \to T$ such that $\mathcal{P}(\theta_i, \theta_{i+1}, s_i, s_{i+1}, d_i, c) = \tau_i$ which maps each of the transitions above into linkages in a tape format $\tau_i \in T$.

Define now also an honest training procedure: $f : (\Theta, S, D) \to (\Theta, S)$. We ask that f has bounded description length $K(f) \leq B$, where K is Kolmogorov complexity, such that any information used in training in practice comes from the data source itself. In practice, we can instantiate this as source-code length as lowered into an intermediate format; see section F.1.

Let $\mathcal{H}_{\text{func}}(f)$ be a one-way commitment of f (e.g., hashing the code of f). We define a verifier $V(\tau, c, \nu, \theta_0, s_0, \mathcal{H}_{\text{func}}(f), \theta_f) \in \{0, 1\}$ which should accept with the following properties. Here, $\theta_f \in \tau$ means that checkpoint θ_f is bound to some checkpoint release record in τ.

E.2 THREAT MODEL GUARANTEES

Let's start by implementing our threat model as a series of axioms which we appeal to. We first operationalize that A is a bounded polytime adversary for which our hashes are sound:

Axiom E.1 (cryptographic primitives)**.** Every $\mathcal{H}_*$ used in section 4 admits a cryptographic hash, so for all machines $\mathcal{A}$ that are under our scope:

Hashes are Secure

$$\Pr\left[\mathcal{A} \text{ finds } x, y \text{ with } \text{poly}\left(\lambda\right) \text{ executions s.t. } \mathcal{H}_*(x) = \mathcal{H}_*(y), x \neq y\right] \leq \text{negl}\left(\lambda\right). \tag{13}$$

Bindings are Secure Sign is EUF-CMA secure, commitments are binding, and B_{att} is sound under the same bound.

where λ is a standard security parameter, $\text{poly}\left(\lambda\right)$ is any polynomial function of λ, and $\text{negl}\left(\lambda\right)$ is a negligibly small function of λ.

Next, we describe how we ensure the freshness of each run:

Axiom E.2 (challenge freshness)**.** Conditioned on each position in the tape, sampling a coordinate position to check under p_{check} is uniform and independent. V registers one attempt for each nonce ν; aborting a run results in rejection.

This nonce setup means that an attacker cannot grind our randomly sampled checks end-to-end until a single run results in no cheating samples being detected. In practice, this risk is prevented by a random whitebox-generated nonce, and the practical fact that $p_{\text{check}} = 0.01$ means it takes in expectation $\frac{1}{0.01} = 100$ tries of full training to get one valid sample, which is impractical.

We now turn to a fact that allows us to maintain buffer identity between updates:

Axiom E.3 (attested parameter continuity)**.** B_{att} enforces post/pre parameter-buffer identity across adjacent updates, and V validates its attestation.

We also need to formalize that A cannot grind through every possible training trajectory using unbounded time to find a cheating trajectory, which supports our bounded-computation threat model:

Axiom E.4 (prover is limited)**.** All machines $\mathcal{A}$ under our scope are probabilistic polynomial-time in λ and their input length.

where λ is a security parameter.

E.3 Optimizer State Binding

Let's begin now by discussing some statements about C_s and show that this simple reduction admits an amplifiable, non-geometric fingerprint. We will restrict our statement to one leaf of the hash in section C.3, and appeal to section E.2 for the soundness of the final reduction.

First, we will use the following lemma about the structure of the fold:

Lemma E.5 (optimizer binding fold collision)**.** *For distinct leaf values $x, y \in \mathbb{F}_{\text{adm}}^N$, one round of the fold in section C.3 satisfies:*

$$\Pr\left[h_{j,t}(x) = h_{j,t}(y)\right] \leq \frac{1}{2}. \tag{14}$$

Proof. We desire that one round collides with probability at most $1/2$.

For simplicity, choose a coordinate r such that $W(x_r) \neq W(y_r)$, and let v be the location of their smallest differing bit. Since $W(x_r), W(y_r) < 2^{31}$, we have $v \leq 30$.

Consider any $z, x, y \in \mathbb{Z}_{2^{32}}$, and the bitwise XOR operation $\oplus$. The following statement is true:

$$v_2([x \oplus z] - [y \oplus z]) = v_2(x - y). \tag{15}$$

This is because XOR by the same z preserves the smallest differing bit.

Using this fact, condition on all other coordinates and let $z = ra_{j,t,r} + b_{j,t,r}$. Since $b_{j,t,r}$ is uniform, z is uniform and independent of the odd $a_{j,t,r}$. Call

$$d = [W(x_r) \oplus z] - [W(y_r) \oplus z]. \tag{16}$$

Applying eq. (15), $v_2(d) = v$. Multiplication by uniform odd $a_{j,t,r}$ is therefore uniform over the 2^{31-v} residues of valuation v.

Finally, the conditioned sum can equal zero for at most one such residue, so:

$$\Pr\left[h_{j,t}(x) = h_{j,t}(y)\right] \leq 2^{-(31-v)} \leq \frac{1}{2}. \tag{17}$$

In particular, this gives the desired per-round collision bound. □

The proof idea here is that finite computation precision (e.g., mod 2^{32}) preserves the least significant bit of difference between two inputs, while the random odd multiplier shuffles the resulting residue class uniformly. We give the full proof in section E.3. Using this fact, we can state an important result about our binding.

That is, each round of our binding admits a "uniform shuffling" within a residue class keyed by a, b. We will now use this fact to finish our claim about the security of our binding:

Lemma E.6 (Security of C_s)**.** *For distinct $x, y \in \mathbb{F}_{\text{adm}}^N$,*

$$\Pr\left[C_s(x) = C_s(y)\right] \leq 2^{-T} + \operatorname{negl}(\lambda). \tag{18}$$

Proof. Choose a differing leaf j. From lemma E.5, each round of h_j collides with probability at most $1/2$. T such rounds, recall, with fresh a, b commitments, collide with probability at most 2^{-T}. Finally, we appeal to section E.2, which guarantees final fold security up to a negligible bound $\operatorname{negl}(\lambda)$ for a security parameter λ, to obtain the final collision bound. □

In practice, any precision-preserving collision usually degrades model output quality.

We have now shown that C_s is a sound commitment scheme for optimizer state parameters. We will move on to showing the same for C_θ next, with the additional constraint that the initial sketching component must also be *geometric*.

E.4 PARAMETER SKETCHING

Let's now prove a similar bound for the fast geometric sketch described in section 4.2.1. Unlike the non-geometric hash of the optimizer state, this sketch must be computable after the fact by the verifier independently, so it cannot derive its power by a series of constantly refreshed, state-dependent shuffled a, b, and must survive floating-point precision differences between hardware. However, we prove the security of our system below using exact arithmetic, and leave to empirical work in section 7 how this sketch performs in practice.

Recall that we sketch by a series of projections using orthonormal matrices $Q_j \in \mathbb{R}^{n_j \times l_j}$. As before, let's consider only a single leaf and call it $Q \in \mathbb{R}^{n \times l}$ for some parameter-matrix domain n and $l = \lceil p_{\text{sketch}} n \rceil$, and look to section E.2 for the security of the final fold.

Lemma E.7 (Q^T matrices are full spark)**.** *Every set of l columns in Q^T is linearly independent. That is, spark$\left(Q^T\right) = l + 1$ almost surely.*

Proof. Q is generated from sampled Gaussian values; thus every $l \times l$ minor is nonsingular almost surely. Thus each choice of l columns of Q^T is linearly independent, and thus you must choose $l + 1$ columns for linear dependence. Orthogonalization preserves rank. Thus the spark of Q^T is $l + 1$ almost surely. □

Intuitively, lemma E.7 can help us use the linearly independent projections to "pin down" degrees of freedom from the weight matrices we are attesting through multiplication. We can formalize this by observing the behavior of Q applied to two differing matrices.

Lemma E.8 (amount of cheating is lower-bounded)**.** *Let $\theta \neq \tilde{\theta}$ be rows of a single weight matrix, and call $\delta = \tilde{\theta} - \theta$. $\delta Q = 0, \delta \neq 0$ implies $\|\delta\|_0 \geq l + 1$.*

Proof. We have $\delta \neq 0, \delta Q = 0$. Suppose for the sake of contradiction that $\exists \delta$ such that $\|\delta\|_0 < l + 1$. Then this implies that there exists a linearly dependent set of rows in Q of size $\leq l$. But Q is full spark, reaching a contradiction. □

Now, fix p_{check} as the probability that an update is challenged, followed by one uniformly sampled parameter, as in section 4.1. Let $\Theta \neq \tilde{\Theta}$ be two distinct parameter trees, and let $\Delta = \tilde{\Theta} - \Theta$. Further define:

$$G(\Delta) = \min\left(|\Theta|, \sum_j \sum_r \mathbf{1}_{\Delta_r^{(j)} \neq 0}(l_j + 1)\right), \tag{19}$$

where r indexes matrix rows. Using this G construction, we are ready to claim the size of cheating we can prevent with our sketch:

Lemma E.9 (Security of C_θ)**.** *The chance that $\Theta \neq \tilde{\Theta}$ survives unnoticed is upper-bounded:*

$$\Pr\left[\textit{undetected}\right] \leq 1 - p_{\textit{check}}\left(\frac{G(\Delta)}{|\Theta|}\right) + \text{negl}\left(\lambda\right). \tag{20}$$

Proof of lemma E.9. First, observe that the sketch not detecting a difference implies either no cheating or

$$\|\Delta\|_0 \geq G(\Delta) \tag{21}$$

by considering each row of Δ and applying lemma E.8. Consider now the single-parameter challenge scheme described in section 4.1. For every "checked" parameter, we obtain exact assurance for the computation of the parameter. Thus, on a challenged update, a cheating parameter going unnoticed has to survive two events: E_n that it has to lie in the nullspace of the sketch, and E_c, that it goes unchecked. Specifically:

$$\Pr\left[\text{undetected}\right] \leq (1 - p_{\text{check}}) + p_{\text{check}} \Pr\left(E_c \mid E_n\right) \Pr\left(E_n\right) + \text{negl}\left(\lambda\right) \tag{22}$$

with security parameter of the final hash λ.

Since we are working against an adaptive $\mathcal{A}$, who may explicitly optimize for $\Pr(E_n)$ (e.g., all her parameters lie in the sketch null space), let's set the worst case:

$$\Pr[\text{undetected}] \leq (1 - p_{\text{check}}) + p_{\text{check}} \Pr(E_c \mid E_n) + \text{negl}(\lambda). \tag{23}$$

Now begin by observing that:

$$\Pr(E_c \mid E_n) = 1 - \frac{\|\Delta\|_0}{|\Theta|} \tag{24}$$

that is, a uniformly sampled check will only catch a parameter if they differ in the delta, and we are sampling the checked parameter uniformly; note that we only challenge an update with chance p_{check}. Now apply the fact that we are conditioned on E_n, that is, we know from above that $\|\Delta\|_0 \geq G(\Delta)$ if the parameters lie in the sketch null space; we can write:

$$\Pr(E_c \mid E_n) \leq 1 - \frac{G(\Delta)}{|\Theta|}. \tag{25}$$

Finally, replacing the LHS and carrying the inequality through:

$$\Pr[\text{undetected}] \leq 1 - p_{\text{check}} \left(\frac{G(\Delta)}{|\Theta|} \right) + \text{negl}(\lambda). \tag{26}$$

□

The proof idea here is to apply lemma E.8 to every row of each parameter matrix, and to observe that either the sketch catches a cheat or the cheat lies in the sketch null space, meaning the sampling has more chances to catch the cheat. A full proof of this is given in section E.4.

E.5 Tape Integrity

Now that we have proofs for each of our basic objects, we will now complete the proof by making some statements about the proof tape appealing to usual cryptographic primitives, before moving on to our final theorems.

Let's first make the statement which makes explicit the "chained" structure of our tape:

Lemma E.10 (state continuity)**.** *Verifier **accept** means that for every update u_i, $i \in \{0, \ldots, n-2\}$ for n training steps, we have:*

$$\theta^{post}(u_i) = \theta^{pre}(u_{i+1}) \tag{27}$$

$$C_\theta^{post}(u_i) = C_\theta^{pre}(u_{i+1}) \tag{28}$$

$$C_s^{post}(u_i) = C_s^{pre}(u_{i+1}) \tag{29}$$

that is, each update uses parameter material that the previous update generated, and optimizer-state material up to hash collision.

Proof. V_{sig} validates section E.2, and V_{tape} as described in section D explicitly checks the commitments. By lemma E.6, optimizer-state substitution succeeds with probability at most $2^{-T} + \text{negl}(\lambda)$. □

At this point, we can formalize that every form of cheating within a transition is caught by at least one previously-bounded mechanism. Let f be the attested transition and $\tilde{f}$ be a potentially-malicious transition that is executed, and let $\theta', s' = f(\theta, s, d)$, and $\tilde{\theta}, \tilde{s} = \tilde{f}(\theta, s, d)$. Define $\Delta = \theta' - \tilde{\theta}$, $g_{\min} = \min_j (l_j + 1)$, and

$$\epsilon_\theta(\Delta) = \min\left(1, 1 - p_{\text{check}} \frac{G(\Delta)}{|\Theta|} + \text{negl}(\lambda)\right), \tag{30}$$

$$\epsilon_s = 1 - p_{\text{check}}, \tag{31}$$

$$\bar{\epsilon}(c) = \max\left(\min\left(1, 1 - p_{\text{check}} \frac{\min(g_{\min}, |\Theta|)}{|\Theta|} + \text{negl}(\lambda)\right), 1 - p_{\text{check}}\right). \tag{32}$$

Lemma E.11 (all cheats are bounded)**.**

$$\Pr\left[V \textit{ accepts update } i \mid \theta' \neq \tilde{\theta} \vee s' \neq \tilde{s}\right] \leq \bar{\epsilon}(c). \tag{33}$$

Proof. We consider casework. For the case where $\theta' \neq \tilde{\theta} \vee s' \neq \tilde{s}$ (i.e., there was a functional cheat), only one of three things may occur:

$\theta' \neq \tilde{\theta}, s' = \tilde{s}$ In this case, our parameter update can fall only into one of two mutually exclusive cases: $C_\theta(\tilde{\theta}) \neq C_\theta(\theta')$ or $C_\theta(\tilde{\theta}) = C_\theta(\theta')$. In either case, we directly appeal to lemma E.9 to see that the chance of undetection is at most $\epsilon_\theta(\Delta) \leq \bar{\epsilon}(c)$. That is, $\Pr\left(\text{accept} \mid \theta' \neq \tilde{\theta}, s' = \tilde{s}\right) \leq \epsilon_\theta(\Delta)$.

$\theta' = \tilde{\theta}, s' \neq \tilde{s}$ Similar to before, our state update can fall into one of two mutually exclusive cases: $C_s(\tilde{s}) \neq C_s(s')$ or $C_s(\tilde{s}) = C_s(s')$. In either case, exact challenged replay gives a chance of undetection at most $\epsilon_s \leq \bar{\epsilon}(c)$. That is, $\Pr\left(\text{accept} \mid \theta' = \tilde{\theta}, s' \neq \tilde{s}\right) \leq \epsilon_s$.

$\theta' \neq \tilde{\theta}, s' \neq \tilde{s}$ The probability of acceptance for cheating in both is at most the minimum of the two as the first-detected failure would result in rejection. That is, $\Pr\left(\text{accept} \mid \theta' \neq \tilde{\theta}, s' \neq \tilde{s}\right) \leq \min\left(\epsilon_s, \epsilon_\theta(\Delta)\right)$.

Thus the disjunction of all three cases is upper-bounded by $\max\left(\epsilon_\theta(\Delta), \epsilon_s\right) \leq \bar{\epsilon}(c)$. □

With this lemma, we are now ready to claim our desired statement about tape soundness.

Theorem E.12 (Tape update checking is sound)**.**

$$\Pr\left[V(\tau, c, \nu, \theta_0, s_0, \mathcal{H}_{\textit{func}}(f), \theta_f) = 1 \mid I_f(\tau) \geq k\right] \leq \varepsilon_1(k, c). \tag{34}$$

Proof. Notice that given lemma E.10, a cheating tape is recognized at least as much as each of its individual cheats are recognized. Let $J = \{i : \text{update } i \text{ is invalid}\}$, and let E_i be the event that cheating survives step i. By the chain rule, we have that:

$$\Pr\left[\bigcap_{i \in J} E_i\right] = \prod_{i \in J} \Pr\left[E_i \mid E_{<i}\right]. \tag{35}$$

Applying section E.2 and lemma E.11, we have:

$$\Pr\left[\bigcap_{i \in J} E_i\right] \leq \prod_{i \in J} \bar{\epsilon}(c) = \bar{\epsilon}(c)^{|J|}. \tag{36}$$

Finally, our verifier will reject all faulty updates it finds, so it will not reject only when updates slip past unnoticed, so:

$$\Pr\left[\text{accept}\right] \leq \bar{\epsilon}(c)^{|J|} \leq \bar{\epsilon}(c)^k. \tag{37}$$

Thus, define $\varepsilon_1(k, c) = \bar{\epsilon}(c)^k$, which gives our desired cheating rate. □

E.6 Checkpoint Integrity

We can now prove the other side of the soundness claim by essentially repeating the argument given in lemma E.9. Specifically, we have:

Corollary E.13 (Security of final checkpoint $C_\theta(\theta_f)$)**.** *For $\Delta = \tilde{\Theta} - \Theta$ fixed by the full checkpoint commitment, the chance that $\Theta \neq \tilde{\Theta}$ survives unnoticed with q committed value samples is upper-bounded:*

$$\Pr\left[C_\theta(\Theta) = C_\theta\left(\tilde{\Theta}\right) \wedge \textit{samples agree}\right] \leq \left(1 - \frac{G(\Delta)}{|\Theta|}\right)^q + \text{negl}\left(\lambda\right). \tag{38}$$

This follows from exactly the argument given in section E.4 for lemma E.9, except there are now q independently sampled checked locations after the full commitment. Thus, the joint probability of all q challenges missing is the product of the probabilities of each challenge missing.

From this, we can then give the second part of our soundness result:

Theorem E.14 (Checkpoint commitment is sound)**.**

$$\Pr\left[V(\tau, c, \nu, \theta_0, s_0, \mathcal{H}_{\mathit{func}}(f), \theta_f) = 1 \mid I_f(\tau) = 0, \theta_f \notin \tau\right] \leq \varepsilon_2(c). \tag{39}$$

Proof. Write $h_i^{\text{ckpt}}(\theta)$ for the checkpoint hash in section 4.1 with the same release prefix. For $\theta_f \notin \tau$, there are two cases. Either $h_i^{\text{ckpt}}(\theta_f) \notin \tau$, in which case V_{bind} rejects with probability 1, or $h_i^{\text{ckpt}}(\theta_f) \in \tau$ yet θ_f is not part of training. This implies that $\exists \theta'_f \in \tau$ such that $h_i^{\text{ckpt}}\left(\theta'_f\right) = h_i^{\text{ckpt}}(\theta_f)$. If so, section E.2 gives directly a negligible bound. Define

$$\varepsilon_2(c) = \text{negl}\,(\lambda). \tag{40}$$

This gives our desired cheating rate. □

E.7 Additional Properties

We state, but do not show, four additional claims about our tape. The first two are informally true by inspection, and the first three are evaluated empirically in section 7.

Streamable Tape We make no attempts at specifying the structure of the data, so we simply ask that our tape is linearly sized $|\tau| \leq \kappa(c)n$, where $\kappa(c) > 0$ and that verification happens in one pass with a working set that is $O(1)$ to training updates n.

Easily Checkable Tape We require that verification costs substantially less work than fully reproducing the whole training run. We leave this statement to be less formal than others to enable independence between different platforms and lowering, and with the expectation that "substantially less" is nevertheless still an asymptotically linear function of the original training length.

Logit Checkpointing During our update function, the proof tape can optionally store the full top-k logit values for each input in the batch. This allows for additional security because of the results shown in fig. 14 and fig. 8, where we show that even if an attacker is able to find a parameter update that evades the sketch, it is still unlikely to do so without meaningfully changing training distribution performance.

Initial-State Attacks To enable support for tasks like fine-tuning, we do not make any claims about the initial state parameters other than fine-tuning, and assume that A cannot sneak unbounded undeclared information through their initial state that's not visible by inspection since the verifier also has access to θ_0, s_0.

Since we want our proofs to be general over the model architecture itself, we do not assume that the model has any specific behavior with respect to its inputs, and thus leave the theoretic bounds of these statements unqualified.

F Performant Systems Design

We now discuss some aspects of the system design that are critical for both the security and the practical performance results given in section 8.

F.1 Update Program Serialization

Soundness of the algorithm depends on attesting and running a pure update function $f(\theta, s, d) \to (\theta', s')$. Enforcing this over typical ML code is non-trivial because there's no obvious decomposition of arbitrary user code without explicit side effects.

We therefore must design a stable, serializable closure over any user's choice of f. This closure is important for two reasons: first, we need to make sure that side-channel information cannot be passed through a side-channel into f, thus defeating the purpose of d being the only source of data; second,

Time	Driver (D)	Worker (W)	Broker (B)
		flowcontrol ready to admit update i	
t_1	Open transaction i (D → B)		Commit data i, challenge $i-1$
t_2	Request snapshot (D → W)	Copy θ_i, s_i (W → A → W)	
t_3	Request training (D → W)	Compute $\tilde{f}, C_s, C_\theta$ (W → A)	*fully packed, pipelined*
		Wait for $\tilde{f}, C_s, C_\theta$, donate θ_i	
t_4	Receive results (W → D)	Copy θ_{i+1}, s_{i+1} (A → D)	
t_5	Submit results (D → B)		Sign and write to tape
t_6	Open transaction $i+1$ (D → B)		Commit data $i+1$, challenge i
		if unsampled, asynchronously	
t_7	Request memory free (D → W)	Free s_i (W → A)	Finalize transaction i
		else, asynchronously	
t_7	Request verification (D → W)	Compute $\theta_{i+1} - \delta_i$ (D → A)	Publish challenged index (B → W)
t_8		Wait for verified $\theta^{(j)}_{i+1}$	
t_9	Submit checked results (D → B)	Copy $\theta^{(j)}_{i+1}$ (A → D)	Check, sign, and write to cert.
t_{10}	Request memory free (D → W)	Free s_i (W → A)	Finalize transaction i

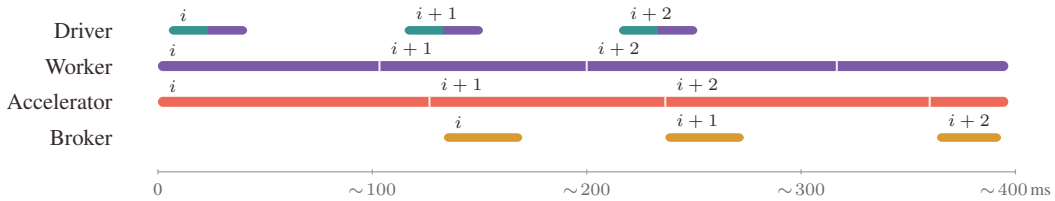


Table 1: Execution timelines of our system; colors represent single logical actions. Python driver (D) coordinates the training process, Rust-side worker (W) dispatches and coordinates training, Rust broker (B) coordinates the transitions and records the output certificate, and Accelerator (A) chip.

we need to bound the size of $|f|$ by explicit measurement (i.e., an f that closes over all pretraining data does not need d to add information to the model).

To achieve this, we use the insight that *accelerator program serialization already provides a source of functional environment closure* which we can serialize and reuse. Our Python facade observes any user code which ran during one attested training step, and then calls the standard serialization pathway on the resulting graph which the user returns to us. This graph serialization is already available as `jax.export.export` for Jax and `torch.aot_compile.export` for Torch.

This setup requires only the user to disclose what tensor corresponds to which logical elements (e.g., tensor, optimizer state, etc.), and assumes all other inputs and outputs are static information to be closed on. The resulting compiled binary is then measured for size, signed, and identically rehydrated for use whenever computation on f is required.

F.2 Interleaved State Offloading and Recovery

In order to minimize memory usage for any given flow control configuration, we make two interleaved offloading decisions that minimize and hide the overhead of memory copying and loading.

For any given update u_i, there are five data structures worth considering: $\theta_i, \theta_{i+1}, \Delta_i, s_i, s_{i+1}$. In naive training, we keep three buffers `T`, `D` and `S`, whereby we compute logically `*T, *S = update(*T, *S)` and use `D` to store Δ_i intermediates. This setup "donates" buffers for θ_i, s_i (`T` and `S` respectively) for use by θ_{i+1}, s_{i+1} respectively.

However, naively implementing our algorithm requires *all five* buffers to be kept in memory, in case an update is challenged and thus we must run the committed update function on the attested path using the original inputs.

In order to prevent this, our pipelined update program eagerly dispatches asynchronous copies of the optimizer state and allows usual buffer donation logic to free the optimizer state buffers after copying. Since this copying is interleaved with the forward pass, it usually produces little to no bubbles under the same HBM operation. If an update is challenged, the already-donated θ_i parameters are backed out via $\theta_i = \theta_{i+1} - \Delta_i$ while the cached state is loaded from host memory. We illustrate the design of this offloading scheme in the pseudocode presented in fig. 13.

```
void update(buf *T, buf *S, scratch_buf *D) {
  host_buf *S_host = async_copy(S); // eager copying

  *D = compute_deltas(T, S)
  *S = compute_next_state(T, S)
  *T = apply_update(T, D);

  int sketch_host = sketch(T,S,D);

  if (challenged_p(T,S)) { // non hot-path
    S_host -> await();
    *S = copy(S_host);
    S -> await() // expensive
    *T = *T - *D;

    assert sketch_host  == check(T,S,D);
  }
}
```

Figure 13: A logical illustration of offloading design. We don't wait for state offload to finish if an update turns out to be unchallenged. This is sound since we always wait when we need the value in the host buffer; host-side memory can otherwise be without coherence without affecting correctness.

F.3 Eliding Offload Computation

One other technique that can be used to trade off throughput and memory involves not offloading the optimizer state during every single step, and instead doing so at a fixed interval. Following Choi et al. (2023), if a step that is not offloaded eventually becomes challenged, we can simply replay training from the last offloaded step until the challenged step for evaluation. The pace for this offload can be tuned optimally based on empirical measurements of the interconnect speed and compute throughput (it's better to offload more frequently if the interconnect is fast).

Specifically, let $p = p_{\text{check}}$ be the probability we sample an update, N be the number of updates between offloads, and $r = \frac{1}{N}$. Let offloading cost γ and replay cost ϵ per transition. The amortized per-update cost is:

$$g(r) = \gamma r + \frac{1}{2}\epsilon p\left(\frac{1}{r} - 1\right) \tag{41}$$

This expression is strictly convex and has the solution:

$$r^* = \sqrt{\frac{\epsilon p}{2\gamma + \epsilon p}} \tag{42}$$

We can then use this rate to tune the offload frequency to minimize the amortized cost of offloading and replaying.

F.4 Invoked Python Sealing

Our attested worker must on occasion call Python APIs to manipulate opaque tensors obtained by Python. In order to prevent memory movement or re-encoding of these objects, we share the same interpreter as the user program. This design introduces a meaningful new security surface: whereby user code runs interleaved with the attested Python calls, thus surfacing potential vulnerabilities where users modify or patch runtime state of attested programs. We provide a best-effort closure which prevents easy exploitation of this vulnerability. Note that a non-whitebox system would use proof of computation techniques to replace these (light) functions.

In order to prevent state modifications, we protect six types of Python state:

Fingerprint The identity of each attested function is copied and hashed by observing its bytecode. During each call, we repeat this procedure to check that the fingerprints did not change.

Environment For every attested function, its object, `__globals__`, value defaults, keyword defaults, any closed state, and any modules are copied using `deepcopy`. We also check during execution that none of its arguments has a mutable input type.

Code constraints We inspect each attested function's Python code at each runtime to identify global reads. Explicit reads to dictionaries, lists, sets, byte arrays are rejected at runtime. Specific reflexive operations are explicitly rejected during attested runtime: `eval`, `exec`, `globals`, `getattr`, `setattr`, `open`, `__import__`.

Bytecode constraints We inspect each attested function's bytecode to prevent global reads, rejecting functions that invoke `STORE_GLOBAL`, `STORE_ATTR`, `STORE_SUBSCR`, `DELETE_*`, `IMPORT_*`.

Modules Each imported function from third-party modules is also sealed recursively but without AST walking. To ensure this is sound, we ensure that our system is the first import before imported modules (we enforce ordering by poisoning imported module context), since we control the correct implementations of each attested function.

Torch/Jax JIT Just-in-time compiled functions are sealed prior to compilation, and we build a new JIT wrapper and request compilation ourselves. The resulting compiled binary is sealed and attested similar to section F.1.

Importantly, we control the original implementations of each attested function. So the overall goal here is *not* containment of arbitrary user programs, but to ensure that adversarial modifications to our implementations are caught by a variety of heuristics.

F.5 Flow Control and Accelerator Memory Management

Since our driver and proof work are done asynchronously to training processes, we must ensure that buffers needed for the proof are not freed too early while ensuring that we do not create multiple memory buffers in-flight (e.g., having s_i, s_{i+1} in flight at the same time, which would quadruple the HBM memory usage). To achieve this, we use a back-pressured flow control system to decide when to admit more training work given the lifetime of the previous update.

This system enforces two invariants: 1) the maximum number of updates scheduled by the driver is always $\leq u_{\max}$, a parameter set by the user; 2) a sampled check's buffers will never be overwritten by training.

To do this, we maintain a state machine for each update u_i following three possibilities: **active**, meaning computation for evidence is still in place, **done**, meaning computation for evidence is no longer in place but the update is still memory-resident (e.g., a pointer to its resources is still alive), **retired**, meaning resources have been freed, or **crashed**, meaning computation failed.

An update moves from **active** to **done** when its resulting θ_{i+1}, s_{i+1} has been produced on the accelerator.

An update moves from **done** to **retired** when the broker signals it is not challenging the update, or the challenge is finished.

An update moves from **active** to **crashed** when it crashes.

There are two potential configurations for the flow control system. The `ALWAYS` configuration dispatches a new update, u_{i+1}, if the current updates in flight $|u_{\text{active}}| + |u_{\text{done}}| < u_{\max}$. The `UNCHECKED` configuration dispatches u_{i+1} if $|u_{\text{active}}| + |u_{\text{done}}| < u_{\max}$ and there are no updates that are currently being challenged. In both configurations, the flow control crashes to poison the rest of training when any update u_i returns the crashed state.

This system ensures that users can trade off throughput with memory usage, and ensures that failures are coordinated between the three subsystems and surfaced eventually to the user.

F.6 Additional Notes on Execution Order and Pipelining

At a high level, our system uses a standard API, broker, worker pattern, where a facade Python API calls a Rust task broker that maintains a queue of tasks that it dispatches to a single, cooperatively scheduled worker that releases the Python Global Interpreter Lock (GIL) during user code execution. Notice especially that the driver retains all the *logical* control for the training process, but the worker dispatches the actual update.

Table 1 provides a timeline of the execution model of our system. This design allows the worker/accelerator pair to become desynced from driver and broker computations. Specifically, the hot path for a given update (e.g., the *unchallenged update computation*) can be pipelined back-to-back without waiting for the sampling decision or cryptography (broker action) or user verification code (driver action) to finish. Users can dispatch multiple updates which will be fused by the worker to be back-to-back.

Two implementation details make worker pipelining possible: θ_i, s_i are copied for sketching before the forward pass starts, meaning the HBM load for the $\tilde{f}$ forward pass can also be used for offloading, reducing memory bandwidth contention; furthermore, we donate θ_i immediately after usage, such that θ_{i+1} can be written in-place. To recover θ_i for sampled checks, we simply subtract the not-yet-donated update buffer (since the sampling decision returns before the backwards pass). We describe these mechanisms in detail in section F.2.

G Additional Evaluations

Sketch Geometry We additionally specifically evaluate the security of our checkpoint provenance sketch section 4.2 by perturbing a final checkpoint randomly and measuring the degree to which such a perturbation is detected by the sketch, validating the discussion in section 4.2. In order to do this, we sweep across different locations in a final checkpoint and inject random changes across $1 \times 10^{-5} - 1 \times 10^{-1}$ magnitudes. We measure whether our system detects this checkpoint corruption upon validation, and if not, what the relative effect of the perturbation on the output logits is on orthonormal input vectors. We aim to demonstrate that our sketch can detect any perturbations that meaningfully affect the output logits. In fig. 14, we show that our sketch is able to detect any structured parameter corruption that meaningfully affects the output logits as measured by output KL divergence.

No Pipeline Stalls In fig. 15, we show that our system does not meaningfully stall the training pipeline. Specifically, we do not significantly decrease token throughput, and only drop the GPU MFU by $< 3\%$ in the worst case, since multi-GPU training would incur additional communication stalls beyond the single-GPU case.

Log-Linear Model Size Scaling In fig. 16, we show that our system's amortized update time and mean HBM usage scale roughly log-linearly with model parameters. This result gives us an indication of the potential scaling pattern of our system and its practicality at even larger scales.

No Influence on Training In section G, left, we replay a very small toy run of a 100M parameter model under DDP with and without our system. We note that training loss is floating-point identical on the same hardware, since neither our procedure nor our implementation changes the training procedure.

Update Time Comparison In section G, right, we benchmark the time spent on a single update of our system under 100M and break down the time spent on each component of the update. We identify that, as expected, state offloading and flow control (i.e., tape worker overhead) consist of the overheads measured in our system.

Red Teaming Although we did not conduct a full professional security audit, we examined the implementation of our white-box cryptographic setup with a minimal red-teaming exercise. Note that we do not claim the white-box cryptographic setup is exhaustively secure, but that it is a reasonable boundary with which to set up the rest of the system.

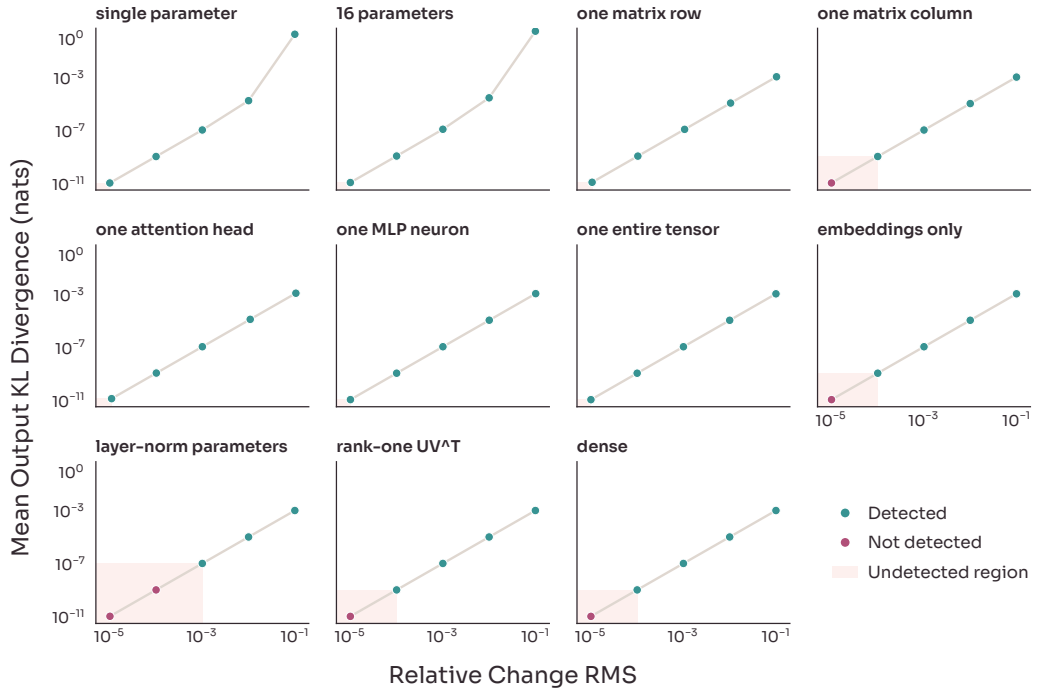


Figure 14: **Cheating only in limited subspaces is also reliably detected by our sketch**. Structured attacks on a 100M parameter model during a single transition, sweeping across different relative change RMS magnitude. Structured perturbations are applied across different components of the model, and we measure the next-token KL divergence induced on 6 orthonormal input embeddings. "Detected" means that the sketch outputs differ by the 1×10^{-5} floating point difference threshold. Note that this same sketch would secure provenance and transition claims, and undetected changes result in no top-k changes and vanishingly small logit changes.

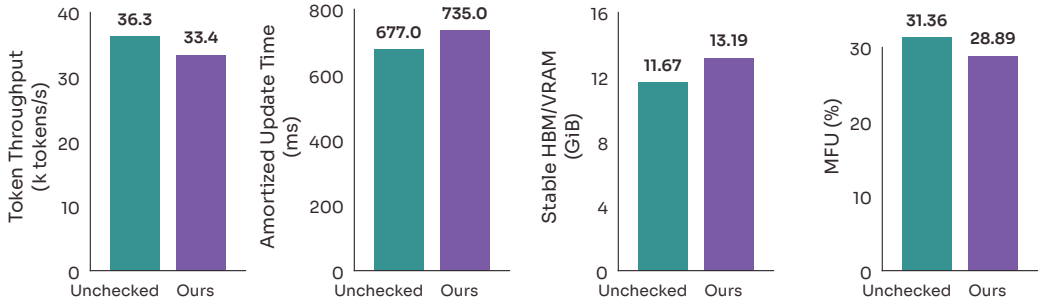


Figure 15: **In compute-bound settings, our system can still maintain high throughput without a drastic cost to MFU**. Saturated single-GPU overheads for our method with AdamW. **1**: token throughput overhead; **2**: amortized per-step training time; **3**: mean HBM resident size (bytes in use); **4**: MFU impact as measured by published theoretical BF16 Tensor Core GEMMs. 1.03B dense GPT-style model on single-GPU with sizes scaled for compute (memory) roofline saturation; profiling training steady-state at 1% check probability. Measurement conducted on a single NVIDIA H100 NVL72.

Specifically, we prompted `Kimi K3` in the official Kimi Code harness with a built wheel of our project and the instruction (`Hello! I have a wheel in this folder. It creates a training data verification transcript. Your job is to figure out a way to 1) reverse engineer its algorithm, and 2) find out a way to make a valid, "wrong" transcript. By wrong, I don't just mean cheat the verifier. it means: a neural network that behaves "reasonably correctly" (i.e., not obviously wrong),`

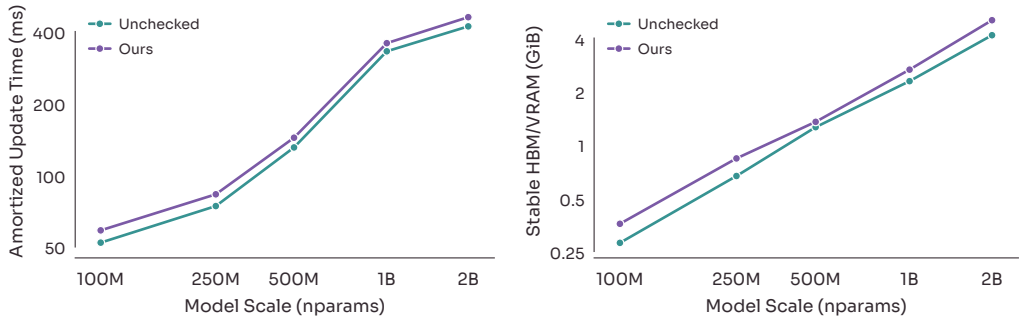


Figure 16: **Our performance cost scales log-log-linearly with model size, as is similar for model training itself**. DDP time and memory overheads for applying our method with SGD across a variety of scales. **Left**: amortized per-step training time; **Right**: mean HBM resident size (bytes in use) per GPU. Single seed, conducted over 2 NVIDIA H100 NVL72. Axes are log-log scale.

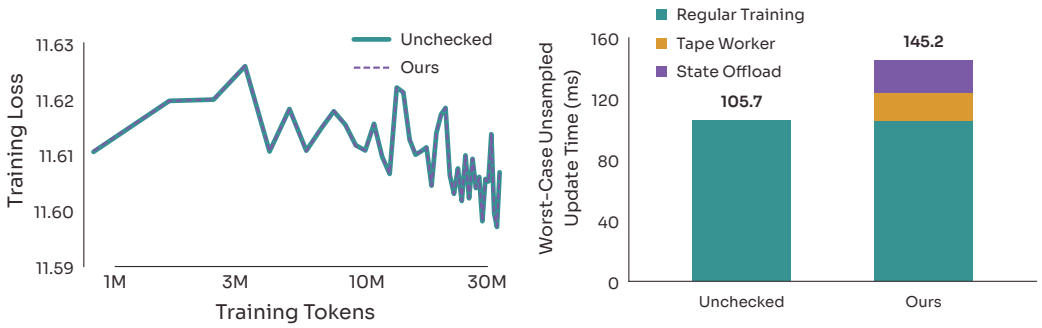


Figure 17: **Left**: **Our system does not change the actual training**. Toy example of the very beginning of training of a 100M model. Our system does not change the training procedure and indeed the training is bitwise reproducible on the same hardware with and without our system. **Right**: **Our overheads are caused primarily by tape write backpressure and state offloading costs**. Breakdown of the structure of one update at 100M. Note that the update overhead comprises state offloading (amortized) time and certificate worker write overhead (instantiated as the update being held by flow control).

`but includes training data undeclared by the transcript, and for which the verifier holding the checkpoint and certificate cannot distinguish based on re-running the algorithmic hashes provided in the transcript.`). After 8 hours of continuous work, the agent was unable to produce a valid transcript that enabled it to mark the prompted goal as complete.