ZKBoost: Zero-Knowledge Verifiable Training for XGBoost

Nikolas Melissaris[1], Antigoni Polychroniadou[2], Akira Takahashi[2], Chenkai Weng[3], and Jiayi Xu[3]

[1]CNRS, IRIF, Université Paris Cité
[2]JPMorgan AI Research & AlgoCRYPT CoE
[3]Arizona State University

Abstract

Gradient boosted decision trees, particularly XGBoost, are among the most effective methods for tabular data. As deployment in sensitive settings increases, cryptographic guarantees of model integrity become essential. We present ZKBoost, the first zero-knowledge proof of training (zkPoT) protocol for XGBoost, enabling model owners to prove correct training on a committed dataset without revealing data or model parameters. Naively re-executing XGBoost training in ZK would incur prohibitive costs, primarily due to the oblivious partitioning of training samples and unknown tree splits. Moreover, previous work on ZKP of training and inference had subtle security issues, such as leakage of tree topology and soundness gaps allowing cheating model providers to deviate from the correct execution of training and inference. We make two key contributions to address these challenges: (1) a generic zkPoT template for XGBoost that can be instantiated with any general-purpose ZKP backend, significantly improving prover costs compared to naive re-execution of the training process; and (2) a VOLE-based instantiation that overcomes the security issues of previous ZK proofs of training at minimal costs. To maximize efficiency, we develop a fixed-point version of XGBoost, which is particularly well suited for efficient instantiation of ZKP, and show it matches standard XGBoost accuracy to within 1% on real-world datasets.

[1]nikolas@irif.fr
[2]{antigoni.polychroniadou,akira.takahashi}@jpmorgan.com
[3]{jiayixu7,chenkai.weng}@asu.edu

Contents

1 Introduction **3**
1.1 Our contribution . . . 3

2 Preliminary and Problem Statement **4**
2.1 XGBoost . . . 5
2.2 Zero-Knowledge Proof of Training . . . 5

3 Design of zkPoT for XGBoost **5**
3.1 zkPoT Template for XGBoost Training . . . 5
3.2 ZKP Gadgets for Nonlinear Relations . . . 7

4 Experiments **8**
4.1 Fixed-point XGBoost . . . 8
4.2 Zero Knowledge XGBoost-FP . . . 9

5 Conclusion **10**

A Additional Preliminaries **13**
A.1 Further Details of XGBoost . . . 13
A.2 Details of Commit-and-Prove Zero Knowledge Proofs . . . 15
A.3 Zero-Knowledge Proof based on Vector Oblivious Linear Evaluation . . . 15

B Related Work **16**

C Details of Fixed-Point XGBoost Training **17**
C.1 Numerical Safeguards in Fixed-Point XGBoost . . . 18

D Details of Certification Algorithm for XGBoost Training **20**

E Additional Details on ZK XGBoost-FP **23**
E.1 Proof of Comparison . . . 23
E.2 Proof of Division and Truncation . . . 25
E.3 Additional Components . . . 25
E.4 Security Analysis . . . 26

F Details of Certificate for Random Forest Training **26**

G Additional Experiments **28**
G.1 Fixed-Point XGBoost Experiments . . . 28
G.2 Discussion for Fixed-Point XGBoost Experimental Results . . . 28
G.3 Multiclass Classification . . . 29
G.4 ZKP Benchmarks . . . 30

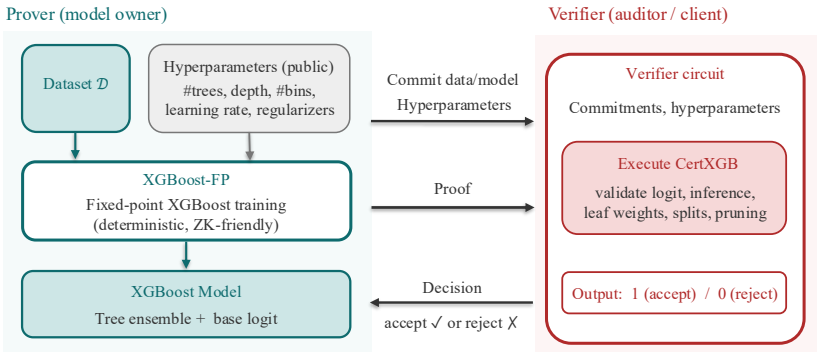


Figure 1: Overview of the ZKBoost protocol for certifying XGBoost training in zero-knowledge. To simplify, we show the owner of the model also owning the dataset but the dataset can belong to some other party in practice.

1 Introduction

As reliance on ML models grows, concerns about integrity and accountability are increasing. Today, when a model $\mathcal{M}$ is deployed, there is no evidence that it was truly obtained by training on dataset $\mathcal{D}$ with prescribed hyperparameters. This gap is important. A dishonest provider could ship a hand-crafted model, mix in unauthorized data, or skip parts of training. Clients have no way to distinguish such shortcuts from genuine training, because model providers in practice are not willing to reveal proprietary training data or the model parameters due to privacy and intellectual property concerns. An emerging approach to address this gap is zero knowledge proofs (ZKP) [GMR89], which allows a *prover* to convince a *verifier* of the truth of a statement without revealing anything beyond its validity. In the context of machine learning, it enables *zero knowledge Proof of Training* (zkPoT), allowing a provider (prover) to convince clients (verifier) that "private model $\mathcal{M}$ is the result of a public training algorithm Train on cryptographically committed dataset $\mathcal{D}$" without exposing either $\mathcal{D}$ or intermediate training states.

There has been tremendous progress recently in (see Sec. B for the literature review), especially for neural networks [APKP24, LF25, WSS+24], logistic regression [GGJ+23, TGM+25] and ordinary decision trees [PP24], enabling a number of important applications: (i) *trustworthy ML-as-a-service*, where clients obtain assurance of training integrity; (ii) *decentralized ML*, where smart contracts can reward provable training on public data; and (iii) *compliance with data restrictions*, where clients verify that models were trained on approved data sources (e.g., census data) or that certain protected content (e.g., copyrighted material) was excluded [WSS+24]. The last application is particularly powerful: by combining zkPoT with additional ZKP for data-compliance constraints, it prevents providers from arbitrarily manipulating data.

In this work, we turn our attention to zkPoT for gradient boosted decision trees, and in particular the XGBoost library [CG16]. XGBoost has become one of the most widely used methods for structured data, routinely outperforming deep networks on medium-sized tabular datasets [GOV22, SGL24], dominating ML competitions, and seeing widespread adoption in finance (especially in fraud detection) and healthcare. As such, ensuring integrity and provenance of XGBoost models has wide-reaching implications in practice.

1.1 Our contribution

We propose ZKBoost (illustrated in Figure 1), the first zkPoT protocol allowing a model provider to prove that a classifier was correctly obtained by executing the XGBoost training algorithm on a private dataset. We make two technical contributions, followed by a practical implementation and evaluation on standard benchmarks:

Generic template for zkPoT of XGBoost. To provide a general and optimized template for zkPoT of XGBoost, we first develop a *certification algorithm* CertXGB that verifies, given a resulting model $\mathcal{M}$ and dataset $\mathcal{D}$, that $\mathcal{M}$ was correctly produced by XGBoost training—*without re-executing the entire training procedure itself.* The key insight is a fundamental restructuring of the training validation process without sacrificing accuracy. Naive re-execution of XGBoost training builds each

tree *top-down* (i.e., partitioning training samples data at each node from root to leaves) and *sequentially* (with each tree depending on prior-tree predictions). The former requires expensive machinery if instantiated in ZKP (e.g., oblivious data partitioning) and the latter prevents parallelization across trees. Our CertXGB instead verifies each tree *bottom-up* and multiple trees *in parallel*. This restructuring decouples individual tree validation from inter-tree dependency checks, enabling parallel proof generation across trees and fundamentally changing the proof cost structure. CertXGB is general: it plugs into any ZKP backend for arithmetic circuits. Our certification further supports validation of pre-binning and pruning (without leaking tree topology), which is crucial for practical XGBoost to prevent overfitting but unaddressed in prior zkPoT for ordinary decision trees [PP24] (see Sec. B for a comparison).

Secure zkPoT instantiation with VOLE-ZKP. To instantiate zkPoT, we design a ZK protocol in which the verifier checks that the prover has faithfully executed CertXGB, thereby ensuring that the private tree ensemble $\mathcal{M}$ was genuinely obtained by running XGBoost on the committed training data $\mathcal{D}$. Concretely, the prover first generates cryptographic commitments to $\mathcal{M}$ and $\mathcal{D}$, and both parties then jointly execute a state-of-the-art vector oblivious linear evaluation (VOLE)-based ZKP protocol [YSWW21a, WMK16], which offers fast proving time with practical communication costs. A subtle but critical challenge arises in this instantiation. ZKP protocols typically emulate program execution as combinations of modular arithmetic over a finite ring, and a malicious prover may deliberately induce modular overflow in intermediate computations and exploit it to make model $\mathcal{M}$ appear valid to the verifier even if it was *not* faithfully produced by XGBoost training (see Sec. 3.2 for details). To prevent such adversaries, we design sound ZKP subprotocols for common ML operations including *comparison*, *division*, and *truncation*, alongside global *range checks* of training traces. These subprotocols are of independent interest, as the overflow vulnerability and our countermeasures are not specific to XGBoost and apply broadly to ZKML systems.

Evaluation of ZKBoost and Fixed-point XGBoost. We implement ZKBoost in C++ and evaluate it on three real-world datasets: breast cancer, credit default, and covertype. As no prior zkPoT for full XGBoost exists, we also compare against Sparrow [PP24]—the closest related work, which supports ordinary decision trees and random forests—by instantiating a simpler version of our protocol for random forest training. Despite providing stronger security guarantees (tree-topology hiding and more rigorous soundness checks), our scheme is 3–6× faster than Sparrow across dataset sizes. Our implementation also resolves a gap between standard XGBoost and a program representation amenable to ZKP. Standard XGBoost relies on floating-point arithmetic, which is doubly problematic for zkPoT: it is incompatible with the finite-ring arithmetic underlying ZKP systems, and it is inherently non-deterministic—results can vary across hardware platforms, compilers, and rounding modes, so there is no unique training trace for the verifier to check against. As a side contribution, we therefore implement and evaluate a *fixed-point* version of XGBoost, building on standard approximation techniques from the privacy-preserving ML literature (e.g., [Cd10, JGB+23]). We show empirically that fixed-point XGBoost matches standard floating-point XGBoost to within 1% accuracy across all datasets and hyperparameter configurations.

2 Preliminary and Problem Statement

Data Structures and Notation We use the bracket notation $[n]$ to denote the set $\{1, 2, \ldots, n\}$ for a positive integer n. Datasets are collections $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ with feature vectors $\mathbf{x}_i \in \mathbb{R}^d$ and binary labels $y_i \in \mathcal{Y} = \{0, 1\}$. In this paper, we use the following notation: n as the number of data points in $\mathcal{D}$; m as the number of trees (i.e. weak learners) in $\mathcal{T} = \{T_k\}_{k\in[m]}$; d as the number of features; B as the number of bins; h as the height of T_k; $N = 2^h$ as the number of leaves in T_k, assuming T_k is a full binary tree (containing dummy nodes as explained in 3.1) with height h;[1] $N-1$ as the number of non-leaf nodes in T_k; γ, λ as the regularization parameters; η as the learning rate. To index various quantities, we use k to index trees; i to index data points; f to index features; b to index bins; $\ell \in [2N-1]$ to index nodes in a tree; $l \in [N]$ to index leaf nodes in a tree; j to index others (e.g., level of a node). With this notation, each tree T_k is represented as a tuple $(\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k) \in [d]^{N-1} \times \mathbb{R}^{N-1} \times \mathbb{R}^N$. For each non-leaf node $\ell \in [N-1]$, $f_{k,\ell}$ indicates the feature index used for splitting, and $t_{k,\ell}$ indicates the threshold. Each leaf node $l \in [N]$ contains a weight

[1] We focus on equal-width pre-binning and a canonical tree structure, which simplifies certification but differs from some production XGBoost features (e.g., quantile histograms or specialized missing-value handling).

$w_{k,l}$. For each data point $\mathbf{x}_i$, T_k classifies it into a leaf node $l_{k,i}$ by traversing the tree from the root to a leaf node according to the feature values of $\mathbf{x}_i$.

2.1 XGBoost

We focus on the high-level training workflow. We defer more comprehensive background to Section A.1 and the original paper [CG16]. A decision tree maps an input feature vector $\mathbf{x} \in \mathbb{R}^d$ to a prediction by recursively partitioning the feature space using threshold-based splits and assigning a value at each leaf. While such models are simple and interpretable, a single tree often lacks sufficient expressive power. Gradient boosting addresses this limitation by constructing an *ensemble* of trees $\mathcal{T} = \{T_k\}_{k\in[m]}$ sequentially, where each new tree is trained to correct the residual errors of the current model. XGBoost [CG16] is a widely used and optimized implementation of this paradigm, relying on first and second order loss derivatives to guide split selection and leaf-weight assignment across multiple boosting rounds. As a result, each stage of training depends on the predictions produced by all previous trees, making the training process inherently stateful. This dependency highlights why certifying XGBoost training is substantially more challenging than certifying inference or the construction of a single decision tree.

2.2 Zero-Knowledge Proof of Training

Zero-Knowledge Proofs and Commitments. ZKP (of knowledge) allows one party (a *prover*) to prove knowledge of a *secret witness* w for a *public statement* x to another party (*verifier*). Such proofs are constructed for a concrete *NP relation* $\mathcal{R}$, describing the relationship between x and w. Formally, ZKP for an NP relation $\mathcal{R}$ is a tuple of interactive Turing machines $(\mathcal{P}, \mathcal{V})$, where $\mathcal{P}$ is prover and $\mathcal{V}$ is verifier. Then $\mathcal{P}$ and $\mathcal{V}$ interact with each other, where both $\mathcal{P}$ and $\mathcal{V}$ take x as common inputs, and $\mathcal{P}$ additionally takes w as a private input. At the end of interaction, $\mathcal{V}$ outputs a binary b. Proof systems that are used in verifiable ML typically require the following security properties: For an NP relation $\mathcal{R}$, they must provide *completeness* (i.e., if prover and verifier follow the protocol with input $(\mathsf{x}, \mathsf{w}) \in \mathcal{R}$, verifier always accepts), *knowledge soundness* (i.e., if verifier accepts, then it must be that prover owns a valid witness w satisfying given NP relation w.r.t. statement x), and *zero knowledge* (i.e., the transcript of the interaction between the prover and the (malicious) verifier leaks nothing except that there exists a witness w such that $(\mathsf{x}, \mathsf{w}) \in \mathcal{R}$). ZKP is typically paired with cryptographic *commitments* to secret witness w, ensuring that the prover is bound to a specific value while keeping it hidden from the verifier. Formally, a commitment scheme is defined as an algorithm Com that allows committing to a message m with randomness r: $c \leftarrow \mathsf{Com}(m; r)$. As these are standard building blocks in cryptography, we refer the reader to [Gol01] for more details.

Zero-Knowledge Proof of Training (zkPoT). Once ZKP and commitment schemes with the above properties are given, we can define a secure zkPoT for a training algorithm $\mathsf{TrainXGB}$ that takes a dataset $\mathcal{D}$ as input and outputs a model $\mathcal{M} = \mathsf{TrainXGB}(\mathcal{D})$. For training verification, we define the relation

$$R_{\text{xgb}} = \{((\mathcal{C}_{\mathcal{M}}, \mathcal{C}_{\mathcal{D}}), (\mathcal{M}, \mathcal{D}, r, \rho)) \ : \ \mathcal{C}_{\mathcal{M}} = \mathsf{Com}(\mathcal{M}; r); \mathcal{C}_{\mathcal{D}} = \mathsf{Com}(\mathcal{D}; \rho); \mathcal{M} = \mathsf{TrainXGB}(\mathcal{D})\} \tag{1}$$

A zkPoT protocol ensures that: (1) Completeness: An honest prover with dataset $\mathcal{D}$ and model $\mathcal{M} = \mathsf{TrainXGB}(\mathcal{D})$ can produce a valid proof. (2) (Knowledge) Soundness: Any prover that outputs a valid proof must "know" such a dataset $\mathcal{D}$ and a valid model $\mathcal{M}$ derived from $\mathcal{D}$ via $\mathsf{TrainXGB}$. (3) Zero-Knowledge: The proof leaks no information about $\mathcal{D}$ or $\mathcal{M}$ beyond the commitments. To formalize the above properties, we use the so-called *simulation-based security* defined in terms of a commit-and-prove ideal functionality (see Appendix A.2 for details).

3 Design of zkPoT for XGBoost

3.1 zkPoT Template for XGBoost Training

We first introduce our certification algorithm $\mathsf{CertXGB}$ for verifying the correctness of XGBoost training, and use it to describe a generic zkPoT template (Protocol 1) for relation R_{xgb} (1). Essentially, $\mathsf{CertXGB}$ represents a circuit that takes a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ and a claimed model $\mathcal{M} = (\mathcal{T}, z_0)$

as input, and then outputs 1 ("accept") or 0 ("reject") by checking that the model $\mathcal{M}$ is correctly produced by training on $\mathcal{D}$. The complete procedure is deferred to Appendix D.

Overview Recall that the original XGBoost (Appendix A.1) essentially proceeds as follows:

1. Initialize the base score (logit) z_0 from $\mathcal{D}$.
2. **Iterative Tree Building.** For each boosting round $k = 1, \ldots, m$, it grows a new tree T_k *from root to leaves* by (a) computing gradients and Hessians, (b) finding the best splits maximizing the gain, and (c) computing leaf weights. Finally, it computes the updated scores $\mathbf{z}_k = (z_{k,i})_{i=1}^n$.

Instead of naively re-executing the above training procedure, CertXGB takes advantage of the fact that the final model $\mathcal{M} = (\mathcal{T}, z_0)$ is provided as input. At a high level, CertXGB verifies that each tree T_k in $\mathcal{T}$ is correctly built by shuffling the order of operations in Step 2 above: it first evaluates T_k on every data point $\mathbf{x}_i$ to determine the reached leaf indices $l_{k,i}$ and update the score $z_{k,i}$, and then reconstructs the histograms of gradients and Hessians at each node using these indices and scores *from leaves to root*. Finally, it checks that each split in T_k maximizes the gain computed from the reconstructed histograms, and that each leaf weight is consistent with the corresponding histogram. In slightly more detail, CertXGB verifies that $(\mathcal{T}, z_0)$ is the output of TrainXGB on input $(\mathbf{x}, \mathbf{y})$, following the steps below. Note that Step 3, dominating the validation costs, can be parallelized, because each tree validation is independent of others and inter-tree dependencies can be handled separately in Step 2. See Section C. The reader may jump to the detailed procedure by clicking each subroutine.

1. ValidateLogit: To validate the base score z_0, CertXGB computes the initial prediction probability $p = \frac{1}{n}\sum_{i=1}^n y_i$ and then converts it to the logit $z_0' = \sigma^{-1}(p) = \log(p/(1-p))$. The circuit asserts that $z_0 = z_0'$ and sets $z_{0,i} = z_0$ for all $i \in [n]$.
2. **Initialize intermediate scores.** For each tree $k \in [m]$ and each data point $i \in [n]$, CertXGB evaluates the tree $T_k = (\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k)$ on $\mathbf{x}_i$ to compute the reached leaf $l_{k,i} \in [N]$, and updates the score $z_{k,i} \leftarrow z_{k-1,i} - \eta \cdot w_{k,l_{k,i}}$.
3. **Tree Validation.** For each tree $k \in [m]$, do:
   (a) ValidateInference: Validate that the computed leaf indices $\mathbf{l}_k = (l_{k,i})_{i=1}^n$ are correct by checking that each $l_{k,i}$ is consistent with the feature splits $(\mathbf{f}_k, \mathbf{t}_k)$ and the input $\mathbf{x}_i$.
   (b) InitHists: Computes histograms (G_k, H_k) by aggregating gradients and hessians over bins and leaves using the bin indices and leaf indices $\mathbf{l}_k$. Here, when computing the gradients $g_i = p_i - y_i$ and hessians $h_i = p_i(1-p_i)$, CertXGB uses the prediction probabilities computed via sigmoid: $p_i = \sigma(z_{k-1,i}) = 1/(1+e^{-z_{k-1,i}})$.
   (c) ValidateLeafWeights: Validate the leaf weights $\mathbf{w}_k$ by checking consistency with the histograms (G_k, H_k).
   (d) ValidateSplits: Validate the splits $(\mathbf{f}_k, \mathbf{t}_k)$ by ensuring (1) the gain derived from each split is maximal, *or* (2) if the gain is ≤ 0 (i.e., pruning condition is met), the corresponding node contains dummy values.

In Appendix D, we prove the following claim:

Claim 1 *The certificate algorithm* CertXGB *verifies the correctness of XGBoost training. That is, for any* $\mathcal{D} = (\mathbf{x}, \mathbf{y}), \mathcal{T} = \{T_k\}_{k\in[m]}$ *and* z_0, $\mathsf{CertXGB}(\mathbf{x}, \mathbf{y}, \mathcal{T}, z_0) = 1$ *if and only if* $(\mathcal{T}, z_0) = \mathsf{TrainXGB}(\mathbf{x}, \mathbf{y})$.

Pruning without leaking tree structure XGBoost prunes internal nodes (marking them as leaves) to prevent overfitting when the maximum gain is ≤ 0; revealing which nodes are pruned leaks tree topology to the verifier. We replace pruned nodes with dummy values $(f_{\text{dum}}, t_{\text{dum}})$: $f_{\text{dum}} \in [d]$ can be an arbitrary feature, and t_{dum} is set to a vacuous constant (e.g., `DBL_MIN`) so that any sample goes right. ValidateSplits, if instantiated with ZKP, can then check obliviously that dummies are used exactly when the pruning condition holds.

Fixed-point arithmetic and approximation of non-linear operations. While the above procedure describes the high-level logic of CertXGB, the actual implementation involves careful handling of fixed-point arithmetic and approximations to ensure that all computations are sound and efficient in a proof system. For example, the sigmoid $\sigma(z_{k-1,i})$ is implemented as a piecewise-linear approximation, and all divisions are reformulated to avoid floating-point operations. The complete details are provided in Appendix C.

Protocol 1: zkPoT for XGBoost

Parameters: Number of points n, features d, trees m, bins B, leaves $N = 2^h$ at depth h, learning rate η, regularizers λ, γ.
Public Input: Auxiliary commitments $\mathsf{com}_{\mathcal{D}}$ to training data $\mathcal{D} = (\mathbf{x}, \mathbf{y})$, $\mathsf{com}_{\mathcal{T}}$ to trees $\mathcal{T} = \{T_k\}_{k\in[m]}$ with $T_k = (\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k)$, and com_z to base logit z_0.
Private Input of $\mathcal{P}$: Fixed-point training data $(\mathbf{x}, \mathbf{y})$, the fitted tree ensemble $\mathcal{T}$, auxiliary traces, and commitment randomness.

Validate Commitments. $\mathcal{P}$ and $\mathcal{V}$ run a subprotocol for proving knowledge of commitment openings: $\mathsf{com}_{\mathcal{T}} = \mathsf{Com}(\mathcal{T}; r_{\mathcal{T}}), \mathsf{com}_z = \mathsf{Com}(z_0; r_z), \mathsf{com}_{\mathcal{D}} = \mathsf{Com}(\mathcal{D}; r_{\mathcal{D}})$.

Validate Binary Input. $\mathcal{P}$ and $\mathcal{V}$ run a subprotocol for asserting $y_i \in \{0, 1\}, \forall i \in [n]$

Compute Edges and binID. $\mathcal{P}$ and $\mathcal{V}$ run a subprotocol for $(\mathsf{edges}, \mathsf{binID}) \leftarrow \mathsf{PreBinFP}(\mathbf{x}, B)$, where edges are bin left edges and binID are bin indices for each feature of each sample; expose authenticated $(\mathsf{edges}, \mathsf{binID})$ to be reused throughout.

Validate Base Logit. $\mathcal{P}$ and $\mathcal{V}$ run a subprotocol for $\mathsf{ValidateLogit}(\mathbf{y}, z_0) = 1$ to validate the initial logit $z_0 = 2\,\mathrm{atanh}(2p - 1)$ with $p = \frac{1}{n}\sum_i y_i$. They then define the replicated vector $\mathbf{z}_0 = (z_{0,i})_{i=1}^n$ with $z_{0,i} = z_0$.

Initialize Intermediate Scores. For each tree $k \in [m]$ and data point $i \in [n]$, $\mathcal{P}$ computes the reached leaf $l_{k,i} \in [N]$ by evaluating T_k on $\mathbf{x}_i$, updates the score $z_{k,i} \leftarrow z_{k-1,i} - \eta \cdot w_{k,l_{k,i}}$, and defines $\mathbf{z}_k \leftarrow (z_{k,i})_{i=1}^n$. Let $\mathbf{l}_k = (l_{k,i})_{i=1}^n$.

For each tree $k \in [m]$, perform the following:

Validate Inference. $\mathcal{P}$ and $\mathcal{V}$ run a subprotocol for $\mathsf{ValidateInference}(\mathbf{x}, \mathbf{f}_k, \mathbf{t}_k, \mathbf{l}_k) = 1$ to validate $\mathbf{l}_k$.

Initialize Node Histograms. $\mathcal{P}$ and $\mathcal{V}$ run a subprotocol for $(G_k, H_k) \leftarrow \mathsf{InitHists}(\mathbf{z}_{k-1}, \mathbf{y}, \mathsf{binID}, \mathbf{l}_k)$, where $G_k[f][\ell][b]$ and $H_k[f][\ell][b]$ are the sum of gradients and hessians of samples in node ℓ whose feature f falls into bin b. These are computed by summing over samples in each leaf l and propagating them up to the root.

Validate Leaf Weights. $\mathcal{P}$ and $\mathcal{V}$ run a subprotocol for $\mathsf{ValidateLeafWeights}(G_k, H_k, \mathbf{w}_k) = 1$ to validate all leaf weights $\mathbf{w}_k$.

Validate Tree Splits. $\mathcal{P}$ computes $(\mathbf{f}_k, \mathbf{t}_k) \leftarrow (f^*_{k,\ell}, t^*_{k,\ell})_{\ell\in[N-1]}$, where $(f^*_{k,\ell}, t^*_{k,\ell})$ for ℓ-th internal node is a split maximizing the gain derived from G_k, H_k. $\mathcal{P}$ and $\mathcal{V}$ run a subprotocol for $\mathsf{ValidateSplits}(G_k, H_k, \mathbf{f}_k, \mathbf{t}_k, \mathbf{f}^*_k, \mathbf{t}^*_k, \mathsf{edges}) = 1$ to check that (1) the gain derived from each split in $(\mathbf{f}_k, \mathbf{t}_k)$ is maximal, *or* (2) if the gain is ≤ 0, the corresponding node contains dummy values.

3.2 ZKP Gadgets for Nonlinear Relations

In Sections 3.1 and C we build certification relation tailored to XGBoost and its fixed-point implementation ($\mathsf{TrainXGB}$). Next, we tackle the proof of low-level nonlinear functions that dominate the overhead of zkPoT for $\mathsf{TrainXGB}$. Denote $[\![\cdot]\!]$ as the witnesses committed by $\mathcal{P}$, the proof needs

- The proof of comparison between two signed fixed-point numbers, denoted as $[\![s]\!] \leftarrow \mathbf{1}\{[\![x]\!] < [\![y]\!]\}$. It proves $s = 1$ if $x < y$, and $s = 0$ otherwise.
- The proof of division between two fixed-point numbers, denoted as $[\![z]\!] \leftarrow [\![x]\!] \,/\, [\![y]\!]$.
- The proof of truncation after the multiplication of two fixed-point numbers, denoted as $[\![y]\!] \leftarrow \mathsf{Trunc}([\![x]\!])$, where $[\![y]\!] = \lfloor [\![x]\!] / 2^f \rfloor$ assume f-bit precision.
- The proof of histogram construction denoted as $[T] \leftarrow \mathsf{Histogram}([\![\boldsymbol{a}]\!], [\![\boldsymbol{v}]\!])$ which sets the i-th histogram bucket $[\![T_i]\!] = \sum v_j \cdot \mathbf{1}\{a_j = i\}$.

Existing approaches do not satisfy our goal due to efficiency, vulnerability or lack of support for fixed-point arithmetic. We design rigorous constraint systems to verify these operations with the focus of both efficiency and soundness. We sketch our main contribution in a high level and defer the formalized description and the instantiation of other components to Appendix E.

Comparison. We improve the idea of [HCL+24] to prove the comparison relation by bits-decomposition. Note that we use a signed representation. Hence, we have $\mathbf{1}\{[\![x]\!] < [\![y]\!]\} = \mathsf{MSB}([\![x - y]\!])$, where MSB refers to the most significant bit of $z = x - y$. The main task is shifted to proving $[\![s]\!] = \mathsf{MSB}([\![z]\!])$

given committed $(\llbracket z \rrbracket, \llbracket s \rrbracket)$. We describe prior approach in detail in Appendix E.1, and focus on improving its consistency checks on invalid bits-decomposition.

We abstract out the problem in the following way. The proof of bit-decomposition requires $\mathcal{P}$ to commit to groups of bits $(z_0, \ldots, z_{t-1})$ each of d-bit, and show that $z = 2^{n-1} \cdot s + \sum_{i=0}^{t-1} 2^{id} \cdot z_i$. However, a cheating $\mathcal{P}$ may commit to incorrect $(\tilde{z}_0, \ldots, \tilde{z}_{t-1})$ such that $z+p = 2^{n-1} \cdot s' + \sum_{i=0}^{t-1} 2^{id} \cdot \tilde{z}_i$, which would completely flip the MSB s. In the context of zkPoT for XGBoost, this is particularly damaging: comparison results determine whether one split gain is greater than another, so a flipped MSB could allow a cheating $\mathcal{P}$ to commit to a tree split that does *not* achieve the maximum gain—passing verification while certifying a model that deviates from the prescribed training algorithm.

Our solution employs a Mersenne prime in the form of $p = 2^n - 1$. The only chance a cheating $\mathcal{P}$ has is when $z = 0$, it may claim $s = 1$ and $\tilde{z}_i = 2^d - 1$ for all $i \in [0, t)$. By defining $\llbracket w \rrbracket = \mathbf{1}\{\llbracket z \rrbracket \neq 0\}$, we observe that $\mathsf{MSB}(z) = \begin{cases} 0 & \text{if } z = 0, w = 0 \\ s & \text{if } z \neq 0, w = 1 \end{cases}$. It implies that $(1 - w) \cdot s = 0$ should always be true. Additionally, we employ the non-equality-zero check from [PHGR13] to prove $\llbracket w \rrbracket = \mathbf{1}\{\llbracket z \rrbracket \neq 0\}$. Our solution to address the soundness issue only requires committing 2 extra values and proving 3 multiplicative relations, while the previous work almost costs 2× [HCL+24].

Division and Truncation. Proof of division takes inputs $(\llbracket x \rrbracket, \llbracket y \rrbracket, \llbracket z \rrbracket)$ and proves that $z = \lfloor x/y \rfloor$. In our TrainXGB, the numerators and denominators may range from 0 to nearly $p/2^f$. Previous proofs of division are not suitable since they either only work for bounded small values [LXZ21, PP24] or requires thousands of logical constraints [WYX+21a].

To prove the relation, we leverage the existence of a residue $r \in [0, y)$ such that $x = z \cdot y + r$. Additionally, two range checks are required to maintain the soundness: $r \in [0, y)$, and $z \in [0, \lfloor p/y \rfloor]$. The first check ensures r is a residue. The second check is needed to prevent the similar wrap-around issue happened in the proof of MSB. Namely, there could be many possible pairs of (z', r') such that $x = z' \cdot y + r' \bmod p$. The proof should ensure a z that satisfies $x = z \cdot y + r$ without modulo p. The truncation is a simplified division with public and positive denominators $y = 2^f$.

Proof of Histogram Construction. Our proof commits to a vector $\{(a_j, v_j)\}_{j \in [N]}$ and a histogram $T = \{T_i\}_{i \in [n]}$, and proves $\llbracket T_i \rrbracket = \sum v_j \cdot \mathbf{1}\{a_j = i\}$. A naive approach following the previous proof of random forest [PP24] is to utilize a ZK-RAM: for $j \in [N]$, prove a read at location a_j, increment its value by v_j, and write it back. It incurs $O(N + n)$ overhead with a large constant [YH24].

We observe that the proof of histogram construction is simpler than ZK-RAM since it only consists of *write* operation and the operator is public. Hence, it can be proven by a special weighted LogUp proof by showing an equation $\sum_{j \in [N]} \frac{v_j}{X + a_j} = \sum_{i \in [n]} \frac{T_i}{X + i}$ is correct with respect to a variable X [Hab22]. In our implementation, this reduces the overhead by around 3×.

Global Range Checks for Soundness. To ensure the soundness, global range check is needed for every intermediate values that are committed during the proof. This is because a cheating prover may leverage the overflow to cause inconsistency in numerical operations. We carefully analyze the range of witnesses and implement the range checks throughout the protocol.

Instantiation and Security. Our ZKP protocol can be instantiated by any general-purpose proof system. To achieve the maximum prover efficiency, we realize it in the VOLE-ZK framework [YSWW21a]. We state our main theorem of security below and provide details and security analysis in Appendix E.4.

Theorem 1. *Define the relation by the proof of fixed-point XGBoost Training described in R_{xgb} (1), Protocol 1 securely realizes $\mathcal{F}_{\mathsf{CP}}$ in the $(\mathcal{F}_{\mathsf{CVOLE}}, \mathcal{F}_{\mathsf{ZK}})$-hybrid model.*

4 Experiments

In this section, we first analyze how fixed-point representation affects the accuracy and efficiency of the XGBoost training in plaintext. Secondly, we benchmark the performance of our ZKBoost implementation in terms of the proving time, bandwidth usage, and memory usage, and compares it with the baseline scheme.

4.1 Fixed-point XGBoost

Setup. We compare our fixed-point arithmetic gradient boosting implementation ("Fixed") against XGBoost (scikit-learn API), using identical hyperparameters (depth, `bin`=128, $\eta = 0.3$, $\lambda = 1$,

Table 1: **Left**: Running time in LAN and WAN (minutes), communication overhead (GB), and memory usage (GB). Three datasets are BR, CR, CO. With 100 trees of depth 5. **Right**: Running time for the proof of random forest training (minutes). Parameters are dataset size $n \in \{2^{14}, 2^{16}, 2^{18}\}$, $h = 5, m = 64, d = 16$. Sparrow's bagging time (30%) is taken out.

	LAN	WAN	Communication	Memory
BR	18.65	31.93	52	0.284
CR	21.81	37.64	61	0.433
CO	52.53	93.15	154	0.710

n	2^{14}	2^{16}	2^{18}
Sparrow	17.64	45.01	155.19
Ours, LAN	5.52	10.01	25.96
Ours, WAN	9.76	18.38	52.69

$\gamma = 0$) and identical train/test splits per configuration. Each result reported is an average over 10 runs and we observed negligible variance. We evaluate *equal-thread* runs with both OpenMP and BLAS pinned to one thread.

Datasets. We use four standard binary classification benchmarks: **Breast Cancer** ($n = 569, d = 30$), **Default of Credit Card Clients** ($n = 30001, d = 23$), two **Covertype** subsets (converted for binary classification), **Covertype 50k** ($n = 50000, d = 54$) and **Covertype 100k** ($n = 100{,}000, d = 54$), and **Adult** ($n = 45222$, $d = 104$). We abbreviate the first three to **BR**, **CR**, **CO**.

Accuracy parity. Across all benchmarks and the tested hyperparameter grid, our fixed-point training *tracks* floating-point XGBoost to within statistical error. Every configuration satisfies $|\Delta| \leq 1\%$. Notably, even on deeper/larger ensembles that are not presented here, the maximum observed gap remains under 0.01. These results confirm that adopting fixed-point arithmetic, together with our logit initializer, piecewise sigmoid, and fixed-point gain computation, does not degrade predictive accuracy in practice. Due to space constraints we defer the tables to Appendix G.1 and the detailed performance discussion in Appendix G.2.

4.2 Zero Knowledge XGBoost-FP

Implementation and Setup. We implement the ZK XGBoost-FP in C++ over the VOLE-ZK framework [YSWW21a] based on EMP-toolkits [WMK16]. All experiments only employ 1 thread and report end-to-end running time. The prover and verifier are each hosted by a AWS EC2 m5.2xlarge instances located in the same region. Each is equipped with 8 vCPUs and 32GB RAM. To simulate various network condition, we use the Linux tc tool to configure the bandwidth and latency. Specifically, we emulate a local area network (LAN, 5Gbps bandwidth) and a wide-area network (WAN, 1Gbps bandwidth and 60ms round-trip latency).

Building Blocks. We improved the proof of comparison (against wrap-around attack) and histogram construction. The former makes 5.3× improvement in LAN and 2.2× in WAN. The latter makes 4× improvement in LAN and 3× in WAN. Details in Appendix G.4.

Benchmarking Datasets. We benchmark the overhead of proving the XGBoost training over datasets BR, CR, CO and show the results in Table 1 (left). With the large 60ms latency, the WAN running time is 83% to 87% more than it LAN due to VOLE interactions. However, all proofs are generated in a reasonable amount of time. Note that as demonstrated in our accuracy tests, training with 50 to 100 trees already achieves high accuracy. Hence, the results in Table 1 can be viewed as an upper bound of runtime in the real world. The overall performance on general-purpose servers also demonstrates that our lightweight scheme is suitable for commodity hardware.

Running Time with Respect to Number of Trees. We demonstrate the relation between the running time and the number of trees ranging from 25 to 200. We use the largest CO dataset with 40k data points, 54 features, and tree-depth 5. The result in Figure 2 (left) shows that the running time is strictly linear to the number of trees. We use 1 thread for consistency with common ZKP benchmarks. As discussed in Sec. 1.1, our scheme is friendly to multi-threading and can parallelize the proving of each tree.

Running Time with Respect to Dataset Size. We benchmark training using a high-dimension dataset CO with increasing number of data points, ranging from 10k to 40k. The result in Figure 2 (right) shows that the running time is not significantly impacted by the number of data points. This is due to the efficient proof of histogram construction mentioned in Section 3.2, which eliminates the bottleneck on arranging data points to histogram cells. It also demonstrate that our scheme can prove the training on much larger datasets without significant degradation on performance.

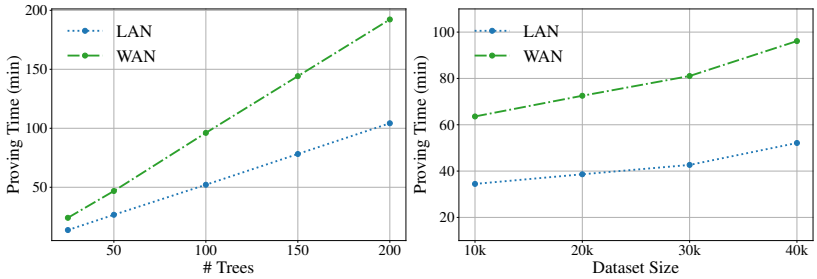


Figure 2: Benchmarking CO with increased number of trees (left) or data points (right).

zkPoT for Random Forest. Our scheme is the only ZKP for XGBoost training. To compare with a baseline, we simplify it to prove random forest training and compare with Sparrow [PP24]. The detailed approach is deferred to Appendix F and we provide a qualitative comparison in B. Sparrow is a memory-efficient zk-SNARK which supports public verifiability, our protocol is interactive. However, our scheme provides a few new features. Firstly, it hides tree topology by filling a full binary tree with dummy nodes for better privacy. Secondly, it enforces more rigorous consistency checks for nonlinear proofs (see Section 3.2). These additional guarantees lead to a larger circuit and, consequently, higher proof complexity.

Additionally, we observe that the ZK-RAM used by Sparrow for proving the read/write operation to histograms can be simplified to a proof of subset. It can also be viewed as a special case of our proof of histogram construction with the *write* value to always be 1. This method helps our scheme gain significant advantage since building the leaf-layer histogram is the main bottleneck for large datasets.

Table 1 (right) shows the running time of our proof of random forest training compared to Sparrow [PP24]. Our scheme achieves better performance across all tests in both LAN and WAN with up to 6× faster in running time.

5 Conclusion

We introduced the first zkPoT for gradient boosted decision trees, realized concretely for XGBoost. Our current instantiation targets binary classification, covering high-stakes applications such as fraud detection, credit risk prediction, and mortality prediction. Two natural directions for future work are (i) extending to multi-class XGBoost, which is conceptually straightforward via our framework (see Appendix G.3), and (ii) replacing the interactive VOLE-based instantiation with a non-interactive succinct proof system to enable public verifiability at the cost of higher proving time.

Acknowledgments

We would like to thank Daniel Escudero for useful discussions during the early stages of this project. Nikolas Melissaris is supported by ERC grant OBELiSC (101115790).

Disclaimer

This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

References

[APKP24] K. Abbaszadeh, C. Pappas, J. Katz, and D. Papadopoulos. Zero-knowledge proofs of training for deep neural networks. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pp. 4316–4330. ACM, 2024.

[BBMH+21] C. Baum, L. Braun, A. Munch-Hansen, B. Razet, and P. Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In *ACM CCS 2021*, pp. 192–211. ACM Press, 2021.

[BBMHS22] C. Baum, L. Braun, A. Munch-Hansen, and P. Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In *CRYPTO 2022, Part IV*, vol. 13510 of *LNCS*, pp. 329–358. Springer, Heidelberg, 2022.

[BCG+13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, vol. 8043 of *Lecture Notes in Computer Science*, pp. 90–108. Springer, 2013.

[BGH19] S. Bowe, J. Grigg, and D. Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019.

[BMRS21] C. Baum, A. J. Malozemoff, M. B. Rosen, and P. Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In *CRYPTO 2021, Part IV*, vol. 12828 of *LNCS*, pp. 92–122, Virtual Event, 2021. Springer, Heidelberg.

[Cd10] O. Catrina and S. de Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In *ESORICS 2010*, vol. 6345 of *LNCS*, pp. 134–150. Springer, Heidelberg, 2010.

[CFQ19] M. Campanelli, D. Fiore, and A. Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2075–2092, 2019.

[CG16] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, p. 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[CWSK24] B. Chen, S. Waiwitlikhit, I. Stoica, and D. Kang. ZKML: an optimizing system for ML inference in zero-knowledge proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*, pp. 560–574. ACM, 2024.

[DIO21] S. Dittmer, Y. Ishai, and R. Ostrovsky. Line-Point Zero Knowledge and Its Applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*, vol. 199 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 5:1–5:24, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[FKL+21] N. Franzese, J. Katz, S. Lu, R. Ostrovsky, X. Wang, and C. Weng. Constant-overhead zero-knowledge for RAM programs. In *ACM CCS 2021*, pp. 178–191. ACM Press, 2021.

[FQZ+21] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu. ZEN: An optimizing compiler for verifiable, zero-knowledge neural network inferences. Cryptology ePrint Archive, Paper 2021/087, 2021.

[GGG17] Z. Ghodsi, T. Gu, and S. Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 4672–4681, 2017.

[GGJ+23] S. Garg, A. Goel, S. Jha, S. Mahloujifar, M. Mahmoody, G. Policharla, and M. Wang. Experimenting with zero-knowledge proofs of training. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pp. 1880–1894. ACM, 2023.

[GJJZ22] S. Garg, A. Jain, Z. Jin, and Y. Zhang. Succinct zero knowledge for floating point computations. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, p. 1203–1216, New York, NY, USA, 2022. Association for Computing Machinery.

[GKR15] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, 2015.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[Gol01] O. Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, 2001.

[GOV22] L. Grinsztajn, E. Oyallon, and G. Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[Hab22] U. Haböck. Multivariate lookups based on logarithmic derivatives. *IACR Cryptol. ePrint Arch.*, 2022:1530, 2022.

[HCL+24] M. Hao, H. Chen, H. Li, C. Weng, Y. Zhang, H. Yang, and T. Zhang. Scalable zero-knowledge proofs for non-linear functions in machine learning. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 3819–3836, 2024.

[IKOS09] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM J. Comput.*, 39(3):1121–1152, 2009.

[JGB+23] N. Jawalkar, K. Gupta, A. Basu, N. Chandran, D. Gupta, and R. Sharma. Orca: FSS-based secure training with GPUs. Cryptology ePrint Archive, Report 2023/206, 2023. https://eprint.iacr.org/2023/206.

[LF25] Y. Li and X. Fan. SUMMER: Recursive zero-knowledge proofs for scalable RNN training. Cryptology ePrint Archive, Paper 2025/1688, 2025.

[LKKO24] S. Lee, H. Ko, J. Kim, and H. Oh. vcnn: Verifiable convolutional neural network based on zk-snarks. *IEEE Trans. Dependable Secur. Comput.*, 21(4):4254–4270, 2024.

[LXZ21] T. Liu, X. Xie, and Y. Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *ACM CCS 2021*, pp. 2968–2985. ACM Press, 2021.

[PHGR13] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pp. 238–252. IEEE Computer Society Press, 2013.

[PP24] C. Pappas and D. Papadopoulos. Sparrow: Space-efficient zksnark for data-parallel circuits and applications to zero-knowledge decision trees. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pp. 3110–3124. ACM, 2024.

[SBLZ25] H. Sun, T. Bai, J. Li, and H. Zhang. zkdl: Efficient zero-knowledge proofs of deep learning training. *IEEE Trans. Inf. Forensics Secur.*, 20:914–927, 2025.

[SGL24] A. Shmuel, O. Glickman, and T. Lazebnik. A comprehensive benchmark of machine and deep learning across diverse tabular datasets. *CoRR*, abs/2408.14817, 2024.

[SLY+25] Y. Sun, H. Liu, K. Yang, Y. Yu, X. Wang, and C. Weng. Committed vector oblivious linear evaluation and its applications. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pp. 3635–3648, 2025.

[SWF+23] A. S. Shamsabadi, S. C. Wyllie, N. Franzese, N. Dullerud, S. Gambs, N. Papernot, X. Wang, and A. Weller. Confidential-profitt: Confidential proof of fair training of trees. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[TGM+25] G. Tan, A. Gascón, S. Meiklejohn, M. Raykova, X. Wang, and N. Luo. Founding zero-knowledge proofs of training on optimum vicinity. Cryptology ePrint Archive, Paper 2025/053, 2025.

[WMK16] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[WSS+24] S. Waiwitlikhit, I. Stoica, Y. Sun, T. Hashimoto, and D. Kang. Trustless audits without revealing data or models, 2024.

[WYKW21] C. Weng, K. Yang, J. Katz, and X. Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pp. 1074–1091. IEEE Computer Society Press, 2021.

[WYX+21a] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 501–518, 2021.

[WYX+21b] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *USENIX Security 2021*, pp. 501–518. USENIX Association, 2021.

[YH24] Y. Yang and D. Heath. Two shuffles make a {RAM}: Improved constant overhead zero knowledge {RAM}. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 1435–1452, 2024.

[YSWW21a] K. Yang, P. Sarkar, C. Weng, and X. Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2986–3001, 2021.

[YSWW21b] K. Yang, P. Sarkar, C. Weng, and X. Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *ACM CCS 2021*, pp. 2986–3001. ACM Press, 2021.

[ZC24] H. Zhu and S. C.-K. Chau. Optimizing zero-knowledge proofs for verifiable decision trees. In *IEEE Symposium on Security and Privacy (SP) Poster Session*, 2024.

[ZFZS20] J. Zhang, Z. Fang, Y. Zhang, and D. Song. Zero knowledge proofs for decision tree predictions and accuracy. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pp. 2039–2053. ACM, 2020.

A Additional Preliminaries

A.1 Further Details of XGBoost

Theoretical Background

XGBoost is a scalable implementation of gradient boosted decision trees. Given a dataset $\mathcal{D}$ and a differentiable convex loss function $\ell : \mathbb{R} \times \mathcal{Y} \to \mathbb{R}$, the algorithm trains an ensemble of m regression

trees $\mathcal{T} = \{T_k\}_{k=1}^m$ such that the final prediction is

$$p = \sigma(z) \qquad z = z_0 - \sum_{k=1}^{m} \eta \cdot w_{k,i}$$

where σ is the sigmoid function, z is the final logit, $w_{k,i} = T_k(\mathbf{x}_i)$ is the output of tree T_k on input $\mathbf{x}_i$, $\eta \in (0, 1]$ is the learning rate, and z_0 is the initial prediction (logit). The training algorithm builds trees iteratively to minimize the regularized objective. At each boosting round k, the algorithm computes the first- and second-order derivatives of the loss:

$$g_{k,i} = \frac{\partial}{\partial p_{k-1}} \ell(p_{k-1}, y_i), \quad h_{k,i} = \frac{\partial^2}{\partial p_{k-1}^2} \ell(p_{k-1}, y_i),$$

where p_{k-1} is the prediction after $k-1$ rounds. These $(g_{k,i}, h_{k,i})$ values are called *gradients* and *Hessians*.

Split finding. For each candidate split (feature f and threshold t), the gain is computed as

$$\mathsf{Gain}(f, t) \;=\; \frac{1}{2} \cdot \left(\frac{G_{\mathcal{L}}^2}{H_{\mathcal{L}} + \lambda} + \frac{G_{\mathcal{R}}^2}{H_{\mathcal{R}} + \lambda} - \frac{(G_{\mathcal{L}} + G_{\mathcal{R}})^2}{H_{\mathcal{L}} + H_{\mathcal{R}} + \lambda} \right) - \gamma,$$

where $G_{\mathcal{L}} = \sum_{i \in \mathcal{L}} g_i$, $H_{\mathcal{L}} = \sum_{i \in \mathcal{L}} h_i$, and similarly for the right child $\mathcal{R}$. Hyperparameters λ, γ control regularization and split pruning. The best split maximizes the gain.

Leaf weights. Once a tree is grown, each leaf l receiving samples $\mathcal{I}_l$ gets a weight

$$w_l = \frac{\sum_{i \in \mathcal{I}_l} g_i}{\sum_{i \in \mathcal{I}_l} h_i + \lambda}.$$

Objective. The ensemble minimizes the regularized objective

$$L_k = \sum_{i=1}^{n} [g_{k,i} T_k(\mathbf{x}_i) + \tfrac{1}{2} h_{k,i} T_k(\mathbf{x}_i)^2] + \Omega(T_k),$$
$$\Omega(T) = \gamma N + \frac{1}{2} \lambda \sum_l w_l^2,$$

where N is the number of leaves. Training iteratively adds trees until m rounds are completed.

Loss function. In this work, we use the cross-entropy loss $\ell(p, y) = -y \log(p) - (1 - y) \log(1 - p)$, where $p = \sigma(z) \in (0, 1)$ is the predicted probability. Then the gradients and hessians wrt logit are

$$g = \frac{\partial \ell}{\partial z} = p - y, \quad h = \frac{\partial^2 \ell}{\partial z^2} = p(1 - p).$$

XGBoost Inference

Computation of the prediction for a given data point $\mathbf{x}$ proceeds in three steps:

1. **Tree Evaluation:** For each tree $T_k \in \mathcal{T}$, compute the weight $w_k = T_k(\mathbf{x})$.
2. **Score Aggregation:** Compute the final score (or *logit*) z:

   $$z = z_0 - \sum_{k=1}^{m} \eta \cdot w_k$$

   where z_0 is the base score determined during training and η is the learning rate.
3. **Probability Transformation:** Transform the score z in the log-odds space via the sigmoid function σ to yield the final probability:

   $$p = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

XGBoost Training

To train a tree ensemble $\mathcal{T}$ on a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, XGBoost first initializes the base score z_0, pre-bins features into B discrete bins, and then iteratively builds trees T_k for boosting rounds $k = 1, \ldots, m$ by selecting splits and computing leaf weights to maximize the gain. Note that z_0 is also part of the trained model, so the final model $\mathcal{M}$ consists of $\mathcal{T}$ and z_0. We elaborate on these steps below:

1. **Base Score Initialization.** It initializes the base score z_0 by taking the log-odds

$$z_0 = \sigma^{-1}(p) = \log \frac{p}{1-p} \tag{3}$$

   where $p = \frac{1}{n}\sum_i y_i$ is the mean label probability.

2. **Pre-binning.** Each feature column is discretized into B equal-width bins, and a lookup table $\mathsf{edges} \in \mathbb{R}^{d \times B}$ of left edges of bins is constructed. This *pre-binning* step replaces direct floating-point comparisons on raw feature values with integer bin indices, enabling efficient and deterministic split evaluation.

3. For each boosting round $k = 1, \ldots, m$, it builds a new tree T_k with the following key steps.

   (a) **Prediction and derivatives.** For every data point $i \in [n]$, the algorithm computes the current probability $p_i = \sigma(z_{k-1,i})$, the gradient $g_i = p_i - y_i$, and the Hessian $h_i = p_i(1-p_i)$.

   (b) **Split search.** For each node, containing samples indexed by $\mathcal{I} \subseteq [n]$, the algorithm scans candidate pairs (f, b) of feature and bin threshold defining partitions $\mathcal{L}$ and $\mathcal{R}$ such that $\mathcal{L} \cup \mathcal{R} = \mathcal{I}$. Then the best split (f^*, b^*) is selected by maximizing the gain:

$$\mathsf{gain} \;=\; \frac{1}{2} \cdot \left(\frac{G_{\mathcal{L}}^2}{H_{\mathcal{L}} + \lambda} + \frac{G_{\mathcal{R}}^2}{H_{\mathcal{R}} + \lambda} - \frac{G_{\mathcal{I}}^2}{H_{\mathcal{I}} + \lambda} \right) - \gamma,$$

   where $G_{\mathcal{I}} = \sum_{i \in \mathcal{I}} g_i$, $H_{\mathcal{I}} = \sum_{i \in \mathcal{I}} h_i$, and similarly for the children $\mathcal{L}$ and $\mathcal{R}$. Hyperparameters λ, γ control regularization and split pruning.

   (c) **Leaf weight computation.** Once the tree structure is finalized, each leaf $l \in [N]$ is assigned the weight

$$w_l = \frac{G_{\mathcal{I}_l}}{H_{\mathcal{I}_l} + \lambda},$$

   (d) **Logit update.** Finally, for each data point $i \in [n]$, the logit is updated as

$$z_{k,i} = z_{k-1,i} - \eta \cdot T_k(\mathbf{x}_i).$$

A.2 Details of Commit-and-Prove Zero Knowledge Proofs

Functionality 1: $\mathcal{F}_{\mathsf{CP}}[\mathcal{R}]$

The functionality $\mathcal{F}_{\mathsf{CP}}$ interacts with three parties: a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and an ideal adversary $\mathcal{S}$. It is parameterized by an NP relation $\mathcal{R}$.

Committing Phase Upon receiving $(\textsc{Commit}, \mathsf{w})$ from $\mathcal{P}$:

1: Store w internally and send $\textsc{Commit}$ to $\mathcal{S}$.
2: Upon receiving $(\textsc{Do-Commit})$ from $\mathcal{S}$, send $\textsc{Commit-Receipt}$ to $\mathcal{V}$.

Proving Phase Upon receiving $(\textsc{Prove}, \mathsf{x}, \mathsf{w})$ from $\mathcal{P}$:

1: If $(\mathsf{x}, \mathsf{w}) \notin \mathcal{R}$, ignore the input.
2: Else, send $(\textsc{Prove}, \mathsf{x})$ to $\mathcal{S}$.
3: Upon receiving $\textsc{Do-Prove}$ from $\mathcal{S}$, send $(\textsc{Prove-Receipt}, \mathsf{x})$ to $\mathcal{V}$.

A.3 Zero-Knowledge Proof based on Vector Oblivious Linear Evaluation

A series of works construct interactive ZKPs based on the vector oblivious linear evaluation (VOLE) [DIO21, WYKW21, BMRS21, YSWW21b, WYX+21b, BBMH+21, BBMHS22, HCL+24]. They require a preprocessing (witness-independent) phase that generates a uniformly sampled VOLE correlation

$\mathbf{w} = \mathbf{v} + \mathbf{u} \cdot \Delta$, from which $\mathcal{P}$ obtains $(\mathbf{u}, \mathbf{w})$ and $\mathcal{V}$ obtains $(\mathbf{v}, \Delta)$. Δ is denoted as the global key. In the proving (witness-dependent) phase, $\mathcal{P}$ and $\mathcal{V}$ transform the VOLE correlation into a commitment to the witness $\mathbf{x}$, i.e. $M[\mathbf{x}] = K[\mathbf{x}] + \mathbf{x} \cdot \Delta$. $M[\mathbf{x}]$ is denoted as the message authentication code and $K[\mathbf{x}]$ is denoted as the local key. We denote the commitment to $\mathbf{x}$ as $[\mathbf{x}]$.

Proving the linear relations is *free* over VOLE commitments since they are linearly homomorphic. I.e., the commitment to $[z] = [x] + [y]$ can be generated by defining $M[z] = M[x] + M[y]$ and $K[z] = K[x] + K[y]$. To prove the multiplicative relation $[z] = [x] \cdot [y]$, previous works [DIO21, YSWW21b] observe that

$$\begin{aligned} & K[x] \cdot K[y] + K[z] \cdot \Delta \\ & = M[x] \cdot M[y] - (x \cdot M[y] + y \cdot M[x] - M[z]) \cdot \Delta \\ & + (x \cdot y - z) \cdot \Delta^2 \end{aligned}$$

Note that $(x \cdot y - z) \cdot \Delta^2$ is canceled if $\mathcal{P}$ honestly commits to these values. Define $B = K[x] \cdot K[y] + K[z] \cdot \Delta$, $A_0 = M[x] \cdot M[y]$ and $A_1 = x \cdot M[y] + y \cdot M[x] - M[z]$, $\mathcal{P}$ simply proves that it holds a valid pair of (A_0, A_1) that satisfies $B = A_0 - A_1 \cdot \Delta$. This can be done by a mask-and-open. Furthermore, a large number of multiplicative relations can be proven in a batch by random linear combination.

For an arithmetic circuit of size C, the total communication overhead for the preprocessing phase is $O(\mathsf{polylog} C)$ with 2 rounds and the online proving phase is $O(C)$ with 1 round. Notably, the end-to-end round complexity is 2 and its online phase can be made non-interactive.

B Related Work

The field of zero-knowledge machine learning has evolved through two main phases: inference verification and training verification.

Comparison with Sparrow Sparrow [PP24], a zkPoT framework for ordinary decision trees using Gini impurity, lays important groundwork for our work. In particular, their histogram-based certification inspired our design of CertXGB, which initializes node histograms from leaf to root. However, gradient boosting requires more complex inter-tree dependencies and split search than simple decision trees, which hinder naive application of Sparrow. In addition, we address several technicalities that did not arise in Sparrow, including ZKP subprotocols for (1) pruning and binning, (2) range checks of training traces, making sure the prover did not cause arithmetic overflow to break soundness, (3) various fixed-point operations, such as division, gradients, hessians, sigmoid, etc., (4) handling corner cases such as division by 0. Moreover, we use VOLE-ZK as a backend for faster proving time, while Sparrow uses succinct non-interactive arguments of knowledge (SNARKs) to achieve sublinear verification time.

Foundational verifiable ML. Early work established the feasibility of verifiable neural network execution. [GGG17] with SafetyNets introduced verifiable execution of deep neural networks using interactive proofs based on the GKR protocol ([GKR15], representing DNNs as arithmetic circuits for verification with practical performance for networks with millions of parameters, though without zero-knowledge privacy.

Zero-knowledge inference verification. Recent systems have made ZK inference practical for large neural networks. [LKKO24] with vCNN presented the first practical zero-knowledge proof system for CNN inference using zk-SNARKs ([BCG+13]), developing circuit-friendly representations of convolution and pooling operations with 8500*times* speedup over naive approaches. [FQZ+21] with ZEN proposed zero-knowledge neural network inference using Quadratic Arithmetic Program-based zk-SNARKs with optimizations for ReLU activations. Building on these foundations, [CWSK24] in ZKML developed a compiler framework that automatically translates TensorFlow models into Halo2 zk-SNARK circuits ([BGH19]), achieving orders-of-magnitude reductions in proof generation time for large models like ResNet-18 and GPT-2 through specialized "gadgets" for activation functions and circuit layout optimization.

Decision tree inference verification. [ZFZS20] formalized the first zero-knowledge proof protocols specifically for decision tree inference and accuracy testing, enabling model owners to prove tree predictions or accuracy without revealing structure or data. Their implementation demonstrates practical performance, proving accuracy of a 1,029-node tree on 5,000 test samples in about 250

seconds with 287 KB proof size. [ZC24] improved upon this work by replacing hash-based commitments with polynomial commitments, achieving over 50× reduction in multiplication gates through a "modular" approach that decouples tree structure from parameters.

Proof of training Training verification is fundamentally more challenging than inference due to the iterative, data-intensive nature of learning. [GGJ+23] coined the term "zero-knowledge proofs of training" and provided the first rigorous security definitions, demonstrating feasibility with a zkPoT protocol for logistic regression using MPC-in-the-head ([IKOS09]) techniques combined with zk-SNARKs. For deep neural networks, [SBLZ25] presented zkDL, introducing zkReLU (SNARK-friendly ReLU gradient proofs) and FAC4DNN (aggregated circuit design), achieving proof generation for an 8-layer DNN (10M parameters, batch size 64) in under 1 second per batch. [APKP24] built Kaizen, a complete zkPoT system using Halo2 that proves the entire training process including weight updates while maintaining privacy of both data and model.

[TGM+25] recently demonstrated a *rejection sampling attack* on zkPoT protocols, allowing a malicious prover to choose the training randomness to bias the model without detection. As our underlying XGBoost training is deterministic, this attack does not apply to our setting.

Decision tree fairness verification. Shamsabadi et al., [SWF+23] introduced Confidential-PROFITT, which proves decision trees were trained under certain fairness constraints. Their approach differs from traditional training verification; instead of proving each training step, they designed a certified training algorithm where information gain on sensitive attributes is bounded, outputting both an optimal tree and a zero-knowledge proof certifying fair training without revealing data or model parameters.

Why fixed-point arithmetic? Garg et al., [GJJZ22] showed that zero-knowledge proofs can certify IEEE-754 floating-point execution by explicitly modeling mantissa alignment, exponent comparison, normalization, and rounding at the bit level. However, each floating-point addition or multiplication expands into hundreds to thousands of Boolean or arithmetic constraints, due to bit decomposition, conditional logic, and normalization circuitry. As a result, the cost of proving a single floating-point operation is orders of magnitude larger than that of a fixed-point addition or multiplication. This overhead compounds in iterative learning workloads, where training involves millions of arithmetic operations across many boosting rounds. In contrast, our fixed-point formulation uses native field arithmetic with explicit truncation, yielding constant-overhead arithmetic operations and enabling scalable certification of gradient-boosted training.

C Details of Fixed-Point XGBoost Training

In this section, we provide further details of our fixed-point XGBoost training algorithm, including the complete training procedure (Procedure 1) and numerical safeguards (Section C.1). The notations and data structures used here are consistent with those in Section 2.1.

Zero-knowledge proof systems operate over finite rings or fields, making floating-point arithmetic impractical as it involves rounding, exponents, and platform-dependent nondeterminism that are difficult to represent succinctly. We therefore redesign XGBoost so that all quantities are represented in *fixed-point arithmetic* with an implicit global scaling factor scale. This section introduces the representation, the proof-friendly approximations to the sigmoid and log-odds functions, and the implications for training correctness.

Fixed-Point Representation. Each real value $x \in \mathbb{R}$ is represented by an integer

$$\tilde{x} = \lfloor x \cdot \mathsf{scale} \rfloor,$$

and all arithmetic is carried out over integers. Addition and subtraction operate directly on these integers, while multiplication and division are defined as:

$$\widetilde{x \cdot y} = \left\lfloor \frac{\tilde{x} \cdot \tilde{y}}{\mathsf{scale}} \right\rfloor, \qquad \frac{\widetilde{x}}{y} = \left\lfloor \frac{\tilde{x} \cdot \mathsf{scale}}{\tilde{y}} \right\rfloor. \tag{4}$$

The division by scale (or multiplication by it) is performed *over integers*, and the result is floored to remain within the integer domain. This keeps every operation deterministic, ensuring that the training relation forms a well-defined NP relation suitable for zero-knowledge proofs. In the following, we omit the tilde notation and treat all variables as fixed-point integers with implicit scaling.

Proof-Friendly Sigmoid and Log-Odds. In XGBoost, the score z in the log-odds space for each boosting round gets converted into probability via sigmoid $\sigma(z) = 1/(1+e^{-z})$, and the base score is conversely initialized by taking the log-odds $\sigma^{-1}(p) = \log(p/(1-p))$. These are not proof-friendly due to the exponential and logarithmic operations, requiring floating-point arithmetic. We replace them with fixed-point variants that approximate these functions accurately in the range $[-2, 2]$:

$$\mathsf{SigmoidWideFP}(z) = \begin{cases} 0, & z \le -2, \\ \dfrac{z+2}{4}, & -2 < z < 2, \\ 1, & z \ge 2, \end{cases} \tag{5}$$

$$\mathsf{LogitFromProbFP}(p) = 2\left(u + \frac{u^3}{3} + \frac{u^5}{5}\right), \quad u := 2p - 1. \tag{6}$$

where all powers and divisions are evaluated according to Eq. (4). Equation (5) is linear within $[-2, 2]$ and saturates outside this range, avoiding costly exponentials. Equation (6) is a truncated Taylor expansion of the $\tanh^{-1}$ function ($\operatorname{atanh}(u) = u + u^3/3 + u^5/5 + \cdots$), providing a smooth and invertible log-odds mapping compatible with integer scaling.

Design Implications. These substitutions render the entire training process discrete and reproducible. All gradient $g_i = p_i - y_i$, Hessian $h_i = p_i(1-p_i)$ for each data point $(\mathbf{x}_i, y_i)$, their (partial) sums G, G_L, G_R, H, H_L, H_R, and gain computations—previously involving floating-point divisions—are now expressed as scaled integer ratios:

$$\mathsf{gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda}\right) - \gamma, \tag{7}$$

where each division follows the fixed-point rule in Eq. (4). Additional numerical safeguards such as feature binning, probability clipping, and bounded leaf weights are applied as in standard XGBoost but are now part of the certified relation itself rather than ad-hoc implementation details.

Empirically (see Section 4), this fixed-point formulation reproduces floating-point XGBoost accuracy within statistical variation (less than 1%), while enabling proof-friendly training semantics suitable for succinct zero-knowledge proofs.

C.1 Numerical Safeguards in Fixed-Point XGBoost

Practical XGBoost implementations incorporate a number of numerical safeguards. Here we make them explicit and they are enforced as part of the certified training process.

Binary label validation. We enforce that each training label satisfies $y_i \in \{0, 1\}$ by checking $y_i \cdot (1 - y_i) = 0$. This prevents malformed inputs that could invalidate gradient and Hessian computations.

Clipping of probabilities and logits. When computing probabilities $p = \sigma(z)$ and their inverse logits $z = \log \frac{p}{1-p}$, we apply explicit clipping. Probabilities are restricted to $[p_{\min}, 1 - p_{\min}]$ (with $p_{\min}$ typically around 10^{-6}), and logits are bounded to $[-L, L]$ where $L = \log \frac{1-p_{\min}}{p_{\min}}$. These clamps serve two roles: (i) they prevent numerical instabilities such as division by zero or $\log(0)$, which would otherwise cause divergence between the implementation and the certificate, and (ii) they align the forward and inverse mappings so that round-trips $\textsc{SigmoidFP}(\textsc{LogitFromProbFP}(p))$ remain consistent under discretization.

Clipping of leaf weights. Each leaf weight is computed as $w = -G/(H + \lambda)$, but then passed through $\textsc{ClipFP}(w, -1, 1)$. This a stability safeguard which avoids extreme updates when H is very small and is also part of the certified relation: the verifier explicitly checks that the prover has applied the same clamp. Without it, it wouldn't be necessary that the certificate would match the training procedure.

Pre-binning of features. Features are first mapped into B equal-width bins, and the training algorithm works only on these integer bin indices. This removes floating-point comparisons on raw feature values from the relation and ensures that every branch decision in the tree is discrete making it easily verifiable in zero-knowledge.

Division safety checks. For every fixed-point division $z = \lfloor x/y \rfloor$, the protocol enforces auxiliary range constraints on x, y, z, and the remainder to prevent wrap-around modulo the field. In particular,

denominators and quotients are required to lie below explicit bounds so that $x = z \cdot y + r$ holds without overflow.

Gain and histogram bounds. Aggregated gradient and Hessian values used in gain computation are range-checked to ensure that squaring and division operations do not overflow the field. These bounds are sufficient to guarantee correctness of split comparisons in the zero-knowledge proof.

Deterministic handling of terminal nodes. When no split yields positive gain, a deterministic terminal split is enforced. This avoids ambiguity in tree structure and ensures that training and certification follow identical control flow.

Procedure 1: TrainXGB

Parameters: trees m, tree height h, learning rate η, regularizers λ, γ, #bins B
Input: Training data $D = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$.
Output: Sequence of trees $T_1, \ldots, T_m$ and base logit z_0.

TrainXGB($\mathcal{D}$):

for $i = 1, \ldots, n$: **assert** $y_i \in \{0, 1\}$
$p \leftarrow (\sum_i y_i)/n$
$\bar{p} \leftarrow \mathsf{ClipFP}(p, p_{\min}, p_{\max})$
$z_0 \leftarrow \mathsf{LogitFromProbFP}(\bar{p})$ // $z_0 \approx \log(\bar{p}/(1-\bar{p}))$
Let $z_{0,i} = z_0$ for all $i \in [n]$
$(\mathsf{edges}, \mathsf{binID}) \leftarrow \mathsf{PreBinFP}(\mathbf{x}, B)$
for $k \in [m]$ **do**
  for $i \in [n]$ **do**
    $p_i \leftarrow \mathsf{SigmoidWideFP}(z_{k-1,i})$ // $p_i \approx \frac{1}{1+\exp(-z_{k-1,i})}$
    $g_i \leftarrow p_i - y_i$
    $h_i \leftarrow p_i(1 - p_i)$
  $T_k \leftarrow (\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k) = (\mathbf{0}, \mathbf{0}, \mathbf{0})$
  $\mathsf{BuildTree}(T_k, 1, \{g_i\}_{i\in[n]}, \{h_i\}_{i\in[n]}, \mathsf{binID}, \mathsf{edges})$
  for $i \in [n]$ **do**
    Let $l_{k,i}$ be the leaf node where $\mathbf{x}_i$ falls in T_k
    $z_{k,i} = z_{k-1,i} - \eta \cdot w_{k,l_{k,i}}$
return $\{T_k\}_{k\in[m]}$ and z_0

BuildTree($T, \ell, \{g_i\}_{i\in\mathcal{I}}, \{h_i\}_{i\in\mathcal{I}}, \mathsf{binID}, \mathsf{edges}$):

if height$(T) = h$ **then**
  $G \leftarrow \sum_{i\in\mathcal{I}} g_i$; $H \leftarrow \sum_{i\in\mathcal{I}} h_i$
  $w' \leftarrow \frac{G}{H+\lambda}$
  $w \leftarrow \mathsf{ClipFP}(w', -1, 1)$
  $T.w_{\ell-(N-1)} \leftarrow w$
else
  $(f^*, b^*, \mathsf{gain}^*) \leftarrow \mathsf{FindSplit}(\{g_i\}_{i\in\mathcal{I}}, \{h_i\}_{i\in\mathcal{I}}, \mathsf{binID})$
  if $\mathsf{gain}^* \le 0$ **then** // pruning via dummy splits; any sample will go right
    $b^* \leftarrow b_{\text{dum}} = 0$
    $f^* \leftarrow f_{\text{dum}}$
  $t^* \leftarrow \mathsf{edges}[f^*][b^*]$
  $\mathcal{L} := \{i \in \mathcal{I} : \mathsf{binID}[i][f^*] < b^*\}$; $\mathcal{R} := \mathcal{I} \setminus \mathcal{L}$
  $\mathsf{BuildTree}(T, 2\ell, \{g_i\}_{i\in\mathcal{L}}, \{h_i\}_{i\in\mathcal{L}}, \mathsf{binID}, \mathsf{edges})$
  $\mathsf{BuildTree}(T, 2\ell+1, \{g_i\}_{i\in\mathcal{R}}, \{h_i\}_{i\in\mathcal{R}}, \mathsf{binID}, \mathsf{edges})$
  $(T.f_\ell, T.t_\ell) \leftarrow (f^*, t^*)$

FindSplit($\{g_i\}_{i\in\mathcal{I}}, \{h_i\}_{i\in\mathcal{I}}, \mathsf{binID}$):

$f^* \leftarrow 0$; $b^* \leftarrow 0$; $\mathsf{gain}^* \leftarrow -\infty$
$G \leftarrow \sum_{i\in\mathcal{I}} g_i$; $H \leftarrow \sum_{i\in\mathcal{I}} h_i$
for $f \in [d]$ **do**
  for $b \in [B]$ **do**
    $\mathcal{L} \leftarrow \{i \in \mathcal{I} : \mathsf{binID}[i][f] < b\}$
    $G_{b,\mathcal{L}} \leftarrow \sum_{i\in\mathcal{L}} g_i$; $H_{b,\mathcal{L}} \leftarrow \sum_{i\in\mathcal{L}} h_i$
    $G_{b,\mathcal{R}} \leftarrow G - G_{b,\mathcal{L}}$; $H_{b,\mathcal{R}} \leftarrow H - H_{b,\mathcal{L}}$

$\mathsf{gain} \leftarrow \frac{1}{2} \cdot \left(\frac{G_{b,\mathcal{L}}^2}{H_{b,\mathcal{L}}+\lambda} + \frac{G_{b,\mathcal{R}}^2}{H_{b,\mathcal{R}}+\lambda} - \frac{G^2}{H+\lambda} \right) - \gamma$
if $\mathsf{gain}^* < \mathsf{gain}$ **then**
$\mathsf{gain}^* \leftarrow \mathsf{gain}$; $f^* \leftarrow f$; $b^* \leftarrow b$
return the best split $(f^*, b^*, \mathsf{gain}^*)$

D Details of Certification Algorithm for XGBoost Training

In this section, we provide the detailed procedure for the certification that verifies the correctness of XGBoost training. The procedure is summarized in Procedure 2. The algorithm takes as input a training data $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ with labels $\mathbf{y} = (y_1, \ldots, y_n)$, a fitted tree ensemble $\mathcal{T} = \{T_k\}_{k\in[m]}$, where each tree T_k is represented by $(\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k)$, and a base logit z_0. It then verifies that $(\mathcal{T}, z_0)$ is the output of the fixed-point XGBoost training algorithm TrainXGB (see Appendix C) on input $(\mathbf{x}, \mathbf{y})$.

Claim 1. *The certificate algorithm* CertXGB *verifies the correctness of XGBoost training. That is, for any* $(\mathbf{x}, \mathbf{y}), \mathcal{T} = \{T_k\}_{k\in[m]}, z_0$, $\mathsf{CertXGB}(\mathbf{x}, \mathbf{y}, \mathcal{T}, z_0) = 1$ *if and only if* $(\mathcal{T}, z_0) = \mathsf{TrainXGB}(\mathbf{x}, \mathbf{y})$.

Proof. **(If)** If $(\mathcal{T}, z_0) = \mathsf{TrainXGB}(\mathbf{x}, \mathbf{y})$, then all the assertions in the subprotocols clearly hold, and thus CertXGB returns 1.

(Only if) For fixed $(\mathbf{x}, \mathbf{y}, \mathcal{T}, z_0)$, suppose $(\mathcal{T} = (\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k)_{k=1}^m, z_0) \neq (\mathcal{T}' = (\mathbf{f}'_k, \mathbf{t}'_k, \mathbf{w}'_k)_{k=1}^m, z'_0) = \mathsf{TrainXGB}(\mathbf{x}, \mathbf{y})$. We show that $\mathsf{CertXGB}(\mathbf{x}, \mathbf{y}, \mathcal{T}, z_0) = 0$.

If $z_0 \neq z'_0$: Since ValidateLogit computes the logit from $\mathbf{y}$ as in TrainXGB, CertXGB derives the same z'_0 from $\mathbf{y}$. Thus, it holds that $z_0 \neq z'_0$, and thus CertXGB returns 0.

If $T_1 \neq T'_1$ and $(\mathbf{f}_1, \mathbf{t}_1) \neq (\mathbf{f}'_1, \mathbf{t}'_1)$ (i.e., the first tree contains an incorrect split): Since CertXGB runs InitHists, which computes p_i, g_i, and h_i from $z_{i,0} = z_0$ for $i = 1, \ldots, n$ as in TrainXGB, they are the same as those in TrainXGB. InitHists also aggregates them into root histograms from leaf histograms. Thus, every possible split and gain computed inside ValidateSplits for the root are the same as those in FindSplit of TrainXGB.

We first consider the case where $(f_{1,1}, t_{1,1}) \neq (f'_{1,1}, t'_{1,1})$, i.e., the split at the root node is incorrect. In CertXGB, it computes $(f^*_{1,1}, t^*_{1,1})$ and the corresponding max gain from the root histograms, which are the same as those in TrainXGB. It runs ValidateSplits, which checks either of the following cases: (1) if gain^* is non-positive, then it checks whether $(f_{1,1}, t_{1,1})$ has dummies; (2) else, it checks whether $(f_{1,1}, t_{1,1}) = (f^*_{1,1}, t^*_{1,1})$. In the former case, TrainXGB would also set dummies for $(f'_{1,1}, t'_{1,1})$ and thus the assertion by ValidateSplits fails. In the latter case, since TrainXGB chooses the optimal split $(f^*_{1,1}, t^*_{1,1})$ which is different from $(f_{1,1}, t_{1,1})$, the assertion by ValidateSplits also fails. Note that ValidateSplits also checks that $(f^*_{1,1}, b^*_{1,1})$ is the smallest among those achieving the max gain, which also holds since TrainXGB chooses the smallest one in case of ties.

Now, we consider the case where $(f_{1,1}, t_{1,1}) = (f'_{1,1}, t'_{1,1})$, i.e., the split at the root node is correct, but some other split in T_1 is incorrect. By ValidateInference, the samples reaching left and right child nodes of the root in CertXGB are the same as those in TrainXGB. Thus, the histograms at these child nodes computed by InitHists are also the same as those in TrainXGB. Then, we can apply the same argument as above to either of the child nodes where the split is incorrect, and show that assertion by ValidateSplits fails in one of the second-level nodes. Applying this argument recursively, CertXGB always detects the incorrect split in T_1.

If $T_1 \neq T'_1$ and $(\mathbf{f}_1, \mathbf{t}_1) = (\mathbf{f}'_1, \mathbf{t}'_1)$ but $\mathbf{w}_1 \neq \mathbf{w}'_1$ (i.e., the first tree contains an incorrect leaf weight): Since the splits are correct, by ValidateInference, the samples reaching each leaf in CertXGB are the same as those in TrainXGB. Thus, the histograms at these leaves computed by InitHists are also the same as those in TrainXGB. Then, ValidateLeafWeights checks whether the leaf weights are correctly computed from these histograms as in TrainXGB, and thus the assertion fails for the incorrect leaf weight.

$T_1 = T'_1$ but for some $k \leq m$, $T_k \neq T'_k$ (i.e., one of the subsequent trees is incorrect): Since CertXGB correctly computes the updated logits $\mathbf{z}_1$ after training T_1, which are the same as those in TrainXGB, we can apply the same arguments as above iteratively to T_k for $k = 2, \ldots, m$ and show that CertXGB detects the incorrect tree.

Therefore, by induction, for each $k \in [m]$, T_k is correctly trained on $(\mathbf{x}, \mathbf{y})$ with the initial scores $\mathbf{z}_{k-1}$, and the updated scores $\mathbf{z}_k$ are correctly computed. This implies that $(\mathcal{T}, z_0) = \mathsf{TrainXGB}(\mathbf{x}, \mathbf{y})$. □

Procedure 2: CertXGB

Parameters: Number of points n, features d, trees m, bins B, leaves $N = 2^h$ at depth h, learning rate η, regularizers λ, γ.
Input: Fixed-point training data $\mathcal{D} = (\mathbf{x}, \mathbf{y})$, the fitted tree ensemble $\mathcal{T} = \{T_k\}_{k\in[m]}$, and base logit z_0.
Output: 1 (accept) or 0 (reject).

for $i = 1, \ldots, n$: **assert** $y_i \in \{0, 1\}$
$(\mathsf{edges}, \mathsf{binID}) \leftarrow \mathsf{PreBinFP}(\mathbf{x}, B)$
assert $\mathsf{ValidateLogit}(\mathbf{y}, z_0) = 1$
$\mathbf{z}_0 \leftarrow (z_{0,i})_{i=1}^n$ with $z_{0,i} = z_0$
for each tree $k \in [m]$ **do** // Initialize intermediate scores
  for each data point $i \in [n]$ **do**
    compute the reached leaf $l_{k,i} \in [N]$ by evaluating T_k on $\mathbf{x}_i$
    $z_{k,i} \leftarrow z_{k-1,i} - \eta \cdot w_{k,l_{k,i}}$
  $\mathbf{z}_k \leftarrow (z_{k,i})_{i=1}^n$
  $\mathbf{l}_k \leftarrow (l_{k,i})_{i=1}^n$
for each tree $k \in [m]$ **do** // Tree validation
  parse $(\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k) \leftarrow T_k$
  assert $\mathsf{ValidateInference}(\mathbf{x}, \mathbf{f}_k, \mathbf{t}_k, \mathbf{l}_k) = 1$
  $(G_k, H_k) \leftarrow \mathsf{InitHists}(\mathbf{z}_{k-1}, \mathbf{y}, \mathsf{binID}, \mathbf{l}_k)$
  for each internal node $\ell \in [N-1]$ **do**
    compute $(f^*_{k,\ell}, b^*_{k,\ell})$ such that gain derived from (G_k, H_k) is maximized
    $t^*_{k,\ell} \leftarrow \mathsf{edges}[f^*_{k,\ell}][b^*_{k,\ell}]$
  $(\mathbf{f}^*_k, \mathbf{t}^*_k) \leftarrow ((f^*_{k,\ell}, t^*_{k,\ell}))_{\ell=1}^{N-1}$
  assert $\mathsf{ValidateLeafWeights}(G_k, H_k, \mathbf{w}_k) = 1$
  assert $\mathsf{ValidateSplits}(G_k, H_k, \mathbf{f}_k, \mathbf{t}_k, \mathbf{f}^*_k, \mathbf{t}^*_k, \mathsf{edges}) = 1$
return 1 (accept)

Procedure 3: PreBinFP

Input: Fixed-point feature matrix $\mathbf{x} = (x_{i,j})_{i\in[n],\, j\in[d]}$, number of bins B
Output: $\mathsf{edges} \in \mathbb{N}^d \times \mathbb{R}^{B+1}$, and $\mathsf{binID} \in [B]^{n\times d}$

for $f = 1$ **to** d **do**
  $c_{min} \leftarrow \min_{i\in[n]} x_{i,f}$; $\quad c_{max} \leftarrow \max_{i\in[n]} x_{i,f}$
  $\delta_f \leftarrow \dfrac{c_{max} - c_{min}}{B}$
  Define a lookup table $\mathsf{edges}[f] \leftarrow (c_{min} + (b-1)\cdot\delta_f)_{b=1}^{B}$ consisting of left edges of B equal-width bins.
  $\mathsf{edges}[f_{\text{dum}}][b_{\text{dum}}] \leftarrow t_{\text{dum}}$ // define dummy (small enough) edge for pruning, e.g., $t_{\text{dum}} = \texttt{DBL_MIN}$
  for $i = 1, \ldots, n$:

$$\mathsf{binID}[i][f] \leftarrow \sum_{b=1}^{B} \mathbf{1}\{\mathsf{edges}[f][b] \le x_{i,f}\}$$

return $(\mathsf{edges}, \mathsf{binID})$

Procedure 4: ValidateLogit

Parameters: $p_{\min}, p_{\max}$ for clipping probabilities
Input: Labels $\mathbf{y} = (y_i)_{i\in[n]}$ and base logit z_0
Output: Accept/Reject

$p \leftarrow \frac{1}{n}\sum_{i=1}^n y_i$
$p' \leftarrow \mathsf{ClipFP}(p, p_{\min}, p_{\max})$
$u \leftarrow 2p' - 1$

$z_0' \leftarrow 2 \cdot \left(u + \frac{u^3}{3} + \frac{u^5}{5}\right)$ // truncated atanh series with fixed-point ops
assert $z_0 = z_0'$

Procedure 5: ValidateInference

Input: Data points $\mathbf{x} = (x_{i,f})_{i\in[n],f\in[d]}$, node features $\mathbf{f}$, node thresholds $\mathbf{t}$, and leaf indices $\mathbf{l} = (l_i)_{i=1}^n$ reached by each $\mathbf{x}_i$
Output: Accept/Reject

Define a table of root-to-leaf paths $\mathsf{path}[l] = (f_{l,j}, t_{l,j})_{j=1}^h$ for each $l \in [N]$
for $i = 1$ **to** n **do**
    (Path lookup) Retrieve $\mathsf{path}[l_i] = (f_{l_i,j}, t_{l_i,j})_{j=1}^h$.
    (Batched permutation) Define a permutation $(\bar{x}_{i,1}, \ldots, \bar{x}_{i,d})$ of $(x_{i,1}, \ldots, x_{i,d})$ such that the first h entries align with the path features $(f_{l_i,1}, \ldots, f_{l_i,h})$; **assert** permutation correctness.
    (Branch decisions) For $j = 1, \ldots, h$, set $a_j \leftarrow \mathbf{1}\{\, t_{l_i,j} \le \bar{x}_{i,j} \,\}$ and **assert**:

$$l_i + 2^h - 1 \;=\; 2^h + \sum_{j=1}^{h} a_j \cdot 2^{h-j}.$$

Procedure 6: InitHists

Input: Scores $\mathbf{z} = (z_i)_{i=1}^n$, labels $\mathbf{y} = (y_i)_{i=1}^n$, binID, leaf indices $\mathbf{l} = (l_i)_{i=1}^n$
Output: Histograms (G, H)

for $f \in [d]$, $\ell \in [2N-1]$, $b \in [B]$, initialize $G[f][\ell][b] \leftarrow 0$, $H[f][\ell][b] \leftarrow 0$.
for $i = 1$ **to** n **do**
    $p_i \leftarrow \mathsf{SigmoidWideFP}(z_i)$; $g_i \leftarrow p_i - y_i$; $h_i \leftarrow p_i \cdot (1 - p_i)$
    for $f = 1$ **to** d **do**
        $b \leftarrow \mathsf{binID}[i][f]$; $\ell \leftarrow l_i + (N-1)$
        $G[f][\ell][b] \leftarrow G[f][\ell][b] + g_i$; $H[f][\ell][b] \leftarrow H[f][\ell][b] + h_i$
for $\ell = N-1$ **down to** 1 **do** // propagate up to root
    for $f = 1$ **to** d **do**
        for $b = 1$ **to** B **do**
            $G[f][\ell][b] \leftarrow G[f][2\ell][b] + G[f][2\ell+1][b]$
            $H[f][\ell][b] \leftarrow H[f][2\ell][b] + H[f][2\ell+1][b]$
return (G, H)

Procedure 7: SigmoidWideFP

Input: Fixed-point logit
Output: $y \in \{0, \ldots, \mathsf{scale}\}$

if $z \le -2$ **then**
    return 0
else if $z \ge 2$ **then**
    return 1
else
    $y \leftarrow \dfrac{z+2}{4}$ // integer arithmetic only
    return y

Procedure 8: ValidateLeafWeights

Input: Leaf histograms (G, H), weights $\mathbf{w} = (w_l)_{l=1}^N$
Output: Accept/Reject

for $l = 1$ **to** N **do**
    $\ell \leftarrow l + (N-1)$

$G^{\text{leaf}} \leftarrow \sum_{b=1}^{B} G[1][\ell][b]$; $H^{\text{leaf}} \leftarrow \sum_{b=1}^{B} H[1][\ell][b]$ // use any feature f, as the sum is feature-independent

$w'_l \leftarrow \dfrac{G^{\text{leaf}}}{H^{\text{leaf}} + \lambda}$

assert $w_l = \mathsf{ClipFP}(w'_l, -1, 1)$

Procedure 9: ValidateSplits

Input: Node histograms (G, H), resulting splits $(\mathbf{f}, \mathbf{t}) = (f_\ell, t_\ell)_{\ell=1}^{N-1}$, indices $(\mathbf{f}^*, \mathbf{t}^*) = (f_\ell^*, t_\ell^*)_{\ell=1}^{N-1}$ leading to maximum gain, and edges. Let $e_0 = 0$.
Output: Accept/Reject

for $\ell = 1$ **to** $N-1$ **do**
  for $f = 1$ **to** d **do**
    $G \leftarrow \sum_{b=1}^{B} G[f][\ell][b]$; $H \leftarrow \sum_{b=1}^{B} H[f][\ell][b]$
    $G_L \leftarrow 0$, $H_L \leftarrow 0$
    for $b = 1$ **to** B **do**
      $G_L \leftarrow G_L + G[f][\ell][b]$; $H_L \leftarrow H_L + H[f][\ell][b]$
      $G_R \leftarrow G - G_L$; $H_R \leftarrow H - H_L$
      $\mathsf{gain}[f][b] \leftarrow \frac{1}{2} \cdot \left(\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda} \right) - \gamma$ // fixed-point divisions
  Retrieve b_ℓ^* s.t. $t_\ell^* = \mathsf{edges}[f_\ell^*][b_\ell^*]$
  Retrieve $\mathsf{gain}^* \leftarrow \mathsf{gain}[f_\ell^*][b_\ell^*]$
  assert $(\mathsf{gain}^* > \mathsf{gain}[f][b]) \vee (\mathsf{gain}^* = \mathsf{gain}[f][b] \wedge f_\ell^* \leq f \wedge b_\ell^* \leq b)$ for all $f \in [d], b \in [B]$ // tie-breaker check
  $e_\ell \leftarrow (\mathsf{gain}^* \leq 0 \vee e_{parent})$ // check if the current node should be terminal, or the upper-level node is terminal
  assert $(1 - e_\ell) \cdot (f_\ell = f_\ell^* \wedge t_\ell = t_\ell^*) \vee e_\ell \cdot (f_\ell = f_{\text{dum}} \wedge t_\ell = t_{\text{dum}}) = 1$ // if not dummy, use max gain split; else, use terminal split

Procedure 10: ClipFP

Input: x, lower bound a, upper bound b with $a \leq b$ represented as fixed-point numbers
Output: Clipped value y.

$y \leftarrow x$
if $y < a$ **then**
  $y \leftarrow a$
if $y > b$ **then**
  $y \leftarrow b$
return y

E Additional Details on ZK XGBoost-FP

In this section, we provide detailed explanation on the proof of comparison, division, and truncation, as well as additional components in proving XGBoost-FP shown in Figure 2.

E.1 Proof of Comparison

Our proof of comparison builds upon the idea of [HCL+24] to prove the comparison relation by bits-decomposition. It first converts the proof of comparison into a MSB proof $\mathbf{1}\{[\![x]\!] < [\![y]\!]\} = \mathsf{MSB}([\![x - y]\!])$ due to the signed representation. This is true because $\mathsf{MSB}([\![x - y]\!]) = 1$ if $x - y < 0$ and $\mathsf{MSB}([\![x - y]\!]) = 0$ otherwise. We shift our focus into proving $[\![s]\!] = \mathsf{MSB}([\![z]\!])$ for committed $([\![z]\!], [\![s]\!])$. To ensure the soundness of MSB proof, it requires a *one-bit gap* between the scale of encoded signed number and the underlying field: for a finite field $\mathbb{F}_p$, we require $x, y \in [0, \lfloor p/4 \rfloor) \cup [p - \lfloor p/4 \rfloor, p - 1]$. Otherwise, the subtraction may cause an overflow and lead to an incorrect result. Existing approaches all somehow require the $\mathcal{P}$ providing the rest bits of z and proving the correctness of bit-decomposition.

Bit-decomposition. Assume that a $\mathcal{P}$ commits to $(\llbracket z \rrbracket, \llbracket s \rrbracket)$ and needs to prove that $\llbracket s \rrbracket = \mathsf{MSB}(\llbracket z \rrbracket)$. Define $n, \mathbb{F}_p$ s.t. $\log |\mathbb{F}_p| \leq n$, a naive way to prove the sign is to apply a bit-decomposition as in [WYX+21b]. However, it incurs $O(n)$ communication cost to commit to all n bits $(z_0, \ldots, z_{n-1})$ and need to prove an additional binary adder circuit. [HCL+24]'s approach only incurs $O(n/d)$ for an arbitrarily defined d by leveraging the proof of table lookup [FKL+21, YH24, Hab22].

Without loss of generality, we assume (d, t) such that $n - 1 = d \cdot t$. The idea is to group every consecutive d bits of z into $(\tilde{z}_0, \ldots, \tilde{z}_{t-1})$, in which $\tilde{z}_i = \sum_{j=0}^{t} 2^j \cdot z_{id+j}$. In this case, $\mathcal{P}$ only needs to commit to $t = (n-1)/d + 1$ elements and prove that $z = 2^{n-1} \cdot s + \sum_{i=0}^{t-1} 2^{id} \cdot \tilde{z}_i$. Additionally, $\mathcal{P}$ needs to show that each $\tilde{z}_i \in [0, 2^d - 1]$ via a membership proof from a list $T = (0, 1, \ldots, 2^d - 1)$, which can be instantiated by table lookup with only $O(1)$ cost per access [YH24].

Preventing Malicious Prover. A soundness issue with the above scheme is that a cheating $\mathcal{P}$ may commit to incorrect $(\tilde{z}_0, \ldots, \tilde{z}_{t-1}, s)$ such that $z + p = 2^{n-1} \cdot s + \sum_{i=0}^{t-1} 2^{id} \cdot \tilde{z}_i$. For example, assume that $p = 2^{61} - 1$ and $z = 0$, a cheating $\mathcal{P}$ can commit to $s = 1$ and all $\tilde{z}_i = 2^d - 1$, and prove that $z = p \bmod p = 0$. Since $\bmod p$ happens implicitly in ZK operations, extra measures are needed to detect such wrap-around. [HCL+24] attempted to address this issue but its approach doubles the cost of secure comparison.

Our solution employs a Mersenne prime in the form of $p = 2^n - 1$, and rules out the only cheating case when $z = 0$, $s = 1$ and $\tilde{z}_i = 2^d - 1$ for all $i \in [0, t)$. By defining $\llbracket w \rrbracket = \mathbf{1}\{\llbracket z \rrbracket \neq 0\}$, we observe that $\mathsf{MSB}(z) = \begin{cases} 0 & \text{if } z = 0, w = 0 \\ s & \text{if } z \neq 0, w = 1 \end{cases}$. It implies that $(1 - w) \cdot s = 0$ should always be true. Additionally, we employ the non-equality-zero check from [PHGR13] to prove $\llbracket w \rrbracket = \mathbf{1}\{\llbracket z \rrbracket \neq 0\}$. Our solution to address the soundness issue only requires committing 2 extra values and proving 3 multiplicative relations, which incurs less overhead than [HCL+24]. We defer the formal algorithm description to Figure 2 in Appendix E.

Cost Analysis. The proof of comparison requires committing to $3 + n/d$ witnesses, performing n/d lookup on a table of size 2^d, and proving 3 multiplicative relations. The proof for table lookup generally takes $O(1)$ per access. Its $O(2^d)$ setup cost will be amortized since the table is reused across the whole proof [YH24, Hab22].

Protocol 2: ZKP Gadgets

Check Nonzero. On input $\llbracket x \rrbracket, \llbracket y \rrbracket$, prove that $\llbracket y \rrbracket = \mathbf{1}\{\llbracket x \rrbracket \neq 0\}$.

1. If $x = 0$, $\mathcal{P}$ commits to $\llbracket u \rrbracket$ such that $u = 0$. Otherwise, it commits to $u = x^{-1}$.
2. Prove that $\llbracket y \rrbracket - \llbracket u \rrbracket \cdot \llbracket x \rrbracket = 0$ and $(1 - \llbracket y \rrbracket) \cdot \llbracket x \rrbracket = 0$.

Comparison. On input $\llbracket x \rrbracket, \llbracket y \rrbracket, \llbracket s \rrbracket$, prove that $s = \mathbf{1}\{\llbracket x \rrbracket < \llbracket y \rrbracket\}$. Define parameters (d, t) such that $n - 1 = d \cdot t$. Construct a public lookup table $\mathcal{T} = (0, \ldots, 2^d - 1)$.

1. Commits to $\llbracket z \rrbracket = \llbracket x \rrbracket - \llbracket y \rrbracket$ and $\llbracket s \rrbracket = \mathsf{MSB}(\llbracket z \rrbracket)$. Decompose z into $(\llbracket \tilde{z}_0 \rrbracket, \ldots, \llbracket \tilde{z}_{t-1} \rrbracket)$ such that $\tilde{z}_i = \sum_{j=0}^{t} 2^j \cdot z_{id+j}$.
2. Prove that for $i \in [0, t)$, $\tilde{z}_i = \mathcal{T}[\tilde{z}_i]$.
3. Prove that $\llbracket z \rrbracket = 2^{n-1} \cdot \llbracket s \rrbracket + \sum_{i=0}^{t-1} 2^{id} \cdot \llbracket \tilde{z}_i \rrbracket$.
4. Commit to $\llbracket w \rrbracket = \mathbf{1}\{\llbracket z \rrbracket \neq 0\}$ and prove its correctness using the above nonzero check. Prove that $(1 - \llbracket w \rrbracket) \cdot \llbracket s \rrbracket = 0$.

Division. On input $\llbracket x \rrbracket, \llbracket y \rrbracket, \llbracket z \rrbracket$, prove that $z = \lfloor x/y \rfloor$. Define m_q and m_d as the upper bound of the abstract value of the quotient and denominator.

1. $\mathcal{P}$ commits to abstract values $\llbracket \bar{x} \rrbracket, \llbracket \bar{y} \rrbracket, \llbracket \bar{z} \rrbracket$ and prove that $\bar{i} = (1 - 2 \cdot \mathsf{MSB}(i)) \cdot i$ for $i \in \{x, y, z\}$.
2. $\mathcal{P}$ commits to the residual $\llbracket \bar{r} \rrbracket$ and prove that $\llbracket \bar{x} \rrbracket = \llbracket \bar{y} \rrbracket \cdot \llbracket \bar{z} \rrbracket + \llbracket \bar{r} \rrbracket$.
3. Prove that $0 \leq \llbracket \bar{r} \rrbracket < \llbracket \bar{y} \rrbracket$, $\llbracket \bar{y} \rrbracket \leq m_d$ and $\llbracket \bar{z} \rrbracket < m_q$.
4. Prove that $\mathsf{MSB}(z) = \mathsf{MSB}(x) \oplus \mathsf{MSB}(y)$ using the fact that for any binary values α, β, $\alpha \oplus \beta = \alpha + \beta - 2\alpha \cdot \beta$.

Truncation. On input $\llbracket x \rrbracket, \llbracket z \rrbracket, f$, prove that $z = \lfloor x/2^f \rfloor$. The proof is the same as the division except that $y = 2^f$ is public so that its range check is avoided. Additionally, z is bounded by $m_q = \lfloor p/2^{f+1} \rfloor$. If p is a Mersenne prime, the check of $0 \leq \llbracket r \rrbracket < 2^f$ and $z < m^q$ can both be done by bit-decomposition proofs.

E.2 Proof of Division and Truncation

Division. Proof of division takes inputs $([\![x]\!], [\![y]\!], [\![z]\!])$ and proves that $z = \lfloor x/y \rfloor$. Note that this statement disallows directly proving it by $x = z \cdot y$ because of the floor operation. To simplify the problem, we assume that both (x, y) represent positive values. To generalize it to arbitrary inputs, we can first prove their absolute values and then determine the sign of z based on the signs of x, y. It can also be trivially generalized for fixed-point representations by left-shift x before the division (assume that the left shift does not overflow).

In our XGBoost-FP, the numerators and denominators may range from 0 to nearly $p/2^f$. Previous proofs of division are not suitable since they only work for bounded small values [LXZ21, PP24]. Another approach proves the floating-point division but requires thousands of constraints [WYX+21a].

To prove the relation, our approach leverages the existence of a residue $r \in [0, y)$ such that $x = z \cdot y + r$. Additionally, two range checks are required to maintain the soundness: $r \in [0, y)$, and $z \in [0, \lfloor p/y \rfloor]$. The first check ensures r is a residue. The second check is needed to prevent the similar wrap-around issue happened in the proof of MSB. Namely, there could be many possible pairs of (z', r') such that $x = z' \cdot y + r' \bmod p$ is z'. The proof should ensure a z that satisfies $x = z \cdot y + r$ without modulo p.

In our work, we define upper bounds for the positive denominator y and quotient z, denoted as m_d and m_q, such that $m_d \cdot m_q < p/2$. $\mathcal{P}$ proves that $[\![y]\!] < m_d$, $[\![r]\!] < [\![y]\!]$ and $[\![z]\!] < m_q$. This is not necessarily sufficient for general proof of division, but is sufficient for our XGBoost-FP since with a relatively large p, these values are expected to lie within a reasonable range.

Truncation. The truncation is a special type of division with public and positive denominators $y = 2^f$. It follows similar ideas as in the above proof of division to prove the range of r and z. However, its proof is much simpler because f is usually small so that the range check of $r \in [0, 2^f)$ can be implemented by a table lookup. Also, the Mersenne prime $p = 2^n - 1$ results in $\lfloor (p-1)/2^f \rfloor = 2^{n-f} - 1$, which also enables a range check of $z \in [0, 2^{n-f} - 1]$ to be resolved by our bit-decomposition proof.

We defer the formal algorithm description for division and truncation to Figure 2 in Appendix E.

Cost Analysis. The division proof invokes 7 comparison/MSB and proves 4 multiplicative relation. The truncation is simplified to 2 MSB proof, 1 bit-decomposition proof, 1 table lookup, and 2 multiplications. The concrete overhead depends on the underlying proof system and the construction of lookup table.

E.3 Additional Components

Input validation. Our model training requires the global truth $\boldsymbol{y}$ to be binary values. Hence, a check is performed on witnesses to ensure that $y_i \cdot (1 - y_i) = 0$ for $i \in [n]$. Our protocol also supports arbitrary input validation on input dataset $\boldsymbol{x}$ depending on the dataset type.

Global Range Checks. In principle, the proof of numerical computation loses soundness whenever the overflow occurs. This implies that a range check is needed for every intermediate values that are committed during the proof. This can be realized by the proof of comparison mentioned above. To reduce the overhead for range checks, we examine the XGBoost proof and find out that most of intermediate values are already bounded because of the checks that we applied in the proof of comparison, division, and truncation. Additional range checks are only required in a few places.

Essentially, the range check is required when validating the leaf weights and the splits of internal nodes. The former computes $w \leftarrow G/(H + \lambda)$. To respect the fixed-point representation, the proof in fact validates $(G \cdot \mathsf{scale})/(H + \lambda)$ so that the result w is lifted by scale. Hence, the value $|G|$ should be restricted in $[0, p/(2 \cdot \mathsf{scale}))$. Similarly, we compute $G^2/(H + \lambda)$ when validating the split. Since G^2 is already lifted by scale^2, it can be directly fed into the proof of division without a truncation. However, we still need to validate that $G \in [-\sqrt{p/2}, \sqrt{p/2})$ to prevent the overflow when proving G^2.

In some situations, the checks can be avoided if the input to specific operations are already bounded. When validating base logit z'_0, the approximation of atanh is guaranteed not to overflow since $p \leftarrow \sum_{i=1}^{n} y_i/n < 1$, due to y_i being binary values. In validating the hessian $h_i \leftarrow p_i \cdot (1 - p_i)$, the overflow is also not needed because p_i is an output of the sigmoid function.

Public Commitment and Proof of Opening. Our framework also supports the commit-and-prove ZKP such that $\mathcal{P}$ can first publicly commit to the dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, model parameters T_k, and other auxiliary information, and later prove their consistency in ZKP. The public commitment

can be transmitted to a public bulletin board or corresponding verifiers anytime before ZKP. This can be realized by [WYX+21b, SLY+25, CFQ19].

E.4 Security Analysis

We provide the security analysis for Theorem 1. We assume that $\mathcal{F}_{\mathsf{ZK}}$ is instantiated by [YH24, Hab22] and $\mathcal{F}_{\mathsf{CVOLE}}$ is instantiated by [SLY+25]. The secure realization of these functionalities can be referred to the cited works. We show Functionalities $\mathcal{F}_{\mathsf{CVOLE}}$ and $\mathcal{F}_{\mathsf{ZK}}$ below and reinstate Theorem 1.

Functionality 2: $\mathcal{F}_{\mathsf{CVOLE}}$

The functionality interacts with three parties: a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and an ideal adversary $\mathcal{S}$. It is parameterized by a secret $\boldsymbol{x}$ and its commitment com_x.

1. Upon receiving init from both parties, and additional input $\boldsymbol{x}$ and decommitment message decom_x from $\mathcal{P}$, check whether com_x correctly decommits to $\boldsymbol{x}$.
2. Upon receiving cvole from both parties, sample a VOLE correlation $\boldsymbol{m} = \boldsymbol{k} + \boldsymbol{x} \cdot \Delta$. Send $(\boldsymbol{m})$ to $\mathcal{P}$ and $(\boldsymbol{k}, \Delta)$ to $\mathcal{V}$. If $\mathcal{P}$ is corrupted, receive $(\boldsymbol{m})$ from $\mathcal{S}$ and recompute $\boldsymbol{k} = \boldsymbol{m} - \boldsymbol{x} \cdot \Delta$. If $\mathcal{V}$ is corrupted, receive $(\boldsymbol{k}, \Delta)$ from $\mathcal{V}$ and recompute $\boldsymbol{m} = \boldsymbol{k} + \boldsymbol{x} \cdot \Delta$. Send these values to corresponding parties.

Functionality 3: $\mathcal{F}_{\mathsf{ZK}}$

The functionality interacts with three parties: a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and an ideal adversary $\mathcal{S}$. It takes input a circuit C that consists of arithmetic components and read-only or read-write RAM components. Upon receiving $(\mathsf{prove}, C, \boldsymbol{x}, \boldsymbol{w})$ from $\mathcal{P}$ and $(\mathsf{verify}, C, \boldsymbol{x})$, it verifies $C(\boldsymbol{x}, \boldsymbol{w})$ by traversing the circuit by topological order and check the relations.

1. For arithmetic components, it verifies arithmetic relations. If any relation is not satisfied by $\boldsymbol{x}, \boldsymbol{w}$, it aborts.
2. For RAM components, it initializes the memory using data from $\boldsymbol{x}$ or $\boldsymbol{w}$. It checks the data output by read accesses, and modifies the table according to write accesses. If any access operation is inconsistent, it aborts.

Theorem 1. *Define the relation by the proof of fixed-point XGBoost Training described in R_{xgb} (1), Protocol 1 securely realizes $\mathcal{F}_{\mathsf{CP}}$ in the $(\mathcal{F}_{\mathsf{CVOLE}}, \mathcal{F}_{\mathsf{ZK}})$-hybrid model.*

Completeness. In the case of a pair of honest $\mathcal{P}$ and $\mathcal{P}$, the completeness is obvious since $\mathcal{P}$ honestly trains the model from the committed dataset and commits to the extended witness. Assume that the prover trains the model and proves ZK XGBoost both using fixed-point computation, the proof passes with probability 1.

Soundness. Our protocol makes black-box access to the underlying $\mathcal{F}_{\mathsf{ZK}}$ and $\mathcal{F}_{\mathsf{CVOLE}}$. Hence, a corrupted $\mathcal{P}$'s view can be simulated by constructing a simulator $\mathcal{S}$ who emulates the functionalities $\mathcal{F}_{\mathsf{ZK}}$ and $\mathcal{F}_{\mathsf{CVOLE}}$. In $\mathcal{F}_{\mathsf{CVOLE}}$, it receives $\mathcal{D} = (\mathbf{x}, \mathbf{y})$, $\mathcal{T} = \{T_k\}_{k\in[m]}$ with $T_k = (\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k)$, z, and their commitments from $\mathcal{P}$ and relay them to $\mathcal{F}_{\mathsf{CP}}$. It also check whether the commitments successfully decommits to these values. If $\mathcal{F}_{\mathsf{CVOLE}}$ aborts, it sends abort to $\mathcal{F}_{\mathsf{CP}}$ and aborts itself. It receives IT-MACs from corrupted $\mathcal{P}$ and samples a correct VOLE correlation for committed data. It emulates $\mathcal{F}_{\mathsf{ZK}}$ with $\mathcal{P}$'s input and the XGBoost-FP relation, and $\mathcal{F}_{\mathsf{ZK}}$ checks whether the XGBoost-FP relation is satisfied. If not, it sends abort to $\mathcal{F}_{\mathsf{CP}}$ and aborts itself. Otherwise, it sends $\mathsf{PROVE-RECEIPT}$.

Zero-Knowledge. $\mathcal{V}$ has no input to $\mathcal{F}_{\mathsf{CP}}$ so the emulation of $\mathcal{V}$'s view is relatively simple. We construct a simulator $\mathcal{S}$, who emulates $\mathcal{F}_{\mathsf{ZK}}$ and $\mathcal{F}_{\mathsf{CVOLE}}$, and interact with corrupted $\mathcal{V}$ accordingly. It relays whatever it receives from $\mathcal{F}_{\mathsf{CP}}$ to corrupted $\mathcal{V}$ and aborts when it aborts.

F Details of Certificate for Random Forest Training

In this section, we provide the detailed procedure for the certificate that verifies the correctness of random forest training. The procedure is summarized in Procedure 11. The algorithm takes as input a training data $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ with labels $\mathbf{y} = (y_1, \ldots, y_n)$, a fitted forest $\mathcal{T} = \{T_k\}_{k\in[m]}$, where each tree T_k is represented by $(\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k)$.

Procedure 11: CertForest

Parameters: Number of points n, features d, trees m, bins B, leaves $N = 2^h$ at depth h.
Input: Fixed-point training data $(\mathbf{x}, \mathbf{y})$, the fitted forest $\mathcal{T} = \{T_k\}_{k\in[m]}$, index sets $\{I_k\}_{k\in[m]}$, where $I_k \subset [n]$.
Output: 1 (accept) or 0 (reject).

$(\mathsf{edges}, \mathsf{binID}) \leftarrow \mathsf{PreBinFP}(\mathbf{x}, B)$
for each tree $k \in [m]$ **do**
  parse $(\mathbf{f}_k, \mathbf{t}_k, \mathbf{w}_k) \leftarrow T_k$
  for each data point $i \in I_k$ **do**
    compute the reached leaf $l_{k,i} \in [N]$ by evaluating T_k on $\mathbf{x}_i$
  $\mathbf{l}_k \leftarrow (l_{k,i})_{i\in I_k}$
  assert $\mathsf{ValidateInference}((\mathbf{x}_i)_{i\in I_k}, \mathbf{f}_k, \mathbf{t}_k, \mathbf{l}_k) = 1$
  $H_k \leftarrow \mathsf{InitHistsLabel}((y_i)_{i\in I_k}, \mathsf{binID}, \mathbf{l}_k)$
  assert $\mathsf{ValidateLeafWeightsLabel}(H_k, \mathbf{w}_k) = 1$
  assert $\mathsf{ValidateSplitsGini}(H_k, \mathbf{f}_k, \mathbf{t}_k, \mathsf{edges}) = 1$
return 1 (accept)

Procedure 12: InitHistsLabel

Input: Labels $\mathbf{y} = (y_i)_{i=1}^n$, binID, leaf indices $\mathbf{l} = (l_i)_{i=1}^n$
Output: Histograms H

for $f \in [d]$, $\ell \in [2N-1]$, $b \in [B]$, initialize $H[f][\ell][b] \leftarrow 0$.
for $i = 1$ **to** n **do**
  for $f = 1$ **to** d **do**
    $b \leftarrow \mathsf{binID}[i][f]$; $\ell \leftarrow l_i + (N-1)$
    $H[f][\ell][b][y_i] \leftarrow H[f][\ell][b][y_i] + 1$
for $\ell = N-1$ **down to** 1 **do** // propagate up to root
  for $f = 1$ **to** d **do**
    for $b = 1$ **to** B **do**
      $H[f][\ell][b][0] \leftarrow H[f][2\ell][b][0] + H[f][2\ell+1][b][0]$
      $H[f][\ell][b][1] \leftarrow H[f][2\ell][b][1] + H[f][2\ell+1][b][1]$
return H

Procedure 13: ValidateLeafWeightsLabel

Input: Leaf histograms H, weights $\mathbf{w} = (w_l)_{l=1}^N$
Output: Accept/Reject

for $l = 1$ **to** N **do**
  $\ell \leftarrow l + (N-1)$
  $H_0^{\text{leaf}} \leftarrow \sum_{b=1}^B H[1][\ell][b][0]$ // use any feature f, as the sum is feature-independent
  $H_1^{\text{leaf}} \leftarrow \sum_{b=1}^B H[1][\ell][b][1]$ // use any feature f, as the sum is feature-independent
  $w_l' \leftarrow \dfrac{H_1^{\text{leaf}}}{H_0^{\text{leaf}} + H_1^{\text{leaf}}}$
  assert $w_l = w_l'$

Procedure 14: ValidateSplitsGini

Input: Node histograms H, resulting splits $(\mathbf{f}, \mathbf{t}) = (f_\ell, t_\ell)_{\ell=1}^{N-1}$, and edges
Output: Accept/Reject

for $\ell = 1$ **to** $N-1$ **do**
  for $f = 1$ **to** d **do**
    $H_0 \leftarrow \sum_{b=1}^B H[f][\ell][b][0]$; $H_1 \leftarrow \sum_{b=1}^B H[f][\ell][b][1]$
    $H_{0,L} \leftarrow 0$, $H_{1,L} \leftarrow 0$
    for $b = 1$ **to** B **do**
      $H_{0,L} \leftarrow H_{0,L} + H[f][\ell][b][0]$; $H_{1,L} \leftarrow H_{1,L} + H[f][\ell][b][1]$
      $H_{0,R} \leftarrow H_0 - H_{0,L}$; $H_{1,R} \leftarrow H_1 - H_{1,L}$

Table 2: Credit Card Default ($n = 30001$, $d = 23$): Accuracy and runtime of our fixed-point algorithm vs XGBoost.

Depth	Trees	FixedAcc	XGBAcc	$\lvert\Delta\rvert$	T_{fixed} (s)	T_{xgb} (s)
4	50	0.8204	0.8208	0.0004	2.70	0.13
4	100	0.8200	0.8185	0.0016	5.37	0.19
5	50	0.8216	0.8191	0.0025	3.41	0.15
5	100	0.8191	0.8171	0.0020	6.83	0.22

$H_L \leftarrow H_{0,L} + H_{1,L}$; $H_R \leftarrow H_{0,R} + H_{1,R}$

$\mathsf{gain}[f][b] \leftarrow \left(1 - \left(\frac{H_0}{H_0+H_1}\right)^2 - \left(\frac{H_1}{H_0+H_1}\right)^2\right) - \frac{H_L}{H_L+H_R} \cdot \left(1 - \left(\frac{H_{0,L}}{H_L}\right)^2 - \left(\frac{H_{1,L}}{H_L}\right)^2\right) - \frac{H_R}{H_L+H_R} \cdot \left(1 - \left(\frac{H_{0,R}}{H_R}\right)^2 - \left(\frac{H_{1,R}}{H_R}\right)^2\right)$

Retrieve b_ℓ s.t. $t_\ell = \mathsf{edges}[f_\ell][b_\ell]$

(Argmax check) **assert** $\mathsf{gain}[f_\ell][b_\ell] \geq \mathsf{gain}[f][b]$ for all $f \in [d], b \in [B]$.

G Additional Experiments

G.1 Fixed-Point XGBoost Experiments

This section provides the plaintext fixed-point XGBoost training versus the standard-precision floating point training. Details are in Tables 2,5 3, 4,6.

Table 3: Breast Cancer ($n = 569$, $d = 30$): Accuracy and runtime of our fixed-point algorithm vs XGBoost.

Depth	Trees	FixedAcc	XGBAcc	$\lvert\Delta\rvert$	T_{fixed} (s)	T_{xgb} (s)
4	50	0.9737	0.9766	0.0029	0.48	0.02
4	100	0.9708	0.9766	0.0058	0.66	0.02
5	50	0.9708	0.9766	0.0058	0.56	0.02
5	100	0.9708	0.9795	0.0088	0.74	0.02

Table 4: Covertype - 50k ($n = 50000$, $d = 54$): Accuracy and runtime of our fixed-point algorithm vs XGBoost.

Depth	Trees	FixedAcc	XGBAcc	$\lvert\Delta\rvert$	T_{fixed} (s)	T_{xgb} (s)
4	50	0.8186	0.8217	0.0021	7.26	0.12
4	100	0.8213	0.8235	0.0022	14.56	0.35
5	50	0.8203	0.8220	0.0017	9.32	0.35
5	100	0.8635	0.8610	0.0025	18.47	0.52

G.2 Discussion for Fixed-Point XGBoost Experimental Results

Performance. As expected, our fixed-point implementation is slower than XGBoost, which is a highly optimized C++ library leveraging vectorization, cache tuning, and parallelism. However, once we level the playing field a bit by pinning both systems to a single thread, the slowdown becomes very manageable. On the Credit Default dataset, slowdowns range between 20× and 30×, while on the larger Covertype-100k dataset they range from 29× to 42×. With parallelism neutralized,

Table 5: Covertype - 100k ($n = 100000$, $d = 54$): Accuracy and runtime of our fixed-point algorithm vs XGBoost.

Depth	Trees	FixedAcc	XGBAcc	$\lvert\Delta\rvert$	T_{fixed} (s)	T_{xgb} (s)
4	50	0.8216	0.8216	0.0000	13.64	0.47
4	100	0.8415	0.8410	0.0004	27.05	0.73
5	50	0.8382	0.8414	0.0032	16.52	0.50
5	100	0.8653	0.8633	0.0020	33.11	0.77

Table 6: Adult ($n = 45222$, $d = 104$): Accuracy and runtime of our fixed-point algorithm vs XGBoost.

Depth	Trees	FixedAcc	XGBAcc	$\lvert\Delta\rvert$	T_{fixed} (s)	T_{xgb} (s)
4	50	0.8633	0.8711	0.0078	11.48	0.31
4	100	0.8645	0.8719	0.0074	22.75	0.45
5	50	0.8617	0.8708	0.0091	15.09	0.32
5	100	0.8615	0.8711	0.0096	30.20	0.49

the residual slowdown is consistent with constant-factor overheads of a Python/NumPy prototype, which are plausibly reducible in a lower-level implementation. A rewrite in a low level language with all the relevant optimizations should substantially reduce constant factors and, with the already-demonstrated parity in accuracy, plausibly bring slowdowns into the low double digits or better. In any case, although the inefficiency is important to note, the absolute running time which is around 30 seconds even on the larger datasets is still very practical.

Small scale (Breast Cancer) as a counterpoint. On the very small Breast Cancer dataset (Table 3), the slowdown is more pronounced in relative terms, averaging around 32×. This reflects the fact that on tiny workloads, the fixed costs of our Python-level and fixed-point implementation dominate, while optimized libraries like XGBoost maintain nearly constant runtime overhead.

G.3 Multiclass Classification

Extension to multiclass classification. Our implementation and evaluation focus on binary classification, which already captures many prediction tasks such as fraud detection, credit-risk prediction, and mortality prediction. Nevertheless, the ZKBoost framework is not intrinsically limited to binary classification. The standard multiclass extension of gradient boosting maintains, for each sample x_i, a vector of class logits

$$z_i = (z_{i,1}, \ldots, z_{i,C})$$

rather than a single scalar logit, where C is the number of classes. At each boosting round, the learner trains one tree per class, and the prediction probabilities are obtained by applying the softmax function

$$p_{i,c} = \frac{\exp(z_{i,c})}{\sum_{c'=1}^{C} \exp(z_{i,c'})}.$$

The gradients and Hessians used for split selection and leaf-weight computation then become class-indexed quantities, for example

$$g_{i,c} = p_{i,c} - \mathbf{1}[y_i = c]$$

and

$$h_{i,c} = p_{i,c}(1 - p_{i,c})$$

, under the usual diagonal approximation used in multiclass tree boosting.

The certification approach of ZKBoost can extend to this setting. Instead of validating a single tree T_k per boosting round, the certificate validates a collection $\{T_{k,c}\}_{c=1}^{C}$ of class-specific trees. The intermediate-score check is lifted from scalar logits to class-indexed logit vectors, while the tree-local validation procedures for inference, histogram reconstruction, split optimality, pruning, and leaf weights are applied independently to each class-specific tree. Thus, the main structural contribution

Table 7: Covertype-50k ($n = 50000$, $d = 54$): Accuracy of our fixed-point one-vs-rest multiclass implementation vs XGBoost.

Depth	**Trees**	**Fixed-OvR**	**XGB**
4	50	0.7845	0.7721
5	100	0.8338	0.8254
6	200	0.8759	0.8678
8	500	0.8891	0.8956

Table 8: Left: Proof of comparison compared to the previous [HCL+24]. Right: Proof of histogram construction based on ZK-RAM and weighted Logup. Numbers in parentheses are the improvements.

	LAN	WAN		LAN	WAN
[HCL+24]	17.59	19.17	w/ ZK-RAM [YH24]	20.48	31.77
Our Comparison	3.29 (5.3×)	8.52 (2.2×)	w/ Weighted LogUp [Hab22]	4.98 (4×)	10.35 (3×)

of CertXGB, which is separating inter-tree score consistency from tree-local bottom-up validation, applies unchanged. The proof cost scales linearly with the number of classes C, up to the additional cost of proving the softmax normalization.

A full multiclass implementation requires substantial engineering changes and is left for future work. However, preliminary measurements suggest that the additional softmax cost is not the dominant bottleneck. Using lookup-table techniques for exponentiation [HCL+24] together with our division proof, a ZK proof of softmax evaluation costs approximately $50\,\mu s$ per data point per boosting round in our prototype, adding about 5% overhead on the Covertype benchmark. We also implemented a fixed-point one-vs-rest multiclass variant as a sanity check for the numerical behavior of fixed-point tree boosting, and observed accuracy comparable to floating-point XGBoost (see Table 7). These results indicate that the main remaining challenge is more of an implementation effort rather than a conceptual limitation of ZKBoost.

G.4 ZKP Benchmarks

The performance proving the comparison and histogram construction are shown in Table 8. The improvement for proof of comparison is due to the reduction in the prevention of wrap-around attack and the switching to the membership proof based on LogUp [Hab22]. Additionally, it benchmarks both the ZK-RAM construction suggested by [PP24] and our improvement with LogUp [Hab22]. We use the VOLE-ZK-RAM for the former [YH24]. It shows that our histogram proof improves the naive implementation by $3 - 4\times$.