Distributed ML Property Attestation Using TEEs

Idil Kara*
ikara@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Gavin Deane*
gdeane@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Artemiy Vishnyakov*
avishnyakov@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Abstract

As large machine learning (ML) providers adopt model cards to document how models are trained, the question becomes: how can a verifier be sure that a card is honest? Prior work such as Laminator shows how a trusted execution environment (TEE) can produce a proof-of-training (PoT) artifact for a *single* node, attesting that its output model was trained on a specific dataset, architecture, and configuration. Modern training pipelines, however, are distributed and data-parallel.

In this work we ask whether these single-node restrictions can be lifted to attest a distributed setting: *if each individual node can attest that it behaved correctly, can we safely conclude that the whole system behaved correctly?* Our key idea is to treat each worker as a Laminator-style prover and to run a coordinator inside a TEE that verifies worker PoT digests and aggregates their updates. Since the coordinator's code is itself remotely attested, an external verifier only needs to trust the coordinator enclave; the distributed training job then collapses to a single PoT artifact stating that, if every node followed its attested code, the final model was trained as claimed or else the artifact fails to verify.

We implement this protocol using PyTorch and a Docker-based TEE emulation, and evaluate it on data-parallel training over the CENSUS dataset. In our CPU-only prototype, attested runs incur a 2.2–3.1× slowdown (120–214% overhead) compared to an unattested baseline, with overhead scaling approximately linearly in the number of workers and epochs.

Keywords

Distributed Training, Non-repudiation, Model Provenance

1 Introduction

Every day, more and more ML models are being used for high-stakes applications such as résumé screening, medical diagnosis and autonomous driving [8]. However, the outputs of these models are governed by their training process and their training data [8]. Previous models have been shown to be racist, sexist, and otherwise unfairly biased due to the oversights of model providers [5, 8–10, 20, 27]. This has led to increased concerns about how these models are trained, and whether the model providers can be trusted [20].

Because of these concerns, many regulations have been created to make model providers accountable for their models [7, 11, 14, 18, 31]. To give transparency, some providers have embraced the use of model cards [24] and datasheets to disclose details about their models, which are then verified through a trusted entity.

For models trained and run inside trusted execution environments (TEEs) [23], it is possible to use remote attestation mechanisms such as Laminator [13] to prove, or attest, model properties to a verifier. This is tremendously useful to a model provider who wants to prove the authenticity of their model's behavior and training process to an auditor or prospective customer.

Distributed training is commonly used for training of large language models. A limitation of Laminator is that, due to the required isolation properties of TEEs [23], distributed training is not supported with property attestation. In this work, we address the question: **can we generate a verifiable proof-of-training artifact for data-parallel distributed training using TEEs, with acceptable overhead and security guarantees comparable to single-node systems such as Laminator?**

We claim the following contributions:

- A DISTRIBUTED ATTESTATION PROTOTYPE[1] for creating proof-of-training attestations with data-parallel training.
- A protocol to combine and attest the behavior of data-distributed ML worker nodes.
- An evaluation of our prototype showing that, compared to a baseline without attestation, proof-of-training overhead increases computation time by 120% to 214% depending on the number of trained epochs and distributed workers.

2 Background

In which we describe the current state-of-the-art for property attestation (Section 2.1), the Laminator framework (Section 2.2), and distributed ML (Section 2.3).

2.1 State-of-the-Art

Some prior approaches to verifying model properties include zero-knowledge proofs (ZKPs) [15], and secure multiparty computation (MPC) [12]. However, each of these approaches has limitations.

ZKPs support multiple verifiers, but are very overhead-intensive [15]. As an example, one such zero-knowledge proof of training protocol has ≈ 4200× computational overhead compared to training without the ZKP protocol. In addition, ZKPs only support properties that are specifically adapted to the ZKP scheme [15]. This is a prohibitive limitation when it comes to deployment in machine learning pipelines.

Secure multiparty computation has the severe limitation of requiring online interactions between provers and verifiers during verification. In addition, retraining the model is required for every verification [12]. This makes such a system nearly useless for model providers who want to verify model properties to many verifiers.

*All three authors contributed equally as first authors.

[1]Code: https://github.com/idilkara/laminator-distributed

Verifiable model property cards were introduced in Laminator [13], a framework that demonstrates binding claims to enclave measurements in TEEs using Intel SGX [19]. This work demonstrated that, given hardware with a TEE, almost any model property could be attested to by several provers and verified offline by several verifiers. However, Laminator is only demonstrated to work when the entirety of model training happens on a single CPU with a TEE. It does not consider vulnerabilities associated with passing data across an untrusted network during training. This severely limits utility in environments where model training is expected to be distributed.

A closely related framework is PraaS [4], a system allowing dataset owners to obtain *verifiable proofs* that their datasets satisfy certain properties by utilising TEEs such as Intel SGX [19]. The main difference between PraaS and Laminator is that PraaS focuses on provable verification of dataset properties [4], whereas Laminator supports provable attestations for datasets, proof of training, model properties, and inference [13]. Similarly to Laminator, PraaS is limited by the assumption that a proof must be generated in a TEE, thus preventing proofs from being parallelized across several CPUs/GPUs.

ExclaveFL [16] is a verifiable federated learning (FL) framework that uses TEEs to generate signed execution statements describing which code ran on which enclave and when. This provides end-to-end traceability for auditors, but *does not* produce a compact, verifiable attestation that binds the training data to the output model.

Also in the domain of FL security is SLAPP [28], an approach to prevent data poisoning in FL. SLAPP focuses on preventing malicious FL participants from poisoning data or bypassing differential privacy noise mechanisms by using stateful proofs of execution for each update. It does not provide a single attestation that the entire model was trained honestly.

A recent framework is Atlas [30], which is used to detect tampering by verifying the ML lifecycle. Atlas focuses on providing end-to-end model lifecycle transparency using provenance metadata and lineage metadata. Notably, Atlas assumes transparency and verification services run inside TEEs (such as Laminator or PraaS [4, 13]) are to be trusted. Atlas does not yet support distributed training across TEEs, and cites this as a possible extension in their future work [30].

Work has also been done on Attestation of Distributed Applications using TEEs [29]. This protocol collects, merges, and verifies attestation quotes from all participating components in a distributed system, allows for system-wide consistency checks, and is technology-agnostic [29]. However, it is not tested on distributed ML workloads.

To the best of our knowledge, there is no existing system that produces a verifiable ML Proof-of-Training artifact for a distributed training job that has low overhead and scalable verification.

2.2 Laminator Framework

TEEs [23] are useful for creating an environment isolated from the operating system where trusted applications (TAs) can run without fear of interference from a compromised operating system or attackers. Useful properties of TEEs for property attestation include isolation, secure storage, and remote attestation [23]. Isolation makes outside software unable to interfere with TA operation or read TA internal state. Secure storage means persistent storage available to TAs which cannot be interfered with by other software. Remote attestation means the TA running within the TEE can prove some aspect of its state to an external party.

Table 1: Summary of notations and their descriptions.

Notation	Description
$\mathcal{P}$	Prover
$\mathcal{V}$	Verifier
$\mathcal{M}_{ar}$	Model architecture
$\mathcal{T}$	Training configuration
$\mathcal{M}^{prov}$	Prover's model to be attested
$\mathcal{D}_{tr}^{prov}$	Prover's training dataset
$\mathcal{D}_{te}^{prov}$	Prover's test dataset
$\mathcal{D}_{adv}$	Forbidden training dataset
$h(\cdot)$	Cryptographic hash function

Laminator [13] uses Intel's Software Guard Extensions (SGX) [19] for its TEEs (or enclaves). SGX enclaves have the useful property where they can remotely attest their state by providing a quote. In addition, Laminator makes use of Gramine Library OS [2] to allow use of unmodified python scripts within an SGX enclave. This allows for anything calculable by a python script to be attested to using Laminator.

Proof of Training is one of the attestations described in Laminator [13]. It allows a prover $\mathcal{P}$ to attest to a validator $\mathcal{V}$ that a model $\mathcal{M}$ was trained on a dataset $\mathcal{D}_{tr}^{prov}$ using model architecture $\mathcal{M}_{ar}$ and training configuration $\mathcal{T}$. This Proof of Training ensures there is no bait-and-switch where a model provider states their model was trained on $\mathcal{D}_{tr}^{prov}$, when in fact it was trained on a disallowed dataset $\mathcal{D}_{adv}$. For simplicity, we re-use the notation introduced in Laminator with additions as needed (summarized in Table 1).

2.3 Distributed Training

Large ML models pose a problem where computation/memory demands grow too large for processing with a single GPU. To address this, several distributed ML training strategies have emerged. One such strategy is data parallelism, where different subsets of training data are ingested by the model on separate GPUs, then gradients are averaged between GPUs [26]. Other parallelism strategies include pipeline parallelism, tensor parallelism, and expert parallelism. In our work, we focus on data parallelism for creation of a proof-of-concept.

Distributed systems make property attestation more difficult in several ways. First, having multiple nodes means separate property attestations need to be computed for each. This creates a challenge on how to combine these attestations into a single aggregate attestation. In addition, attestations generated on subsequent epochs need to be built on previous epoch attestations to ensure the model was never switched out.

Some recent works on security guarantees for distributed machine learning include Atlas [30], ExclaveFL [16], and SLAPP [28], all of which create traceability for verifiers to check a model was trained correctly. These works all make use of TEEs to increase trustworthiness of nodes, and use a log or blockchain based setup to create traceability.

3 The Challenge

In which we detail the adversary model, we consider (Section 3.1), the challenge of aggregating outputs from nodes (Section 3.2), and the desired properties of an ideal solution (Section 3.3).

3.1 Adversary Model

Our core challenge is that, in a distributed setting, each node can be corrupted independently and all communication passes over an untrusted network. We therefore need a protocol that lets a verifier treat the whole distributed system as if it were a single honest node. That is, one can verify that if the distributed system was asked to run X, it cannot covertly run Y without detection.

A minimal trusted component such as a TEE is used to provide functions and tools that other processes and components can leverage to enhance their security. In this context the TEE acts as a root of trust.

The threat model used in Laminator is as follows:

- Adversary can be everywhere including Prover
- Code expected is correct and non-malicious
- Adversary cannot imitate a TEE's signature
- Verifier is trusted

However, the surface area of a distributed system is much larger. Thus, in addition to threats to Laminator, our distributed system must function while relying on insecure network communication, not trusting the OS, Network, Controller, or Workers.
Potential adversary capabilities are as follows:

(1) Dishonest Model Provider: If after the coordinator completes the training and generates the final model and proof of training attestation, an adversary modifies it before the verifier receives it, then the verifier will attempt to verify a tampered model or property card.
(2) Compromised host OS: The computers on which the workers or coordinator are running can be infected with malware that performs adversarial actions.
    - Controller node: The controller node can be infected outside the TEE to act maliciously by sending different training data than it is supposed to or by replacing the trained model with its own at any stage.
    - Byzantine worker: A worker can be compromised outside the TEE, modifying the training data, initial weights, or returned weights.
(3) Modified code within TEE: The controller or worker node code can be modified before the enclave is initialized to perform whatever actions the adversary wants inside the TEE.
(4) Adversary in Network Communication: There could be an adversary that eavesdrops on communication or acts as a Man in the Middle (MITM), modifying the transmitted data.

The threat windows exploited by the adversaries can be chronologically split into the following:

- **Before Training**: the worker or coordinator code can be modified to perform a malicious action; The initial model weights can be set to those of some existing model.
- **During Training**: Modification of Communication between Coordinator and Worker: If an adversary is within the Worker / Coordinator process / OS / network, then they can change any of the data being sent or replay a prior message.
- **After Training**: the coordinator can replace the final trained model sent along with the training attestation proof.
- **During Distribution**: A malicious model distributor can claim to be distributing the model attested by the ML property card but be sending their own.

3.2 Aggregation Problem

At each epoch the coordinator sends a signed message containing model and training configurations (architecture, initial model weights, training data, epoch number) as well as a shard of training data to workers. After training on the received data, workers return a signed message with hashes of their received inputs along with the trained model.

Laminator's [13] proof-of-training attestation gives whether a given output model was trained using expected $\mathcal{M}_{ar}$, $\mathcal{T}$, and $\mathcal{D}_{tr}^{prov}$. However, the assumption is that everything was trained securely within a TEE. The assumption for our design will be that workers act as Laminator nodes.

With every worker returning a proof of training at each step, we wish to find a way to combine them to generate a final attestation that all training was done correctly. A trivial solution is to keep a log of all inputs and outputs at every step, but the cost of verifying will be same as verifying each PoT generated at each epoch by each worker. Thus, a desired property is for the final attestation size and verification cost to be low compared to training time and cost. Methods used in prior work include append-only logs, Merkle trees [29], and incrementally verifiable computation schemes[3]. In our design, we will use the remote attestation properties of Laminator to avoid being forced to keep every log during training.

3.3 Desired Properties

We designed out system with practicality and correctness concerns in mind, hence this system should ideally have the following properties.

- **Soundness**: If our protocol's attestation artifact and model passes verification, then the final model **was** must have been trained with the expected $\mathcal{M}_{ar}$, $\mathcal{T}$, and $\mathcal{D}_{tr}^{prov}$, under the assumption that the verifier is trusted and TEEs are secure.
- **Completeness**: If the final model was trained with the expected $\mathcal{M}_{ar}$, $\mathcal{T}$, and $\mathcal{D}_{tr}^{prov}$, and the verifier is trusted and TEEs are secure, then our protocol's attestation artifact will pass verification.
- **Efficiency**: The overhead of the attested distributed training is low when compared to distributed training without attestation.

4 Design

We detail a formal algorithm (Section 4.1) to satisfy our (desired properties), specify how the system's components behave (Section

4.2), and describe how our design defends against the described adversary model (Section 4.3).

4.1 System Overview

Key parts of our system are pictured in Figure 1. The coordinator receives the training dataset, the training configurations, and the model architecture as the input. Then sends out the training dataset shards and the current model weights alongside the training configuration to the workers for them to compute one epoch of training. After receiving the results, the coordinator verifies the workers' attestations and aggregates the results received to update the model weights. Once the training process is completed, the coordinator sends out the final model alongside a model card containing the PoT attestation. Throughout the process a `Vrf` array is used to keep track of whether each step occurred correctly, and if something went wrong then the array shows which worker misbehaved at which epoch. A detailed algorithm can be found in Algorithm 1.

The training process can be split into three parts:

(1) **Initialization**:
The coordinator and workers initialize their TEE enclaves which makes their code unmodifiable. The coordinator establishes a secure communication channel with each worker by exchanging public keys and generating session nonce with them. The coordinator initializes the model and verifies the workers' code matches the expected hash. If code matches what's expected, then the code within the workers' TEEs can be trusted to perform the expected actions. Otherwise the coordinator writes to the `Vrf` array that the worker has bad code and excludes that worker from training.

(2) **Training Iteration**:
At each iteration, the coordinator sends to the workers the initial model weights, training configuration, and a dataset shard. The workers' hash their received input and train the received model on their respective received dataset shards and return a signed message containing the hashed input and trained weights. Then, the coordinator verifies that the signature came from the worker node, hashes match the sent inputs, and the response is for correct epoch. If there isn't a problem then the returned weights are aggregated to create the next iteration's initial model. If a worker's response is bad, that gets written to the `Vrf` array.

(3) **Post-Training and Verification**:
Coordinator generates the final model and PoT which includes the training configuration, model architecture, hash of the training dataset, and the `Vrf` array to indicate if a worker has misbehaved or given a bad result during training. A verifier can validate the model by checking the TEE status of the coordinator and the hashed code, and comparing to reference from trusted party such as the system administrator to ensure the training process was exactly what was claimed. Once the verifier trusts the coordinator, the verifier can check whether the model architecture, training configuration, and the dataset hash match what is expected by the system administrator. Finally, the verifier checks the verification flags to see if anything went wrong during training. If any of the components don't match with hashes/signatures or a verification flag is invalid then the attestation is invalid and model is rejected. Otherwise, the verifier knows the final model has been correctly trained following the claimed training configuration on the correct training data.

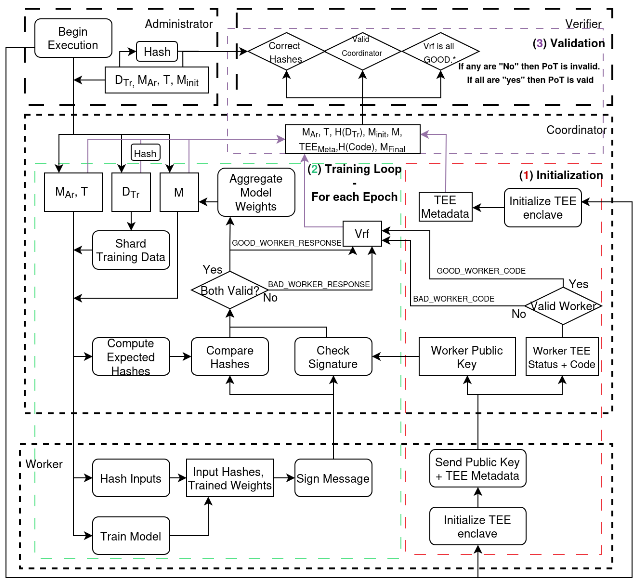

Figure 1: System Diagram

Table 2: Notation and descriptions used in the protocol.

Symbol	Description
$(\cdot)^e$	Epoch number.
$(\cdot)_{i,w}$	data slice i and worker ID w.
$\mathcal{D}^e_{Tr_w}$	Subset of training data for epoch e, worker w.
$\mathcal{M}_{Ar}$	Model architecture.
$\mathcal{M}^e_{\text{init}}$	Initial weights at epoch e.
$\mathcal{M}^e_w$	Trained weights returned by worker w.
$\mathcal{T}$	Training configuration (epochs, steps, seeds).
Vrf	Verification flags for worker responses.
$\mathcal{M}$	Final trained model.

4.2 Node Types

There are 2 types of nodes in the system: the coordinator node and worker nodes. The coordinator node distributes the training data to the worker nodes and aggregates the trained weights returned. The coordinator outputs the final model and the ML property card which will be verified externally. The worker nodes are modeled after the centralized Laminator[13] code to receive training data and model specifications and train the model on the training data, sending the resulting model weights back alongside a proof of training which is a reduced version of the ML property card.

Algorithm 1 Secure Distributed Training with TEE and Attestation. Legend in Table 2

```
Initialization
Coordinator broadcasts public key + metadata.
Workers respond with public keys and identity information.
Establish secure encrypted channel.
Coordinator verifies worker code attestation.
if code attestation invalid for worker $w$ then
    $Vrf[0][w] \leftarrow$ BAD_WORKER_CODE
end if
Coordinator samples initial model weights using seed from $\mathcal{T}$.
Coordinator initializes attestation state with:
  $H(\mathcal{M}_{Ar}),\ H(\mathcal{M}^0_{init}),\ H(T)$
  $Vrf[1:][:] \leftarrow$ INCOMPLETE_ATTESTATION.

Training Loop
for each epoch $e$ do
    Coordinator (inside TEE):
    Shuffle training data using seed.
    Allocate $\mathcal{D}^e_{Tr}$ between workers
    $seed \leftarrow PRG(seed)$
    Coordinator sends (encrypted and signed):
      $\mathcal{D}_{Tr_w},\ \mathcal{M}_{Ar},\ \mathcal{M}^e_{init},\ \mathcal{T},\ e.$
    Worker (inside TEE):
    Decrypt message.
    Compute input hashes $H(\mathcal{D}_{Tr_w}), H(\mathcal{M}_{Ar}), H(\mathcal{M}^e_{init}), H(\mathcal{T}), e.$
    Instantiate model with $(\mathcal{M}_{Ar}, \mathcal{M}^e_{init})$.
    Train model on $\mathcal{D}_{Tr_w}$ following configuration $\mathcal{T}$.
    Send back (encrypted and signed):
      $((H(\mathcal{D}_{Tr_i}), H(\mathcal{M}_{Ar}), H(\mathcal{M}^e_{\text{init}}), H(\mathcal{T}), e, \mathcal{M}^e_w)$.
    Coordinator verifies:
    Validate signature, hashes, and epoch number.
    if any check fails then
        $Vrf[e][w] \leftarrow$ BAD_WORKER_RESPONSE
    else
        $Vrf[e][w] \leftarrow$ GOOD_WORKER_RESPONSE
    end if
    Aggregate all good worker models:
      $\mathcal{M}^{e+1}_{init} \leftarrow$ Avg( $\mathcal{M}^e_w\ \forall w | Vrf[e][w] =$GOOD_WORKER_RESPONSE).
end for

Post-Training Attestation
Coordinator generates final attestation including:
  $\mathcal{M}_{Ar},\ T\ H(\mathcal{D}_{Tr})$
  $H(CODE_{coord})$ (TEE static code hash)
  $Vrf$

External Verification
Verifier checks coordinator TEE status.
Validate $H(CODE_{coord})$ against trusted reference.
Validate dataset hash $H(\mathcal{D}_{Tr})$ to reference from trusted data source.
Validate configuration $\mathcal{M}_{Ar}, \mathcal{T}$ to reference from administrator.
Ensure all flags $Vrf$ = GOOD_WORKER_RESPONSE.
Produce final model card.
```

4.3 Addressing Vulnerabilities

(1) **Dishonest Model Provider**: The authenticity of a downloaded model can be verified using the hash of the model provided in the attestation; thus it will be extremely difficult (assumed impossible) for the adversary to find a model with the same hash as the authentic model.

(2) **Modification of Communication from Coordinator to Worker**: A compromised controller node, network adversary, or compromised worker node can modify the data sent from coordinator to worker. In this case the worker hashes the modified inputs received within its TEE and trains on them, then returns the resulting model. Hashing and hash checking is done within the controller's TEE thus any tampering with the data will be detected due to the hashes not matching or the signature being broken a posteriori.

(3) **Modification of Communication from Worker to Coordinator**: A compromised controller node, network adversary, or compromised worker node can modify the data sent from worker to controller. In this case the worker signs the message containing the hashes and model within its TEE then sends to coordinator. The coordinator verifies the signature before processing the data. If the message is modified the signature will not be valid and the message can be flagged as tampered with.

(4) **Replay Attack**: If a network or worker adversary re-send a valid message from a previous epoch then the e value will not match and coordinator will discard. As is the case if coordinator message is replayed and worker generates a response.

(5) **Modified code files**: Worker or Coordinator code can be modified. If it is after enclave initialization then the TEE will not allow it, if it is before then the enclave will be initialized with the malicious code. Coordinator knows what code the worker must have and checks that the workers' enclaves are initialized with correct worker code using hash of code. Verifier knows the Coordinator code and performs the code hash verification.

The trust is extended from administrator/verifier to coordinator to worker and misbehavior is caught in the same order. Coordinator confirms that workers act correctly and verifier checks that coordinator executes correctly.

5 Evaluation

In which we detail our physical implementation and constraints (Section 5.1), show empirical evidence that our design correctly thwarts attacks described in the adversary model (Section 5.2), and compare performance with a baseline data-distributed setup without our framework (Section 5.3)

To re-state, the goals for our design were to show that the protocol works to attest to data-distributed ML runs without tampering, and show that the protocol rejects tampered-with runs. We also desired to demonstrate a system that is efficient in terms of introduced attestation overhead.

5.1 Hardware and Implementation

Our prototype is implemented using PyTorch for data-parallel training. To mimic a cluster of trusted nodes with TEEs while allowing for reproducible experiments, we run a CPU-only docker setup on a Lenovo Legion 5 16IRX9 Windows PC with 32 GB RAM and an Intel Core i7-14650HX CPU. Training was done using a CPU to reduce complexity in comparing with the single-CPU Laminator system.

Due to challenges in testing with SGX-enabled CPUs, to simulate the TEEs with docker containers, we decided to simulate remote attestation through container specific (RSA) digital signatures instead. Moreover, we mimic a cluster of trusted nodes using a single x86_64 Linux server in a CPU-only configuration and use Docker containers to emulate a coordinator plus multiple worker nodes.

In this setup, each worker container runs one instance of the training script, and the coordinator container is responsible for dataset sharding, sending out tasks to the workers, performing model aggregation, and attestation aggregation at each step. To emulate Laminator's [13] attestation scheme, each container has a generated software key pair on startup, then signs its inputs and outputs at each step in the process. The input for attestation such as the training data and the model configurations is hashed in the coordinator code as received as input, simulating the input being hashed as it is loaded to TEE. Likewise, the worker node, as it receives the message, hashes the inputs then runs the training algorithm on the model specified by in the massage and training data received, signing the input hashes and resulting model before sending back to coordinator.

These digest + signature packages act analogously to attestations generated in Laminator for the purposes of our implementation. That is, the coordinator and the external verifier check these packages in the same way they would check SGX quotes were TEEs implemented. This approach has drawbacks for performance evaluation, such as not properly capturing enclave-specific costs. These limitations are discussed further in Section 6.3.

Datasets and models. The datasets used to evaluate our framework were drawn from the tabular dataset CENSUS [22], which contains 48842 records each with 14 attributes from the 1994 US Census database, the goal being to predict whether an individual has annual income > \$50,000 USD.

The models we train on this dataset are as follows:

- **CENSUS-S**: Small MLP model with one hidden layer of dimension [128], and
- **CENSUS-L**: Larger MLP model with four hidden layers of dimension [128, 256, 512, 256] respectively.

We use mini-batch gradient descent with a fixed batch size and a fixed learning rate. We use synchronous data parallelism with $W \in \{2, 4, 8\}$ workers and sweep the number of epochs $E \in \{10, 50, 100\}$. To test scalability, we vary the number of workers and epochs.

TEE-style attestation logic. In our implementation, each worker signs a digest consisting of hashes of the model architecture $\mathcal{M}_{ar}$, training configuration $\mathcal{T}$, its current data shard to train on, the epoch counters, and the updated model parameters (see Section 4.1). The coordinator verifies worker signatures, aggregates worker models by averaging parameters, and maintains an append-only log of per-step digests. At the end of training, the coordinator produces a single proof-of-training artifact consisting of:

1. Hashes of $\mathcal{M}_{ar}$, $\mathcal{T}$, and the full training dataset $\mathcal{D}_{tr}^{prov}$;
2. A final attestation artifact which states whether the training run was correctly executed without tampering using $\mathcal{M}_{ar}$, $\mathcal{T}$, and $\mathcal{D}_{tr}^{prov}$.
3. Hash of coordinator code and static data to ensure the coordinator code that was run matches the coordinator code the verifier expects.

To verify, the external verifier re-executes the hashes and checks signatures to validate the artifact for the distributed distributed run. The reason this can be trusted without a chain of artifacts is because, if the checks run in the coordinator were removed, these changes would be reflected in the enclaves' signature in a system using TEEs.

5.2 Correctness Demonstrated

To show that ML property attestation can be done in a distributed environment using data parallelism, we must show that any honest execution produces a successful verification and and dishonest execution fails to verify. In addition to the explanation provided in Section 4.4, we conduct empirical tests using our implementation to demonstrate this behavior.

Honest runs. In the honest configuration, all workers are initialized with the same architecture $\mathcal{M}_{ar}$ and configuration $\mathcal{T}$, the coordinator partitions the expected dataset $\mathcal{D}_{tr}^{prov}$ across workers, and all nodes execute the unmodified attested training code. For each of three independent runs we record:

- the final trained model parameters;
- the coordinator's final attestation artifact; and
- the verifier's decision.

In all honest runs, the verifier successfully checks the coordinator's output, recomputes the dataset hash from a trusted copy of CENSUS-S, and validates. These runs demonstrate that our aggregation logic does not introduce false negatives. That is, as long as every worker behaves honestly and the coordinator's code matches the expected measurement, the verifier accepts.

Tampered runs. We then test four adversarial scenarios as follows:

1. **Model replacement.** After training, the final model file on disk is replaced with a randomly initialized model while leaving the attestation artifact unchanged. At verification time, the model hash derived from the supplied model no longer matches the hash encoded in the artifact, and the verifier rejects. This demonstrates protection against a model bait-and-switch.
2. **Wrong dataset.** We modify one worker's configuration so that it trains on an alternate dataset while still producing signatures. The worker's digest now contains a different dataset hash; as a result, the signature sent to the coordinator now differs from what coordinator should expect, had the worker correctly trained on $\mathcal{D}_{tr}^{prov}$. The coordinator flags the run as being tampered-with, and the verifier rejects. This captures attempts by a worker to substitute training data.

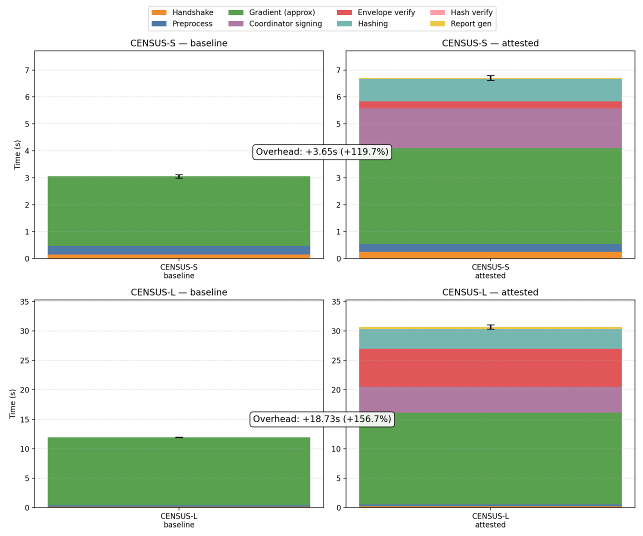

Figure 2: Pipeline breakdown for CENSUS-S and CENSUS-L with $W = 2$ workers and $E = 10$ epochs. Bars show mean over 5 runs; error bars are one standard deviation. Overhead stickers state attested runs' relative time increase over the baseline. Only coordinator-side attestation overhead is separately reported, meaning worker-side attestation overhead is contained in Gradient (approx)

(3) **Random worker model.** We modify a worker so that it discards the trained parameters and instead returns random weights, still signing the digest with the **original** weights. Because the digest binds the trained parameters, the signature sent to the coordinator no longer matches the model weights sent with it. The coordinator the run as tampered, and the verifier rejects. This demonstrates that a malicious worker cannot inject arbitrary updates without detection.

(4) **Controller replay.** We simulate a malicious coordinator (compromised OS) that reuses an old per-step worker proof in a later epoch. The monotone counter and per-step indices embedded in the worker digests now conflict with the global epoch/step sequence encoded in the coordinator's log. In this case, one of two things happen. (1) The TEE-run code is unmodified, and flags the run as tampered with, or (2), the TEE-run code is modified, and the enclave signature no longer matches that expected by the verifier. In either case, the verifier detects that something went wrong and rejects the proof.

Across all tampered runs, the verifier rejects the attestation artifact, either because the coordinator enclave code is unmodified and correctly reports that something was modified, or because the coordinator enclave code is modified and the enclave signature no longer matches what the verifier expects. These results support our claim that ML property attestation can be done in a data-distributed ML pipeline.

5.3 Comparison with Baseline

We now determine the overhead of distributed property attestation compared to the cost of training. This is done by comparing our attested pipeline against a baseline data-parallel training setup without any attestation logic. The assumption for both the baseline and attested runs is that payloads sent are encrypted, so encryption overhead is present in both.

For both the CENSUS-S and CENSUS-L models we measure:

- handshake time (node initialization and key exchange);
- preprocessing time (loading and preprocessing the dataset);
- pure training time (forward + backward passes and parameter averaging);
- attestation generation time (hashing and signing steps inside the training loop, including envelope/hash verification and coordinator signing); and
- report generation time (serializing the final proof-of-training artifact).

All reported timings in this section are the mean over $N = 5$ runs; error bars in the figures show one standard deviation.

5.3.1 *Performance Overhead (Single Configuration).*

Our first performance experiment fixes the number of workers to $W = 2$ and the number of training epochs to $E = 10$ and compares:

- a data-parallel baseline without attestation, and
- the same configuration with our attested training pipeline enabled.

We repeat this for both CENSUS-S and CENSUS-L.

Figure 2 shows a breakdown of the pipeline time for these four runs. In both cases, we can see that the attestation incurs significant overhead compared to the non-attested baseline, taking between 2.2× and 2.6× as long.

The stacked bars make it clear that the extra time comes mostly from attestation work **in the coordinator**. The most dominant coordinator-side work includes:

- **Signing**: time the coordinator spends signing outbound tasks for workers
- **Verifying**: time the coordinator spends verifying incoming worker response signatures and digests
- **Hashing**: time the coordinator spends hashing inputs, outputs, and tasks.

The remaining increase in Gradient (approx) comes from worker-side attestation overhead.

5.3.2 *Scalability Across Workers and Epochs.*

To study how attestation time scales, we sweep the number of workers $W \in \{2, 4, 8\}$ and epochs $E \in \{10, 50, 100\}$ for the CENSUS-S model. For each (W, E) pair, we measure the total training pipeline time with and without attestation, again using the same components as above.

Formally, letting $T^{\mathrm{B}}_{W,E}$ and $T^{\mathrm{A}}_{W,E}$ denote the times for baseline and attested runs respectively, we compute

$$\text{Overhead}_{W,E}(\%) = \frac{T^{\mathrm{A}}_{W,E} - T^{\mathrm{B}}_{W,E}}{T^{\mathrm{B}}_{W,E}} \times 100.$$

Figure 3 summarizes these results. Each subgraph shows one (W, E) configuration with a baseline bar (left) and attested bar (right), and a sticker reporting the absolute and relative increase.

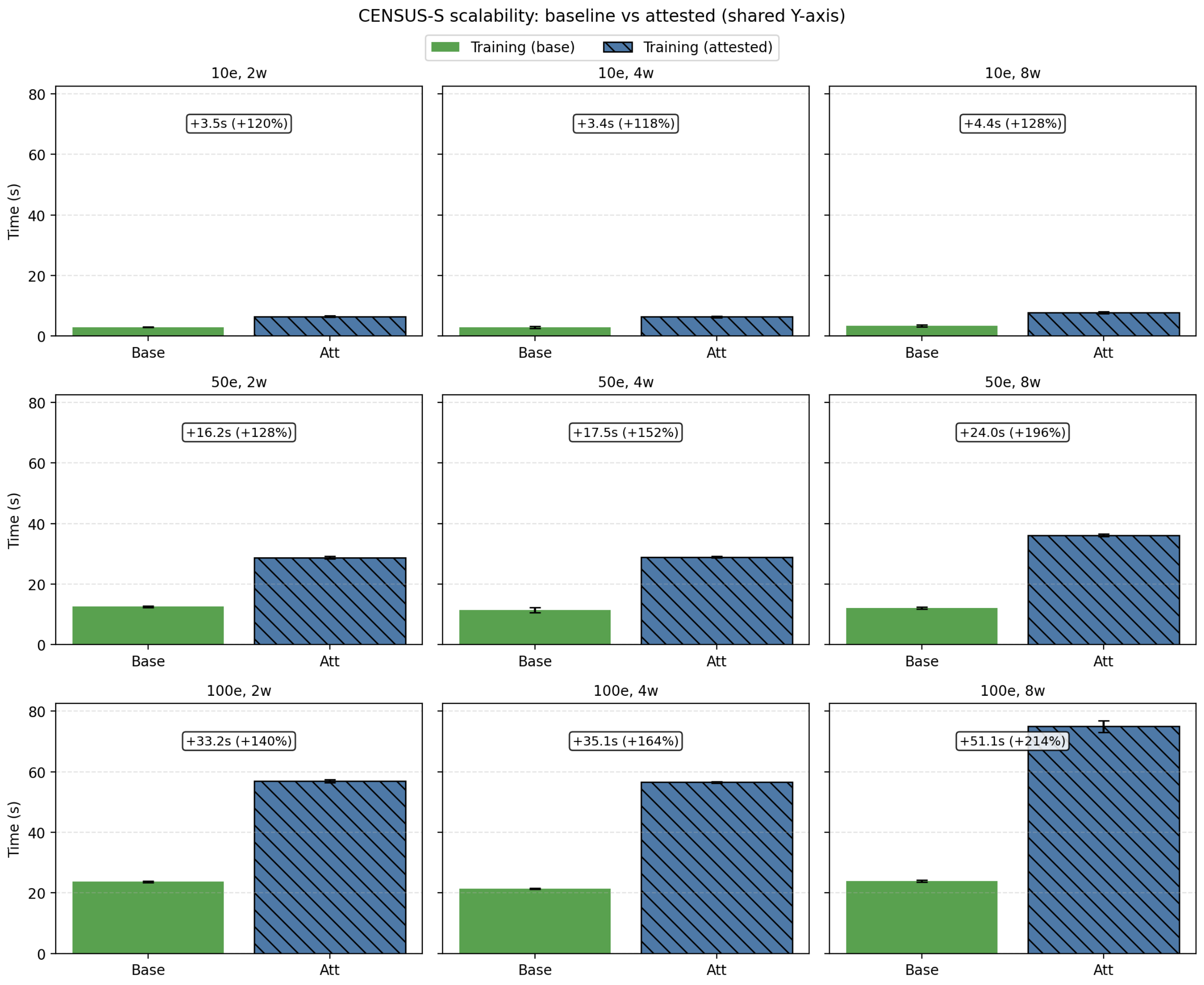

Figure 3: CENSUS-S scalability for baseline vs. attested training across workers $W \in \{2, 4, 8\}$ and epochs $E \in \{10, 50, 100\}$. Bars show mean training pipeline time over 5 runs (one standard deviation error bars). Overhead stickers denote relative increase of the attested pipeline over the baseline for each configuration.

From these results, it is evident that there is a roughly linear correlation between attestation overhead and the number of workers, as well as a roughly linear correlation between attestation overhead and the number of epochs. This matches the implementation, as attestation logic in the implementation is performed on a per-worker, per-epoch basis.

Overall, these experiments show that our prototype incurs a substantial constant-factor slowdown relative to an unattested baseline, and that this slowdown grows with the number of workers because each additional worker incurs its own per-epoch hashing and signing overhead. We discuss what this implies for a real TEEs-based deployment, and how it might be optimized, in Section 6.5.

5.3.3 Memory Overhead. We quantify memory overhead in terms of the difference in size between the output artifacts for baseline and attested runs. That is, the model alone for the baseline versus the model plus proof-of-training artifact for our framework. Table 3 reports both absolute and relative memory overhead. As discussed in Section 6.3, these numbers reflect our software-based emulation of Laminator-style attestations rather than true SGX quote sizes, but they still provide an estimate of the artifact size a verifier would need to store.

Table 3: CENSUS-S memory usage of final weights and attestation artifacts for different scalability configurations. Sizes are in KB (1 KB = 1024 bytes). Relative overhead is $\Delta_{\text{mem}}/|$`final_weights`$|$.

E	W	hash_report (KB)	report_sig (KB)	final_weights (KB)	Δ_{mem} (KB)	$\Delta_{\text{mem}}/$|final_weights| (%)
10	2	262.4	0.6	249.5	13.6	5.4%
10	4	262.8	0.6	249.5	13.9	5.6%
10	8	263.3	0.6	249.5	14.4	5.8%
50	2	264.2	0.6	249.6	15.2	6.1%
50	4	265.7	0.6	249.6	16.6	6.7%
50	8	268.6	0.6	249.6	19.6	7.8%
100	2	266.2	0.6	249.6	17.2	6.9%
100	4	269.1	0.6	249.6	20.1	8.1%
100	8	274.9	0.6	249.6	26.0	10.4%

6 Discussion

6.1 Explicit Assumptions & Security Statement

To re-iterate, this is a protocol designed to be cryptographically self-authenticating so that, no matter what an attacker does outside of TEEs, any deviation leaves a provable inconsistency for the verifier to check.

That is, we work under the following explicit assumptions:

- The external verifier is not compromised
- The external verifier knows the hashes of expected inputs ($\mathcal{M}_{ar}$, $\mathcal{D}_{tr}^{prov}$, $\mathcal{T}$, and expected TEE code) to the system.
- The adversary is not able to forge TEE attestations, except with negligible probability.
- Based on the properties of TEEs, any deviation in code run within a TEE will be reflected in the generated attestation from that TEE.

Given all of these assumptions hold, we posit that this system prevents a dishonest model provider from lying to a trusted verifier without detection.

6.2 Out of Scope Threats

This protocol is purely concerned with preventing an attacker from modifying a distributed training pipeline without detection. We do not consider threats to confidentiality such as attackers gaining unauthorized access to datasets or model architecture. In fact, we assume for strength of our argument that the attacker knows all of this and can modify it as it pleases.

Despite its similarity, we are not concerned with preventing an attacker from tampering with the machine learning pipeline. We simply care whether or not such tampering can be guaranteed to be detected by an external verifier, and by extension, proving that a model that displays no such tampering under our protocol is, in-fact, trustworthy.

Despite previous attacks showing this is possible, we do not address the possibility of side-channel attacks against TEEs [17, 25]. The reason this is out of scope for this work is because this would allow adversaries to feasibly forge TEE attestations, which contradicts our assumptions about the adversary model. Additionally, it is a pointless exercise to design a security protocol with the assumption that the root of trust will fail.

6.3 Limitations of Laminator Abstraction

Our work was limited by access to CPUs with enclaves, this led to the abstraction of simply assuming a hash of inputs was equivalent to a proper TEE abstraction. This is less of a problem for theoretical security than it is for realistic performance evaluation.

We approximate the main performance differences between actual TEE usage and our abstraction are as follows:

- **No TEE loading**: The implementation does not account for any of the overhead associated with loading tensors into a TEE.
- **No additional optimization**: The implmentation does not include any of the optimizations that Intel SGX [19] includes, instead using simple cryptographic primitives for hashing, signing, and verifying.
- **Only one CPU**: The implementation runs on a single CPU device and simulates multiple CPU nodes as seperate docker containers. This simultaneously gives the benefit of there being no network latency between the "CPU"s, and the downside of not having parallelism across real CPUs.

6.4 Performance limitations

The chief performance limitation in this work is the lack of using GPUs. GPUs are commonly known to have significantly greater performance for matrix multiplication tasks and have been empirically shown to be orders of magnitude faster than CPUs for these tasks[6]. This work didn't use GPUs primarily because previously existing work for doing TEE-assisted ML proofs of training only makes use of CPUs, and using GPU TEEs for attesting ML is a different line of research from attesting distributed ML.

Given these limitations, our performance evaluations focuses less on the actual time of the proof and more on the comparative overhead between a system running without attestation versus with attestation.

6.5 Notable Evaluation Results

In our evaluation we found remote attestation overhead scales roughly linearly (or affine) with the number of workers.

This linear scaling with the number of workers highlights a challenge of attesting a distributed framework as opposed to a single-node framework like Laminator [13]. In Laminator (or any single-TEE proof of training system), a single check can be done on input to a TEE, and then a single attestation can be securely generated after many epochs of training have been securely performed. In contrast, a distributed system must have its inputs and outputs checked and signed whenever data enters a TEE from the untrusted network.

As far as we can tell, there is no way to securely attest a distributed system without having some overhead that scales with the number of workers, but there are methods to drastically reduce the overhead of repeated attestation[21]. For instance, RA-TLS merges SGX remote attestation into the TLS handshake so that enclave identity is verified once, and all subsequent communication is protected efficiently using symmetric cryptography—fully amortizing the cost of attestation. This seems like a promising direction to avoid repeating heavy cryptographic operations for our protocol.

6.6 Theoretical O(1) Verification-Time Framework

An interesting idea came up during discussion of how to implement efficient distributed attestation. In short, the protocol we came up with in this work depends on using the property that a remote verifier can attest to what code was run inside a TEE. In our work, we use this property to make the verifier responsible for verifying the coordinator, and the coordinator responsible for verifying the workers. If at any point the coordinator code changes unexpectedly, the verifier becomes aware when it receives a payload that was not generated by the expected code.

In our implementation, the coordinator sends the verifier an array including all the events that happened in the ML pipeline for the verifier to check. However, if it's the case that the verifier can outsource trust to the coordinator because of the code attestation property described above, then the verifier does not actually need to check all the ML pipeline events. That is, the coordinator could be wholly responsible for checking all of the events were as expected, and simply send the verification a boolean value of "yes", everything went well, or "no", something went wrong.

Of course, there is an implicit trade-off here. The verifier might be interested in knowing more about the pipeline or want to know exactly what went wrong. In this case, such a protocol would have to be tuned so the output artifact includes the appropriate amount of information.

Regardless, the theoretical O(1) verification-time framework seems reasonable, assuming the verifier and the TEE can be trusted.

7 Future Work

This work resulted in a protocol for attesting data-distributed ML and a prototype that abstracts away many hardware details. This leaves much work to be done before this protocol could be applied to a real-world system. Several such directions of work are outlined here.

The most important thing that can be done as future work is adapting the protocol and evaluating with real TEE-enabled CPUs. Restrictions accessing hardware limited the realism of our evaluation. This system must be tested in a less abstract environment before further innovations are done.

The second most important step is to figure out how to reduce or amortize the overhead that comes from repeated attestations being done per-worker and per-epoch.

This work only addresses proof-of-training attestations for data parallelism across several CPU nodes. This creates room for many different extensions.

First, the other attestations described in Laminator [13] could be implemented and profiled for performance in the distributed training setting. These include attesting distributional properties of datasets, evaluating model accuracy/fairness, and attesting a specified degree of robustness against adversarial examples.

Another direction of work is to implement and profile this system using GPU nodes as opposed to CPU nodes. GPUs are widely used for ML due to their comparative speedup with parallel computing. TEE-enabled GPUs such as Nvidia's H100 have also been recently released [1], allowing the use of GPU-accelerated computation inside of TEEs. Modifying this framework to use GPUs instead of CPUs would be a logical performance optimization to bring verifiable proofs of training closer to adoption with mainstream ML training.

In this work, only data parallelism was used to support the proof-of-concept. To the best of our knowledge, neither pipeline parallelism nor tensor parallelism have been explored [26] when it comes to proofs of training for machine learning. This is a compelling direction to pursue, considering the increasing use of these techniques in modern model training.

The Atlas framework [30] focuses on providing end-to-end model lifecycle transparency, but does not yet support distributed training across nodes. Our work could be polished, then integrated to support attestation for data parallel training across multiple nodes within Atlas.

Attestation of accuracy, bias, robustness, and fairness can be done by coordinator following the same methodology as Laminator [13]. Attestation of data distribution can be done by the coordinator or by splitting the work between workers.

8 Different Author Contributions

Within this work, Artemiy Vishnyakov was responsible for formalizing the algorithm. Idil Kara was responsible for setting up and validating the distributed ML baseline and proof of concept implementation. Gavin Deane was responsible for the literature review, designing the structure of the paper, and experimental design.

Acknowledgments

To Adam Caufield, for explaining many details of the simulated hardware setup, listening to our ideas during team stand-ups, and giving so much feedback. We also thank Vasisht Duddu, for giving advice on preliminary designs and Laminator's hardware constraints.

References

[1] [n. d.]. NVIDIA H100 GPU Datasheet. https://nvdam.widen.net/s/fdllbtmmbv/h100-datasheet-2430615

[2] 2025. Gramine. https://gramineproject.io/. Accessed: Oct. 01, 2025.

[3] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. 2024. Zero-Knowledge Proofs of Training for Deep Neural Networks. Cryptology ePrint Archive, Paper 2024/162. https://eprint.iacr.org/2024/162

[4] Istemi Ekin Akkus, Ivica Rimac, and Ruichuan Chen. 2024. PraaS: Verifiable Proofs of Property as-a-Service with Intel SGX. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 199–207. doi:10.1109/EuroSPW61312.2024.00027 ISSN: 2768-0657.

[5] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine Bias. *ProPublica* (May 2016). https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing Accessed: May 23, 2016.

[6] Mufakir Qamar Ansari and Mudabir Qamar Ansari. 2025. Accelerating Matrix Multiplication: A Performance Comparison Between Multi-Core CPU and GPU. doi:10.48550/arXiv.2507.19723 arXiv:2507.19723 [cs].

[7] M. C. Buiten. 2019. Towards Intelligent Regulation of Artificial Intelligence. *European Journal of Risk Regulation* 10, 1 (March 2019), 41–59. doi:10.1017/err.2019.8

[8] Brian Christian. 2021. *The Alignment Problem: How Can Machines Learn Human Values?* Atlantic Books.

[9] Kate Crawford. 2021. *The Atlas of AI.* Yale University Press.

[10] Jeffrey Dastin. 2018. Amazon scraps secret AI recruiting tool that showed bias against women. https://www.reuters.com.

[11] Iñigo de Miguel Beriain et al. 2022. *Auditing the quality of datasets used in algorithmic decision-making systems.* Technical Report PE 729.541. European Parliamentary Research Service.

[12] Vasisht Duddu, Anudeep Das, Nora Khayata, Hossein Yalame, Thomas Schneider, and N. Asokan. 2024. Attesting Distributional Properties of Training Data for Machine Learning. In *Computer Security – ESORICS 2024*, Joaquin Garcia-Alfaro, Rafał Kozik, Michał Choraś, and Sokratis Katsikas (Eds.). Springer Nature Switzerland, Cham, 3–23. doi:10.1007/978-3-031-70879-4_1

[13] V. Duddu, O. Järvinen, L. J. Gunn, and N. Asokan. 2025. Laminator: Verifiable ML Property Cards using Hardware-assisted Attestations. *arXiv preprint arXiv:2406.17548* (March 2025). doi:10.48550/arXiv.2406.17548

[14] European Commission. 2024. Regulation (EU) 2024/1689 (Artificial Intelligence Act). https://artificialintelligenceact.eu/.

[15] Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. 2023. Experimenting with Zero-Knowledge Proofs of Training. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1880–1894. doi:10.1145/3576915.3623202

[16] Jinnan Guo, Kapil Vaswani, Andrew Paverd, and Peter Pietzuch. 2025. VerifiableFL: Verifiable Claims for Federated Learning using Exclaves. doi:10.48550/arXiv.2412.10537 arXiv:2412.10537 [cs].

[17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3065913.3065915

[18] High-Level Expert Group on AI. 2019. Ethics Guidelines for Trustworthy AI. https://ec.europa.eu/digital-single-market/en/news/ethics-guidelines-trustworthy-ai

[19] Intel. 2014. *Intel Software Guard Extensions Programming Reference*.

[20] Davinder Kaur, Suleyman Uslu, R. J., and Arjan Durresi. 2022. Trustworthy Artificial Intelligence: A Review. *ACM Computing Surveys (CSUR)* (Jan. 2022). doi:10.1145/3491209

[21] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2019. Integrating Remote Attestation with Transport Layer Security. arXiv:1801.05863 [cs.CR] https://arxiv.org/abs/1801.05863

[22] Ron Kohavi. 1996. Census Income. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5GP7S.

[23] Kari Kostiainen et al. 2010. Key Attestation from Trusted Execution Environments. In *Proceedings of TRUST*. doi:10.1007/978-3-642-13869-0_3

[24] Margaret Mitchell et al. 2019. Model Cards for Model Reporting. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency (FAccT)*. ACM, Atlanta, GA, USA, 220–229. doi:10.1145/3287560.3287596

[25] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A Survey of Published Attacks on Intel SGX. doi:10.48550/arXiv.2006.13598 arXiv:2006.13598 [cs].

[26] OpenAI. 2025. Techniques for training large neural networks. https://openai.com/index/techniques-for-training-large-neural-networks/. Accessed: Sep. 22, 2025.

[27] S. Perez. 2016. Microsoft silences its new A.I. bot Tay, after Twitter users teach it racism [Updated]. https://techcrunch.com/2016/03/24/microsoft-silences-its-new-a-i-bot-tay-after-twitter-users-teach-it-racism/. Accessed: Oct. 01, 2025.

[28] Norrathep Rattanavipanon and Ivan De Oliveira Nunes. 2025. Poisoning Prevention in Federated Learning and Differential Privacy via Stateful Proofs of Execution. doi:10.48550/arXiv.2404.06721 arXiv:2404.06721 [cs].

[29] Ghazal Shenavar. 2025. Attestation of Distributed Applications. (July 2025). https://aaltodoc.aalto.fi/handle/123456789/138160

[30] Marcin Spoczynski, Marcela S. Melara, and Sebastian Szyller. 2025. Atlas: A Framework for ML Lifecycle Provenance & Transparency. doi:10.48550/arXiv.2502.19567 arXiv:2502.19567 [cs].

[31] U.S. Congress. 2022. H.R. 6580 – Algorithmic Accountability Act. https://www.congress.gov/bill/117th-congress/house-bill/6580/text.