Securely Training Decision Trees Efficiently

Divyanshu Bhardwaj
Microsoft Research
Bengaluru, India
dbhardwaj@ucsb.edu

Sandhya Saravanan
Microsoft Research
Bengaluru, India
t-ssaravanan@microsoft.com

Nishanth Chandran
Microsoft Research
Bengaluru, India
nichandr@microsoft.com

Divya Gupta
Microsoft Research
Bengaluru, India
divya.gupta@microsoft.com

Abstract

Decision trees are an important class of supervised learning algorithms. When multiple entities contribute data to train a decision tree (e.g. for fraud detection in the financial sector), data privacy concerns necessitate the use of a privacy-enhancing technology such as secure multi-party computation (MPC) in order to secure the underlying training data. Prior state-of-the-art (Hamada *et al.* [18]) construct an MPC protocol for decision tree training with a communication of $O(hmN \log N)$, when building a decision tree of height h for a training dataset of N samples, each having m attributes.

In this work, we significantly reduce the communication complexity of secure decision tree training. We construct a protocol with communication complexity $O(mN \log N + hmN + hN \log N)$, thereby achieving an improvement of $\approx \min(h, m, \log N)$ over [18]. At the core of our technique is an improved protocol to regroup sorted private elements further into additional groups (according to a flag vector) while maintaining their relative ordering. We implement our protocol in the MP-SPDZ framework [1, 22] and show that it requires 10× lesser communication and is 9× faster than [18].

CCS Concepts

• **Security and privacy → Cryptography**; **Privacy-preserving protocols**.

Keywords

secure multiparty computation; decision tree training; privacy-preserving machine learning

ACM Reference Format:
Divyanshu Bhardwaj, Sandhya Saravanan, Nishanth Chandran, and Divya Gupta. 2024. Securely Training Decision Trees Efficiently. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3658644.3670268

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670268

1 Introduction

One of the most important and popular machine learning algorithms is decision tree learning. In this supervised learning approach, a classification or regression tree is built based on a set of features or attributes present in the training dataset. As with many learning algorithms, the accuracy of decision trees can be greatly improved with larger volumes of data. However, this can be a challenge since data may be present with many different parties who are unwilling to share their data in the clear with each other. Consider, for example, a decision tree model being built to analyze the risk profile associated with banking transactions. While the model built could benefit significantly from data from multiple banks, this is realistically difficult given the privacy concerns.

The natural question is: *can multiple parties, holding subsets of a training dataset, jointly train a decision tree model?* The cryptographic technique of secure multi-party computation (MPC) [9, 16, 39] provides a generic way for multiple mutually distrusting parties with private data to compute any function securely and hence provides a solution to the above problem. However, generic MPC protocols typically require copious amounts of communication between the participating parties (which grows with the complexity of the function being computed). Over the last few decades, this drawback of generic MPC has motivated development of specialized protocols for important functions of interest, one of them being secure decision tree training.

Perhaps the first work to consider secure decision tree training was that of Lindell and Pinkas [26]. The work of Hoogh *et al.* [14] provided the first concretely efficient MPC protocol for decision tree training for the specific case when all attributes are categorical (or discrete) values. For the more general (and useful) case of continuous attributes, the work of Abspoel [3] showed how to build a concretely efficient MPC protocol. This protocol required a total communication[1] of $O(2^h mN \log N)$ to build a decision tree of height h from N data points, each having m attributes, for the case of 3-party secure computation (tolerating one semi-honest corruption). The current state-of-the-art is the work of Hamada *et al.* [18] that introduced a new secure data structure and used that to obtain a protocol (for the same setting) for decision tree training with communication of $O(hmN \log N)$.

[1] All asymptotic communication overheads presented in the paper include an implicit factor of ℓ, the bitlength of elements. We omit this everywhere for the sake of readability.

1.1 Our Results

In this work, we make the following contributions:

- We provide a new and more efficient protocol for secure decision tree training. For the case of 3-party secure computation tolerating one semi-honest corruption, our techniques result in a protocol with communication complexity $O(mN \log N + hmN + hN \log N)$ that improves upon the state-of-the-art [18] by roughly $\min(h, m, \log N)$. Our techniques easily extend to the case of more parties and other threat models (Section 7).
- At the heart of our new protocol is a key sub-protocol that enables us to securely divide multiple groups of sorted secret elements into further sub-groups, according to a private flag vector, while retaining their relative ordering and hiding the group boundaries with a complexity of $O(N)$. Such a protocol might be of independent interest.
- Finally, we implement [2] our protocol and show that our protocol is concretely 10× more communication frugal and 9× faster than the state-of-the-art [18].

1.2 Our Techniques

We retain the high-level protocol structure from Hamada *et al.* [18] (that avoids the exponential dependence on the height of the tree). However, we make crucial changes that reduce the use of expensive sub-protocols, e.g., sorting. At the technical level, we design new (inexpensive) sub-protocols that enable the re-use of the results of expensive sub-protocols, and hence, amortize the cost of expensive sorting. Before we describe our ideas in more detail, let us revisit the protocol of Hamada *et al.* [18] starting with a basic description of (cleartext) decision tree training.

A decision tree with height h has h levels, and is constructed level by level starting from the root to the leaves. Each data point (among the total N data points) belongs to a unique node at each level, based on the sequence of decisions made from the root to this level. To create the next level, for each node, we need to figure out the splitting attribute and the corresponding threshold, and using this, data points in a node get sub-divided into its two children. One popular methodology to pick a good split is "Gini index" (defined in Section 2) where one determines the attribute and threshold with the minimum Gini index.

The secure training protocol in [18] uses a single data structure to store all data points at a level, where data points for a node belong to a group and the group boundaries are kept hidden. Next, they design efficient secure protocols to process over this data structure. To run the decision tree training algorithm securely, they create protocols to securely compute the best split for each node/group and also securely sub-group each group to create the next level. This is done through the following 2 high level steps.

- First, the Gini index for all possible attributes and thresholds is computed. To do this, for each attribute, a groupwise sort is performed using which the Gini index for all thresholds can be computed using communication of $O(N)$. Here, groupwise sort requires communication complexity of $O(N \log N)$ and this itself results in overall complexity of $O(hmN \log N)$ as this sort needs to be performed per attribute and per level.
- Second, once the Gini index is computed per attribute and threshold, the minimum is computed in two steps - (1) for each attribute, compute the threshold with minimum Gini index in each group using a GroupMax protocol (with cost $O(N \log N)$) (2) find the best attribute by computing the minimum over all attributes with a Max protocol (with cost $O(mN)$). So the overall complexity of finding the best Gini index is also $O(hmN \log N)$ as the first step is repeated per attribute and per level.

To reduce the overall complexity, we must reduce the use of sorting and compute the best split given the Gini indices more efficiently.

Our protocol makes two fundamental changes to address the above two performance bottlenecks. First, for each attribute, we sort the data points *only once*, at the beginning. This creates a problem as computing all the Gini indices efficiently requires data points to be sorted *within each group*. To address this, we reduce this functionality to computing a new sorting permutation given the old sorting permutation along with the grouping information, and provide an efficient protocol to realize the same with communication overhead of $O(N)$. Thus, we only must perform the secure sorting once (irrespective of the tree height). This results in an overall communication overhead of $O(mN \log N + hmN)$ as opposed to $O(hmN \log N)$ to compute sorted groups throughout the training.

Second, in order to compute the minimum Gini index, we reverse the two operations performed in [18]. That is, (for each level) we first compute for each sample, the minimum value over all attributes using Max protocol (with cost $O(mN)$), and then apply the GroupMax protocol *once* over the resulting N-element vector to compute the minimum Gini index over all groups (with cost $O(N \log N)$). We show that doing so preserves functionality since the group boundaries remain the same i.e. sorting according to different attributes is within groups and does not change the starting and ending of a group. Crucially, this results in a reduced overall communication overhead of $O(hmN + hN \log N)$ (instead of $O(hmN \log N)$).

1.3 Related Work

Concurrent and independent to our work is ENTS [17] that also improves upon Hamada *et al.* [18]. While they too reduce the number of sorting operations, their asymptotic complexity is the same as [18]. Additionally, they enable almost all the cleartext values to be secret shared over smaller rings and convert them to large rings only when necessary, thus saving on communication. This technique is orthogonal and compatible with our protocol design and the two techniques can be combined. We provide a comparison of ENTS [17] with our unmodified protocol in Section 6.

Protocols for secure training over horizontally (or vertically) partitio- ned dataset are proposed in [29, 31, 38] ([27, 35, 36]). These works assume that the computing servers know the partition of the training dataset in the clear. Moreover, the main efficiency improvements of these protocols come from leaking intermediate outputs in the training which can allow an adversary to learn non trivial information about the data (e.g. an adversary can learn the training data distribution from the leakage in [35]).

In contrast, training algorithms that guarantee the complete privacy of the dataset have been designed either using Fully Homom-

[2] Link to the code.

orphic Encryption [15] or MPC based techniques (such as our work and the works discussed in the previous subsection). In the FHE based decision tree protocols [4, 5], a data owner sends the encrypted dataset to a server which runs the training algorithm. While these algorithms have low communication and round complexity, they have two major limitations: a) the training algorithm is modified to approximate various functions such as comparisons; and b) the computational overhead is exorbitantly high making it impractical (e.g. the runtime in [4] for training even a small decision tree of height 4 with only 10, 000 samples is ≈ 1.5 days).

A long line of work has focused on secure decision tree evaluation only [11, 13, 21, 28, 33, 34, 37]. Our trained decision tree is compatible with these works and can be augmented to facilitate both secure training and evaluation of decision trees.

1.4 Organization

Section 2 introduces notation, the security model and provides a description of the cleartext algorithm for decision tree training. It also introduces secret sharing schemes and specifically the replicated secret sharing scheme used in this work. In Section 3, we describe the various crypto building blocks needed to construct our secure decision tree protocol. This includes element wise operations (such as multiplication, comparison etc.) as well as vector and groupwise operations (e.g. shuffle, GroupMax etc.) for which protocols were presented in prior works. Section 4 provides the complete description of our secure decision tree training protocol, while Section 5 describes the security proof of our new protocols. In Section 6, we present details of our implementation and all experimental results. In Section 7, we discuss extensions of our protocol to other threat models and complex use cases such as multi-class classification, as well as its applicability in training scenarios such as random forests and gradient boosted decision trees.

2 Preliminaries

2.1 Notation

Let $\mathbb{N}$ be the set of natural numbers. $[N]$ denotes the set $\{1, \ldots, N\}$ for $N \in \mathbb{N}$. $\mathbb{Z}_{2^\ell}$ is a ring of integers modulo 2^ℓ for $\ell \in \mathbb{N}$. Subset B of set A is denoted by $B \subseteq A$. We use n to denote $\log_2 N$.

2.2 Security Model

Our protocols satisfy the simulation based security definition [24]. We consider 3 parties S_0, S_1, S_2 and a semi honest adversary $\mathcal{A}$ corrupting one of the parties. All recent works on privacy preserving decision tree training [3, 18] also consider this setting. All parties follow the protocol specification faithfully (semi-honest behaviour) and we show that the view of no single party reveals any information. Suppose Π is the 3-party protocol that computes the functionality $\mathcal{F}$. $\mathcal{A}$ can corrupt at most one party out of S_0, S_1 and S_2. For security, we require that the view of adversary $\mathcal{A}$ and the outputs of honest parties in the real world are computationally indistinguishable from the view and output of a simulator in an ideal world which has access to the ideal functionality $\mathcal{F}$. $\mathcal{F}$ simply receives inputs from all three parties and provides the function output (as defined by the functionality) to all parties i.e. for any semi honest non-uniform PPT adversary $\mathcal{A}$ that corrupts at most one party, there exists an ideal world PPT simulation Sim such that for any input inp to the protocol Π that computes $\mathcal{F}$, we have $\mathsf{Real}(\Pi, \mathcal{A}, \mathsf{inp}) \approx_c \mathsf{Ideal}(\mathcal{F}, \mathsf{Sim}, \mathsf{inp})$ where Real is the joint distribution of the view of adversary $\mathcal{A}$ and output of honest parties in the real world, and Ideal is the joint distribution of the view of the simulator Sim and output of honest parties in the ideal world.

2.3 Cleartext Decision Tree Algorithm

We begin with a description of the cleartext decision tree algorithm we use in this work (which is the same as the one in [3, 18]). We have a labelled dataset $\mathbf{D}$ with m attributes $\mathbf{x} = (x_1, \ldots, x_m)$ and label y. Here, $\mathbf{D}[i] = (x_1[i], \ldots, x_m[i], y[i])$, where $x_j[i]$ is the value of the j^{th} attribute and $y[i]$ is the label value for the i^{th} sample in the dataset. A decision tree is a machine learning model that predicts the value of output variable y given the input attributes $\mathbf{x}$.

A binary decision tree has two kinds of nodes (1) internal nodes associated with a test of the form $x_j < t$ (2) leaf nodes associated with a label b. Each edge out of an internal node denotes a possible outcome of the test i.e. true or false, and a corresponding child node (true child or false child). Typically, the left child is the false child and right child is the true child.

Decision Tree Evaluation. Given input attributes $\mathbf{x}$ of a sample, we can predict its label as follows. Starting from the root node, the test at each internal node is evaluated for the sample and based on the outcome of test, we trace an outgoing edge to reach the child node i.e. if the test $x_j < t$ holds, we traverse to the right child, else, we traverse to the left child. Traversing in this fashion, we reach a leaf node and output the leaf label as the predicted value of the label of the sample. The attributes can be continuous, where attribute values are real numbers, or discrete, where the attribute values are from a finite set.

Decision tree training. The algorithm to securely train decision trees with discrete attributes is straightforward and given in [14]. We focus on the more complex case of continuous attributes as our algorithm can be easily extended to handle discrete attributes as well. A more detailed discussion on how to handle discrete attributes is given in Appendix B.

Algorithm 14 (Appendix A) describes how to train a decision tree of height h with continuous attributes. The decision tree is trained from root node to leaf nodes in a recursive manner. The root node is considered to be at height 0. To distinguish leaf nodes from internal nodes, the algorithm checks the height of the current node. If the height is less than h, the current node is an internal node. In that case, the algorithm selects a test $x_j < t$ to be performed at that node that splits the dataset in a way that minimizes the Gini index of the resulting split. The Gini index G for splitting a dataset $\mathbf{D}$ using test $x_j < t$ is defined as:

$$G_{x_j<t}(\mathbf{D}) = \frac{|\mathbf{D}_{x_j<t}|}{|\mathbf{D}|}\,\mathrm{Gini}(\mathbf{D}_{x_j<t}) + \frac{|\mathbf{D}_{x_j\geq t}|}{|\mathbf{D}|}\,\mathrm{Gini}(\mathbf{D}_{x_j\geq t})$$

where $\mathrm{Gini}(\mathbf{D}) = 1 - \sum_{b\in\{0,1\}}(|\mathbf{D}_{y=b}|^2/|\mathbf{D}|^2)$. The best split has the minimum Gini index and Abspoel *et al.* [3] showed that minimizing the Gini index is same as maximizing the following expression:

$$\mathrm{G}'_{x_j<t}(\mathbf{D}) = \Big(\left|\mathbf{D}_{x_j\geq t}\right| \left(\left|\mathbf{D}_{x_j<t\wedge y=0}\right|^2 + \left|\mathbf{D}_{x_j<t\wedge y=1}\right|^2\right) \quad (1)$$
$$+ \left|\mathbf{D}_{x_j<t}\right| \left(\left|\mathbf{D}_{x_j\geq t\wedge y=0}\right|^2 + \left|\mathbf{D}_{x_j\geq t\wedge y=1}\right|^2\right)\Big) / \left(\left|\mathbf{D}_{x_j<t}\right| \left|\mathbf{D}_{x_j\geq t}\right|\right),$$

where $\mathbf{D}_{x_j<t\wedge y=b} = \{(\mathbf{x}, y) \in \mathbf{D} \mid x_j < t \wedge y = b\}$. The algorithm selects the test $x_j < t$ that maximizes this expression.

Once the test is selected, the algorithm partitions the training dataset into $\mathbf{D}_{x_j<t}$ and $\mathbf{D}_{x_j\geq t}$. The left subtree $\mathcal{T}_l$ is trained on the partition $\mathbf{D}_{x_j<t}$ and right subtree $\mathcal{T}_r$ is trained on the partition $\mathbf{D}_{x_j\geq t}$. The algorithm outputs a decision tree with the current node as root node, $\mathcal{T}_l$ as left child and $\mathcal{T}_r$ as right child.

If the height of the current node is h, it is a leaf node. Let $\mathbf{D}_L$ be the partition of dataset $\mathbf{D}$ corresponding to the L^{th} leaf node. The label of the leaf node is then set to the most occurring label in the partition $\mathbf{D}_L$. In case of binary classification, we ~~can~~ compute the label of L^{th} leaf node as follows:

$$\text{Label} = \sum_{i\in\mathbf{D}_L} y[i] \overset{?}{>} \sum_{i\in\mathbf{D}_L} (1 - y[i])$$

Number encodings. Real numbers are represented using fixed or floating point numbers. However, we do not perform any non-linear computation on attribute values except for comparisons in decision tree training. This means that we can map these attribute values to elements of a sufficiently large ring as long as the ordering is preserved. We use $\mathbb{Z}_{2^\ell}$ ring (setting $\ell = 64$ in the experiments) to represent the continuous attribute values and class labels. In this work, we focus on the case of binary labels. However, the training algorithm can be extended to multi-class classification as well, as described in Section 7.

2.4 Secret Sharing Schemes

A secret sharing scheme [32] allows a dealer to distribute a secret value amongst a set of parties such that only allowed subsets of parties can reconstruct the secret value and any unauthorized subset of parties does not learn anything about the secret. We use $(3, 2)$-Replicated secret sharing (RSS) [20] that allows distributing a secret between 3 parties such that it is completely hidden from any single party and any two parties can reconstruct the secret. A $(3, 2)$-RSS over ring $\mathbb{Z}_{2^\ell}$ has the following algorithms:

- Share: On input $v \in \mathbb{Z}_{2^\ell}$, sample $v_0, v_1, v_2 \in \mathbb{Z}_{2^\ell}$ such that $v_0 + v_1 + v_2 = v \bmod 2^\ell$ and output the shares (x_0, x_1, x_2) where
$$x_0 = (v_0, v_1), x_1 = (v_1, v_2), x_2 = (v_2, v_0)$$
- Reconstruct: On input $x_i = (v_0, v_1), x_j = (v_1, v_2) \in \mathbb{Z}^2_{2^\ell}$ where $i \neq j; i, j \in \{1, 2, 3\}$, output $v' = (v_0 + v_1 + v_2) \bmod 2^\ell$.

We denote the replicated shares of $v \in \mathbb{Z}_{2^\ell}$ by $\langle v\rangle$ and the boolean replicated shares $v \in \mathbb{Z}_2$ by $\langle v\rangle^{\mathbf{B}}$. The share of party S_i is (v_i, v_{i+1}).

3 Crypto Building Blocks

Here, we describe the sub-protocols that serve as building blocks for our secure decision tree training protocol. We first describe 3-party functionalities where parties begin with shares of inputs according to the (3,2)-RSS scheme described in Section 2.4 and receive shares of outputs according to the same secret sharing scheme. We also describe the (existing) protocols realizing these functionalities and their efficiency.

3.1 Element wise operations

We require the following primitives to build our protocols:

- **Secure Multiplication:** denoted by $\langle z\rangle = \Pi_{\mathsf{MULT}}(\langle u\rangle, \langle v\rangle)$.
- **Secure Bit Decomposition:** denoted by $\langle u\rangle^{\mathbf{B}} = \Pi_{\mathsf{A2B}}(\langle u\rangle)$.
- **Secure Comparison:** denoted by $\langle b\rangle = \Pi_{\mathsf{LT}}(\langle u\rangle, \langle v\rangle)$.
- **Secure Equality:** denoted by $\langle b\rangle = \Pi_{\mathsf{EQ}}(\langle u\rangle, \langle v\rangle)$.

We discuss the protocols for these and their communication costs in Appendix D.

3.2 Permuting Secret Vectors

We represent a permutation over N elements as an N-length vector $\mathbf{P}$ where $\mathbf{P}[i]$ denotes the position of the i^{th} element in the permuted order. Let $\mathbf{A} \in \mathbb{Z}^N_{2^\ell}$ be a vector and $\mathbf{P}$ be a permutation. Let $\mathbf{A}^{\mathbf{P}}$ be the rearrangement of $\mathbf{A}$ according to $\mathbf{P}$ i.e. $\mathbf{A}^{\mathbf{P}} = \mathrm{Permute}(\mathbf{A}, \mathbf{P})$. Since $\mathbf{P}[i]$ denotes the new position of the i^{th} element, we have

$$\mathbf{A}^{\mathbf{P}}\left[\mathbf{P}[i]\right] = \mathbf{A}[i] \quad \text{for all } i \in [N]$$

The secret share of a permutation vector is defined as

$$\langle\mathbf{P}\rangle = (\langle\mathbf{P}[1]\rangle, \ldots, \langle\mathbf{P}[N]\rangle)$$

We make use of four functionalities that permute secret vectors – a) randomly shuffle a secret vector; b) apply a secret permutation to a secret vector; c) compose secret permutations; and d) stably sort a secret vector – and we describe them below. The protocols, with linear communication cost realizing these functionalities were given in [7, 8] (see Appendix E). Table 1 gives the concrete cost.

Oblivious Shuffling. (Functionality 1) Party $S_i (i \in \{0, 1, 2\})$ inputs replicated secret shares of vector $\langle\mathbf{X}\rangle_i \in \mathbb{Z}^N_{2^\ell}$ and a pair of permutations $(\pi_i, \pi_{(i+1) \bmod 3})$. The functionality permutes $\mathbf{X}$ according to $\pi = \pi_2 \circ \pi_1 \circ \pi_0$ and outputs replicated secret shares of the permuted vector $\mathbf{Y}$ to parties. Protocol Π_{Shuffle} in [7] (Figure 5) computes the functionality with $4N\ell$ bits of communication in 2 rounds.

The Shuffle functionality can be used to shuffle a vector by permutation π^{-1} by applying the permutations in reverse order i.e. apply π_2^{-1}, then π_1^{-1} and finally π_0^{-1} as $\pi^{-1} = \pi_0^{-1} \circ \pi_1^{-1} \circ \pi_2^{-1}$. A typical use of shuffling is to hide data dependent memory accesses. To do this, parties shuffle a vector using π, perform the memory accesses and then shuffle by π^{-1} to get back the same ordering.

Functionality 1: Shuffle

Input: Party S_i, $i \in \{0, 1, 2\}$ inputs $\langle\mathbf{X}\rangle_i$ and (π_i, π_{i+1})
Output: Parties receive $\langle\mathbf{Y}\rangle$ where $\mathbf{Y}$ is a rearrangement of $\mathbf{X}$ according to $\pi = \pi_2 \circ \pi_1 \circ \pi_0$.

1 Reconstruct $\mathbf{X}$ and $\pi = \pi_2 \circ \pi_1 \circ \pi_0$.
2 Compute $\mathbf{Y} = \mathrm{Permute}(\mathbf{X}, \pi)$.
3 Output $\langle\mathbf{Y}\rangle$.

Apply permutation. Functionality 2 takes shares of a vector $\mathbf{X}$ and permutation $\mathbf{P}$ and outputs shares of $\mathbf{X}^{\mathbf{P}}$ where $\mathbf{X}^{\mathbf{P}} = \mathrm{Permute}(\mathbf{X}, \mathbf{P})$.

Protocol $\Pi_{\mathsf{ApplyPerm}}$ (Algorithm 16) in [8] realizes ApplyPerm with $8N\ell$ bits of communication in 2 rounds.

```
Functionality 2: ApplyPerm
  Input: Parties input $\langle \mathbf{X} \rangle$ and $\langle \mathbf{P} \rangle$
  Output: Parties receive $\left\langle \mathbf{X}^{\mathbf{P}} \right\rangle$ where $\mathbf{X}^{\mathbf{P}} = \text{Permute}(\mathbf{X}, \mathbf{P})$.
  Reconstruct $\mathbf{X}$ and $\mathbf{P}$.
  Compute $\mathbf{X}^{\mathbf{P}}$ where $\mathbf{X}^{\mathbf{P}} = \text{Permute}(\mathbf{X}, \mathbf{P})$.
  Output $\left\langle \mathbf{X}^{\mathbf{P}} \right\rangle$.
```

Composing permutations. Functionality 3 takes shares of permutations $\mathbf{P}, \mathbf{Q}$ as input and outputs shares of permutation $\mathbf{R} = \mathbf{P} \circ \mathbf{Q}$ where $\mathbf{P} \circ \mathbf{Q}$ denotes applying $\mathbf{Q}$ followed by applying $\mathbf{P}$ so $\mathbf{R}[i] = \mathbf{P}[\mathbf{Q}[i]]$ i.e. i^{th} element goes to $\mathbf{Q}[i]$ position which then goes to $\mathbf{P}[\mathbf{Q}[i]]$ position. Protocol Π_{PermComp} (Algorithm 17) in [8] computes PermComp with $8N\ell$ bits communication in 4 rounds.

Composing permutations is particularly helpful when we have multiple permutations (say k permutations) that we have to apply to multiple vectors (say t vectors of length N) sequentially. We would have to apply k permutations to each vector which will cost $kt \times 8N\ell$ bits using ApplyPerm. Instead we use PermComp to compute the final permutation to be applied to vectors and apply this permutation to all l vectors which costs $(k + t) \times 8N\ell$ bits.

```
Functionality 3: PermComp
  Input: Parties input $\langle \mathbf{P} \rangle$ and $\langle \mathbf{Q} \rangle$
  Output: Parties receive $\langle \mathbf{R} \rangle$ where $\mathbf{R} = \mathbf{P} \circ \mathbf{Q}$.
  Reconstruct $\mathbf{P}, \mathbf{Q}$.
  Compute $\mathbf{R}$ where $\mathbf{R}[i] = \mathbf{P}[\mathbf{Q}[i]]\ \forall i \in [N]$.
  Output $\langle \mathbf{R} \rangle$.
```

Sorting permutation for binary key. Functionality 4 takes an N-length binary vector $\mathbf{b} \in \{0,1\}^N$ as input and outputs shares of permutation $\mathbf{P}$ that stably sorts $\mathbf{b}$.

Definition 3.1. Permutation $\mathbf{P}$ stably sorts a list $\mathbf{b}$ if $\forall i, j \in [N]$

$$\mathbf{P}[i] < \mathbf{P}[j] \implies (\mathbf{b}[i] < \mathbf{b}[j]) \text{ OR } (\mathbf{b}[i] = \mathbf{b}[j] \text{ AND } i < j)$$

i.e. $\mathbf{P}$ sorts the vector while preserving the order of entries with same value. Protocol $\Pi_{\mathsf{SortPermBit}}$ (Algorithm 18) in [8] computes SortPermBit with $3N\ell$ bits communication in 1 round.

```
Functionality 4: SortPermBit
  Input: Parties input $\langle \mathbf{b} \rangle$ where $\mathbf{b} \in \{0,1\}^N$.
  Output: Parties receive $\langle \mathbf{P} \rangle$ where $\mathbf{P}$ stably sorts $\mathbf{b}$.
  Reconstruct $\mathbf{b}$.
  Compute permutation $\mathbf{P}$ that stably sorts $\mathbf{b}$.
  Output $\langle \mathbf{P} \rangle$.
```

Sorting permutation for arbitrary key. Functionality 5 takes a vector $\mathbf{x} \in \mathbb{Z}_{2^\ell}^N$ as input and outputs shares of permutation $\mathbf{P}$ that stably sorts $\mathbf{x}$. Protocol Π_{SortPerm} (Algorithm 19) in [8] securely computes SortPerm with N calls to Π_{A2B}, l calls to $\Pi_{\mathsf{ApplyPerm}}$, l calls to $\Pi_{\mathsf{SortPermBit}}$ and l calls to Π_{PermComp}. Note that [18] considers Π_{SortPerm} realized through a comparison based sorting protocol, such as the one in [19]. The implementation of [18] (from [2]) uses Π_{SortPerm} based on an oblivious radix sort protocol as in [8]. The asymptotic improvement of our protocol over [18] remains the same, irrespective of the underlying sort protocol used. To be fair in our comparison of performance with [18], we use the oblivious radix sort protocol in the implementation.

```
Functionality 5: SortPerm
  Input: Parties input $\langle \mathbf{x} \rangle$ where $\mathbf{x} \in \mathbb{Z}_{2^\ell}^N$.
  Output: Parties receive $\langle \mathbf{P} \rangle$ where $\mathbf{P}$ stably sorts $\mathbf{x}$.
  Reconstruct $\mathbf{x}$.
  Compute permutation $\mathbf{P}$ that stably sorts $\mathbf{x}$.
  Output $\langle \mathbf{P} \rangle$.
```

3.3 Groupwise Operations

Groupwise operations, first described in [18], are important constructions used in our decision tree training protocol.

Privately grouped vector. Let $\mathbf{x}$ be a vector of length N, secret shared among parties and grouped internally. This means that the vector is divided into groups of different sizes. To represent these internal groups, parties also have a secret shared *group flag vector* $\mathbf{g}$ of length N such that $\mathbf{g}[i] = 1$ if $\mathbf{x}[i]$ is the first element of any group. For example, let $\mathbf{x} = [2, 5, 6, 8, 10, 13]$ and $\mathbf{g} = [1, 0, 0, 0, 1, 0]$. Then entries with index 1 to 4 belong to first group and entries with index 5 to 6 belong to second group as $\mathbf{g}[5] = 1$ (Assume 1-indexing). Note that $\mathbf{g}[1]$ is always 1 as it is the first entry of the first group. This data structure allows parties to secret share internally grouped vectors and compute groupwise operations efficiently.

The authors of [18] give efficient secure protocols to compute various groupwise operations like Group Sum, Group Prefix Sum and Group Max. We will require these group wise operations as building blocks. Let $i \in [N]$ be an index and l_i, r_i be the leftmost and rightmost index of the group that i belongs to. In the above example, $l_i = 1, r_i = 4$ for $1 \le i \le 4$ and $l_i = 5, r_i = 6$ for $5 \le i \le 6$. Then we can define the following group wise operations:

(1) $\langle \mathbf{a} \rangle = \Pi_{\mathsf{GroupSum}} (\langle \mathbf{x} \rangle, \langle \mathbf{g} \rangle)$
(2) $\langle \mathbf{b} \rangle = \Pi_{\mathsf{GroupPrefixSum}} (\langle \mathbf{x} \rangle, \langle \mathbf{g} \rangle)$
(3) $\langle \mathbf{c} \rangle = \Pi_{\mathsf{GroupMax}} (\langle \mathbf{x} \rangle, \langle \mathbf{g} \rangle)$
(4) $\langle \mathbf{d} \rangle = \Pi_{\mathsf{GroupFirstOne}} (\langle \mathbf{x} \rangle, \langle \mathbf{g} \rangle)$ defined when $\mathbf{x} \in \{0,1\}^N$

where $\forall i \in [N]$,

$$\mathbf{a}[i] = \sum_{j=l_i}^{r_i} \mathbf{x}[j]; \quad \mathbf{b}[i] = \sum_{j=l_i}^{i} \mathbf{x}[j]; \quad \mathbf{c}[i] = \text{Max}_{j \in [l_i, r_i]}\, \mathbf{x}[j]$$
$$\mathbf{d}[i] = 1 \text{ if } \mathbf{x}[i] = 1 \text{ and } \mathbf{x}[j] = 0\ \forall\ j \in [l_i, i),\ \textit{else}\ 0$$

By instantiating SortPermBit and ApplyPerm with protocols from Appendix E, we can compute groupwise operations with $O(N)$ communication in $O(1)$ rounds. In contrast, the communication complexity of these protocols in [18] was $O(N \log N)$ and

Protocol	Input	Cost	Rounds
ApplyPerm	$\langle \mathbf{x} \rangle, \langle \mathbf{P} \rangle \in \mathbb{Z}_{2^\ell}^N$	$8N\ell$	2
PermComp	$\langle \mathbf{P} \rangle, \langle \mathbf{Q} \rangle \in \mathbb{Z}_{2^\ell}^N$	$8N\ell$	4
SortPermBit	$\langle \mathbf{b} \rangle \in \mathbb{Z}_{2^\ell}^N$	$3N\ell$	1
SortPerm	$\langle \mathbf{x} \rangle \in \mathbb{Z}_{2^\ell}^N$	$42nN\ell$	n
GroupSum	$\langle \mathbf{x} \rangle, \langle \mathbf{b} \rangle \in \mathbb{Z}_{2^\ell}^N$	$25N\ell$	6
GroupPrefixSum	$\langle \mathbf{x} \rangle, \langle \mathbf{g} \rangle \in \mathbb{Z}_{2^\ell}^N$	$28N\ell$	7
GroupMax	$\langle \mathbf{x} \rangle, \langle \mathbf{g} \rangle \in \mathbb{Z}_{2^\ell}^N$	$24nN\ell$	$n \log \ell$
GroupFirstOne	$\langle \mathbf{b} \rangle, \langle \mathbf{g} \rangle \in \mathbb{Z}_{2^\ell}^N$	$51N\ell$	$2 \log \ell$

Table 1: Communication (in bits) and Rounds of Protocols

Layer$^{(1)}$	
NID	1
A	2
T	10

Layer$^{(2)}$		
NID	1	2
A	1	3
T	50	20

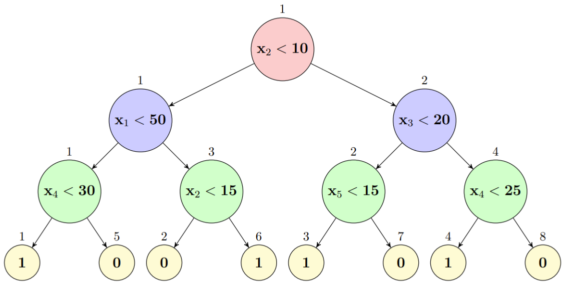
Layer$^{(3)}$				
NID	1	2	3	4
A	4	5	2	4
T	30	15	15	25

Internal Node Layers

Layer$^{(4)}$								
NID	1	2	3	4	5	6	7	8
L	1	0	1	1	0	1	0	0

Leaf Node Layer

(a) Example output of training algorithm for height 3.



(b) Decision Tree constructed from output.

Figure 1: (a) is a sample output of our training protocol for height 3 which takes as input 5 attribute values and outputs a binary label. There are 3 internal node layers and 1 leaf node layer. (b) is the decision tree encoded in the output. The layers are color coded. Each node has a node id (written on top of the node), splitting attribute (A $\in$ [1, 5]) and splitting threshold (T). In the output, Layer$^{(1)}$ has one node with value (1, 2, 10) which means there is one node in 1st layer with node id 1 and test $\mathbf{x}_2 < 10$. From the numbering of node ids, the left child (right child) of root node will have node id 1 (2) respectively in Layer$^{(2)}$. From the description of Layer$^{(2)}$ in output, the left child will have test $\mathbf{x}_1 < 50$ and right child will have test $\mathbf{x}_3 < 20$ as shown in (b). Continuing this way, we can map the output description (a) to the decision tree (b).

$O(\log N)$ rounds. Hence, we save a factor of $\log N$ in both communication and round complexity in groupwise operations. The concrete complexities are summarised in Table 1.

4 Secure Decision Tree Training

We begin with a description of the input and output formats of our secure training protocol.

Input. To begin with, the 3 parties have (3, 2) replicated secret shares of a labelled dataset $\mathbf{D} \in \left(\mathbb{Z}_{2^\ell}^m \times \{0,1\}\right)^N$ where m is the number of attributes and N is the number of samples in dataset $\mathbf{D}$. The height h of the decision tree to be built is public information. $\mathbf{D}[j] = (\mathbf{x}_1[j], \ldots, \mathbf{x}_m[j], \mathbf{y}[j])$ are the attribute and label values for the j^{th} sample. $\mathbf{x}_i \in \mathbb{Z}_{2^\ell}^N$ is a vector of all the N values of the i^{th} attribute and $\mathbf{y} \in \{0,1\}^N$ is the vector of N labels.

Output. We setup some notation. Layer$^{(k)}$, parameterized by $k \in \{1, \ldots, h+1\}$, is a collection of nodes at a distance of $k-1$ from root node. There are $n_k = 2^{k-1}$ nodes in Layer$^{(k)}$. Layer$^{(1)}$ consists of only the root node. Layer$^{(2)}$ consists of child nodes of the root node and so on. If a node belongs to Layer$^{(k)}$, its child nodes belong to Layer$^{(k+1)}$. The leaf nodes belong to Layer$^{(h+1)}$.

Each internal node has three variables associated with it: node id (NID), splitting attribute ($A \in [m]$) and splitting threshold ($T \in \text{domain}(A)$). Each leaf node has two variables: node id (NID) and label ($L \in \{0,1\}$). The NID is unique for each node within a layer. For a node with node id NID in Layer$^{(k)}$, the left child has node id NID and right child has node id $NID + 2^{k-1}$ in Layer$^{(k+1)}$. The node id NID is set to 0 for the root node in Layer$^{(1)}$.

Final output. The final output of the secure training protocol Π_{Train} (Protocol 15) is the trained binary decision tree of height h in the format described below. For all internal node layers $k \in \{1, \ldots, h\}$, parties output $\left\langle \mathsf{Layer}^{(k)} \right\rangle = (\langle \mathbf{NID}_k \rangle, \langle \mathbf{A}_k \rangle, \langle \mathbf{T}_k \rangle)$ where

(1) $\mathbf{NID}_k$ is a vector of length n_k that stores the node id of nodes in the k^{th} layer. If the tree constructed is an incomplete binary tree i.e. the actual number of nodes are less than n_k, then the remaining elements of $\mathbf{NID}_k$ are initialized to 0.

(2) $\mathbf{A}_k$ is a vector of length n_k that stores the attribute for splitting at nodes in k^{th} layer. If $\mathbf{NID}_k[i] = 0$, then $\mathbf{AID}_k[i] = 0$.

(3) $\mathbf{T}_k$ is a vector of length n_k that stores the threshold for splitting at nodes in k^{th} layer. If $\mathbf{NID}_k[i] = 0$, then $\mathbf{T}_k[i] = 0$.

For a given layer, $(\mathbf{NID}_k[i], \mathbf{A}_k[i], \mathbf{T}_k[i])$ are the node id, attribute and threshold of $i^{th}(1 \le i \le n_k)$ node in the layer.

For the leaf node layer i.e. $k = h+1$, parties output $\left\langle \mathsf{Layer}^{(h+1)} \right\rangle = (\langle \mathbf{NID}_{h+1} \rangle, \langle \mathbf{L}_{h+1} \rangle)$ where $\mathbf{L}_{h+1}$ is a vector of length n_{h+1} that stores labels corresponding to leaf nodes. An example output of the training protocol along with the corresponding decision tree is given in Figure 1 to illustrate how the output of the training protocol encodes a binary decision tree.

Intermediary outputs. The layers of the decision tree are trained sequentially. We call State$^{(k)}$, $k \in [h+1]$, an intermediary output of the training protocol. State$^{(k)}$, $k \in [h+1]$ is a tuple of $m+3$ vectors $\{\mathbf{D}_k, \mathbf{g}_k, \{\mathbf{P}_k^i\}_{i \in [m]}, \mathbf{NID}\}$, each of length N, where

(1) $\mathbf{D}_k$ is the training dataset grouped by nodes in layer k,

(2) $\mathbf{g}_k$ is the group flag vector encoding groups in $\mathbf{D}_k$,

Variable	Description
h	height of tree
m	number of attributes
N	number of training samples
ℓ	bitlength of attributes
$\mathbf{D}$	Labeled training dataset
$\mathbf{x}_i$	i^{th} attribute vector
$\mathbf{y}$	Label vector
$\mathbf{g}$	Group flag vector
NID	node id
A	splitting attribute
T	splitting threshold
L	Leaf label
$\mathsf{Layer}^{(k)}$	k^{th} layer information
$\mathsf{State}^{(k)}$	Invariant state at k^{th} layer
n_k	Number of nodes in $\mathsf{Layer}^{(k)}$

Table 2: Variables and their description.

(3) $\mathbf{P}^i_k$ (for all $i \in [m]$) is the permutation that sorts $\mathbf{D}_k$ based on the i^{th} attribute value within each group; and

(4) $\mathbf{NID}$ is the vector of node ids of samples in $\mathbf{D}_k$. For example, if $\mathbf{D}_k[i]$ belongs to the subset of the dataset associated with the j^{th} node, $\mathbf{NID}[i] = j$.

To train the k^{th} layer ($k \in [h+1]$), parties input $\mathsf{State}^{(k)}$ to $\Pi_{\mathsf{TrainInternalLayer}}$ (Protocol 12) and receive $\mathsf{Layer}^{(k)}$ and $\mathsf{State}^{(k+1)}$ as output where $\mathsf{State}^{(k+1)}$ will serve as input to train the next layer.

As described in the cleartext decision tree algorithm, each node has a subset of the dataset on which it is trained. Nodes in the k^{th} layer partition the dataset which is encoded by a group flag vector $\mathbf{g}_k$ of length N as described in Section 3.3. The samples in the dataset belonging to the same group appear consecutively. That means for $i < j$ s.t. $\mathbf{g}_k[i] = \mathbf{g}_k[j] = 1$ and $\mathbf{g}_k[l] = 0$ for all $i < l < j$, all samples from $\mathbf{D}_k[i]$ to $\mathbf{D}_k[j-1]$ belong to the same group. As a layer is trained and new groups are formed, the partition of the dataset changes and the dataset $\mathbf{D}_k$ has to be rearranged to ensure that samples in the same group appear consecutively. Vectors $\mathbf{g}_k, \mathbf{P}^i_k$ and $\mathbf{NID}$ are also updated accordingly. The output $\mathsf{State}^{(k+1)}$ stores these updated vectors which can be used to train the next layer.

Table 2 summarizes the different variables that are used in the training protocol.

4.1 Efficiency Bottleneck in the State-of-art

The communication complexity of the state-of-art secure decision tree training protocol [18] is $O(hmN \log N)$. We identify and discuss the efficiency bottlenecks in this protocol and the challenges in removing these bottlenecks. Then, we redesign the training protocol to solve these challenges and remove the bottlenecks, achieving both asymptotic and concrete improvements in communication complexity over the state-of-art.

Recall that all the nodes in a layer are trained together and the layers of the tree are trained sequentially using the train layer subprotocol. Hence, we focus on the training of one internal node layer. The training of an internal node layer consists of the following steps:

(1) Compute Gini index for all possible attributes and thresholds within each group.
(2) Compute splitting attribute and threshold associated with minimum Gini index in each group. This gives the output Layer.
(3) Compute new partition of the dataset based on the splitting attributes and thresholds. This gives the output State which serves as input to the next layer.

Step (1): To compute the Gini index for all possible thresholds of one attribute, the parties need to sort the attribute and label vectors $(\mathbf{x}, \mathbf{y})$ according to the attribute values within each group. Then the parties can compute Expr. 1 securely for all groups with $O(N)$ communication. The cost of secure sorting is $O(N \log N)$ so the total cost in training due to Step (1) is $O(hmN \log N)$. Note that the cost of computing Gini index for all attributes and thresholds over an *unsorted* dataset requires $O(hmN^2)$ [3] commmunication. So even though the secure sorting is a bottleneck in this step, it is an essential operation.

Step (2): Minimum Gini index (Max Expr. 1) for each group is computed in two steps. First, parties compute the minimum Gini index for each attribute in each group using Π_{GroupMax} and then compute the minimum Gini index among all attributes using Π_{Max}([3], Fig. 13). Π_{GroupMax} contributes $O(hmN \log N)$ to the total cost while Π_{Max} contributes $O(hmN)$.

Step (3): To compute the new partition, parties securely compare the attribute values of each sample with the corresponding threshold. This splits each group into two new groups based on whether the test result is true or false. The dataset is sorted based on the outputs of the test i.e. all samples with false (0) result followed by all samples with true (1) result. This sorting arranges the dataset according to the new groups. The sorted dataset along with the updated group flag vector marking new groups is the input to train the next layer. Using the optimized $\Pi_{\mathsf{SortPermBit}}$ (Functionality 4), this step costs $O(hmN)$.

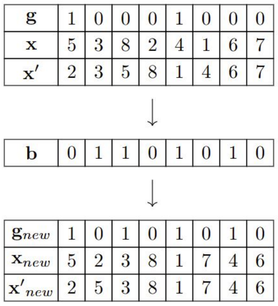
In [18], the secure sorting performed for every layer in **Step (1)** and GroupMax in **Step (2)** are the two bottlenecks in efficiency contributing $O(hmN \log N)$ each to communication.

4.2 Technical Overview

In this section, we discuss the challenges in removing these bottlenecks and how we can overcome these challenges.

Idea 1. First, we note that in [18], for each layer and attribute, parties have to sort, within the groups, the attribute values and label vector $(\mathbf{x}, \mathbf{y})$ according to the attribute values in **Step (1)**. Since the groups are different and contain different sets of elements in each layer, [18] had to run a sorting protocol to sort the attribute values and label vector in every layer (thus leading to a communication complexity of $O(hmN \log N)$ for this step). See Figure 2 for an example on how the attribute vector is updated from $\mathbf{x}$ to $\mathbf{x}_{\text{new}}$ as new groups are created due to the split.

The key observation to address this challenge is that even though $\mathbf{x}$ and $\mathbf{x}_{\text{new}}$ are different, $\mathbf{x}_{\text{new}}$ is simply a permutation of $\mathbf{x}$. Similarly, $\mathbf{x'}_{\text{new}}$ is a permutation of $\mathbf{x'}$. This means that there are permutations ϕ, ψ such that $\mathbf{x}_{\text{new}} = \text{Permute}(\mathbf{x}, \phi)$ and $\mathbf{x'}_{\text{new}} = \text{Permute}(\mathbf{x'}, \psi)$.



$\mathbf{g}$	1	0	0	0	1	0	0	0
$\mathbf{x}$	5	3	8	2	4	1	6	7
$\mathbf{x}'$	2	3	5	8	1	4	6	7

↓

$\mathbf{b}$	0	1	1	0	1	0	1	0

↓

$\mathbf{g}_{new}$	1	0	1	0	1	0	1	0
$\mathbf{x}_{new}$	5	2	3	8	1	7	4	6
$\mathbf{x}'_{new}$	2	5	3	8	1	7	4	6

Figure 2: x is an attribute vector with two groups and x′ is sorted x according to attribute value within groups. Let b be the test results. Then $\mathbf{x_{new}}$ is the new attribute vector with new groups which is obtained by stably sorting x based on b. $\mathbf{x'_{new}}$ is the new sorted attribute vector. As ordering of x changes due to new groups, the permutation that sorts $\mathbf{x_{new}}$ to $\mathbf{x'_{new}}$ also changes.

$$\begin{tikzcd}
\mathbf{x} \arrow[r, "\mathbf{P}"] \arrow[d, "\phi"'] & \mathbf{x}' \arrow[d, "\psi"] \\
\mathbf{x}_{new} \arrow[r, "\mathbf{P}'"'] & \mathbf{x}'_{new}
\end{tikzcd}$$

Figure 3: Borrowing notations from Figure 2, updated sorting permutation for $\mathbf{x_{new}}$ is $\mathbf{P}' = \psi \circ \mathbf{P} \circ \phi^{-1}$.

Given the old sorting permutation $\langle\mathbf{P}\rangle$ (that sorts $\mathbf{x}$), $\langle\phi^{-1}\rangle$ and $\langle\psi\rangle$, the new sorting permutation (that sorts $\mathbf{x}_{\text{new}}$) is $\mathbf{P}' = \psi\circ\mathbf{P}\circ\phi^{-1}$ (see Figure 3) which parties can compute with $O(N)$ using Π_{PermComp}. Moreover, we show that $\langle\phi^{-1}\rangle$ and $\langle\psi\rangle$ can be computed with $O(N)$ using $\Pi_{\mathsf{SortPermBit}}$. Thus, given the old sorting permutation $\langle\mathbf{P}\rangle$, parties can compute the new sorting permutation $\langle\mathbf{P}'\rangle$ with linear communication cost. This allows parties to update the sorting permutation whenever a layer is trained and new groups are formed. These sorting permutations can be applied to $(\mathbf{x}, \mathbf{y})$ directly with a cost of $O(N)$. Thus, the cost of **Step (1)** becomes $O(hmN)$ improving by a factor of $\log N$ over [18]. Parties will also update the sorting permutations in **Step (3)** whose cost remains $O(hmN)$. This now only requires a *one time* secure sorting to compute the sorting permutations initially (which costs $O(mN \log N)$). Moreover, this initial sort is concretely cheaper than the internal sort per layer done in [18] as the initial sort requires sorting by the attribute values whereas the latter sorts by both groups and attribute values.

Idea 2. The bottleneck in **Step (2)** is the GroupMax operation which requires $O(N \log N)$ communication. In [18], the parties invoke m instances of Π_{GroupMax} (over N-length vectors) to compute the threshold with minimum Gini index (Max Expr. 1) for each attribute. Then, for each $i \in [N]$, parties compute attribute (and threshold) with minimum Gini index by invoking N instances of Π_{Max} (over m-length vectors). We make two important observations here; (1) The cost of computing maximum of N values ($O(N)$) is less than the cost of computing group wise maximum (($O(N \log N)$)); and (2) reversing the order of operations does not change the output i.e. we can replace the m invocations of Π_{GroupMax} (on N-length vector) followed by N invocations of Π_{Max} (on m-length vector) with N invocations of Π_{Max} (on m-length vector) and 1 invocation of Π_{GroupMax} (on N-length vector). Reversing the order of these operations preserves functionality crucially because all attributes are grouped according to the same group boundaries. We formally show this in the proof of security of the protocol (Lemma 5.1). The proof relies on the fact that sorting according to different attributes in Step (1) is within groups and does not change the starting and ending index of a group. Thus, now the cost of step (2) becomes $O(hmN + hN \log N)$ from $O(hmN \log N)$ saving a factor of $\min(m, \log N)$.

Protocol overview. Putting these together, at a high-level, our protocol can be broken down into the following steps:

(1) Compute and store the sorting permutations for all attributes. This is a one-time (not per layer) set-up with communication cost $O(mN \log N)$.
(2) For all internal node layers:
  (a) Apply sorting permutation and compute Gini index. The communication cost is $O(mN)$.
  (b) Compute the splitting attribute and threshold in each group which has minimum Gini index. Communication cost is $O(mN + N \log N)$.
  (c) Compute the new partition of dataset based on splitting attribute and threshold, and the updated sorting permutation for all attributes. Communication is $O(mN)$.
(3) For the leaf node layer, compute the most occuring labels in each group. Communication cost is $O(N)$.

Hence, the total communication cost of our new training protocol is $O(mN \log N + hmN + hN \log N)$, which asymptotically improves upon the state-of-the-art [18] by a factor of $\min(h, m, \log N)$.

We describe the subprotocols to compute each step described above along with the improvements in efficiency in detail in the following sections. In Section 4.6, we combine all these subprotocols and present our end-to-end secure decision tree training protocol.

4.3 Set Up Phase/ Initial Sort

Parties compute the shares of permutation $\mathbf{P}_1^i$ for all $i \in [m]$ that sorts $\mathbf{D}$ according to the i^{th} attribute. These are computed once in the beginning and updated for each layer. Updating the permutation is a part of the internal layer training subprotocols and described in Section 4.4.4. We use Π_{SortPerm} that computes the sorting permutation with $O(N \log N)$ communication in $O(\log N)$ rounds for a vector of length N. $\Pi_{\mathsf{SetupPerm}}$ (Protocol 6) requires m calls to Π_{SortPerm}.

4.4 Training Internal Layers

Consider that we have trained nodes till layer $k-1$ ($k \in [h]$) and that we are now training nodes at layer k. The parties will have secret shares of $\mathsf{State}^{(k)} = \{\mathbf{D}_k, \mathbf{g}_k, \{\mathbf{P}_k^i\}_{i\in[m]}, \mathbf{NID}\}$ which satisfies the following invariant:

Protocol 6: $\Pi_{\mathsf{SetupPerm}}$

```
Input: Dataset $\langle \mathbf{D} \rangle = (\langle \mathbf{x}_1 \rangle, \ldots, \langle \mathbf{x}_m \rangle, \langle \mathbf{y} \rangle)$
Output: Sorting permutation $\left\langle \mathbf{P}_1^i \right\rangle$ for all $i \in [m]$
Cost: $O(mnN)$
for $i \in [m]$ do
    $\left\langle \mathbf{P}_1^i \right\rangle = \Pi_{\mathsf{SortPerm}} (\langle \mathbf{x}_i \rangle)$
end
Output $\left\{ \left\langle \mathbf{P}_1^i \right\rangle \right\}_{i \in [m]}$.
```

(1) $\mathbf{D}_k$ is the training dataset grouped by nodes in layer k,
(2) $\mathbf{g}_k$ is the group flag vector that marks the starting of groups in $\mathbf{D}_k$,
(3) $\mathbf{P}_k^i$ (for all $i \in [m]$) is the permutation that sorts $\mathbf{D}_k$ based on i^{th} attribute within each group and
(4) vector $\mathbf{NID}$ is the node id (or group id) of each sample in $\mathbf{D}_k$. For example if $\mathbf{D}_k[i]$ belongs to group of j^{th} node, $\mathbf{NID}[i] = j$. There are $n_k = 2^{k-1}$ nodes in k^{th} layer.

The output of training the k^{th} internal node layer will be $\left\langle \mathsf{Layer}^{(k)} \right\rangle$ defined in Section 4 and $\left\langle \mathsf{State}^{(k+1)} \right\rangle$ i.e. updated state required to train the next layer.

For the base case, $\langle \mathbf{D}_1 \rangle = \langle \mathbf{D} \rangle, \langle \mathbf{g}_1 \rangle = \langle [1, 0, \ldots, 0] \rangle, \langle \mathbf{NID} \rangle = \langle [1, 1, \ldots, 1] \rangle$ as Layer 1 only has the root node. Hence, there is no partitioning of nodes. Moreover, from the setup phase, we have $\left\langle \mathbf{P}_1^i \right\rangle = \Pi_{\mathsf{SetupPerm}}(\langle \mathbf{x}_i \rangle)$. Thus,

$$\left\langle \mathsf{State}^{(1)} \right\rangle = \left\{ \langle \mathbf{D}_1 \rangle, \langle \mathbf{g}_1 \rangle, \{ \left\langle \mathbf{P}_k^i \right\rangle \}_{i \in [m]}, \langle \mathbf{NID} \rangle \right\}$$

satisfies the invariant for the base case.

The $\mathsf{TrainInternalLayer}$ functionality takes as input k and shares of $\mathsf{State}^{(k)}$, and outputs shares of $\mathsf{Layer}^{(k)}$, $\mathsf{State}^{(k+1)}$ where $\mathsf{Layer}^{(k)}$ is the k^{th} decision tree layer and $\mathsf{State}^{(k+1)}$ is the intermediary output used to train the next layer. Protocol 12 securely computes this functionality.

In Lemma 5.3, we show that if the input $\mathsf{State}^{(k)}$ to algorithm 12 satisfies the invariant described above, the output $\mathsf{State}^{(k+1)}$ also satisfies the invariant. Therefore by induction, $\left\langle \mathsf{State}^{(k)} \right\rangle$ satisfies the invariant for all $k \in [h+1]$.

The $\Pi_{\mathsf{TrainInternalLayer}}$ protocol has total $O(mN + N \log N)$ communication cost and consists of the following 4 steps:

(1) Compute all possible thresholds and Gini index from the resulting split for each attribute. (Section 4.4.1)
(2) Select attribute and threshold for best split for all layer nodes. (Section 4.4.2)
(3) Apply tests to compute new groups after split. (Section 4.4.3)
(4) Update the invariant state. (Section 4.4.4)
(5) Output the final output of decision tree training $\mathsf{Layer}^{(k)}$. (Section 4.4.5)

We construct subprotocols for each of these steps in the following sections that outline various optimizations over [18].

4.4.1 Computing Gini index for an attribute. $\mathsf{ComputeGini}$ takes an attribute $(\mathbf{x})$ and label $(\mathbf{y})$ vector, both sorted by attribute value and computes all possible thresholds and the corresponding Gini index of the split. The Gini index (s) is stored as a tuple of integers (p, q) where $s = p/q$. This is done to avoid the expensive secure division operation. Since we only need to perform comparisons on Gini indices, $s_1 < s_2$ is equivalent to $p_1 q_2 < p_2 q_1$ where $s_i = (p_i, q_i)$. $\Pi_{\mathsf{ComputeGini}}$ (Protocol 7) securely computes the functionality. The sorted attribute vector allows us to compute the split for each threshold efficiently. If threshold t is such that $\mathbf{x}[i] < t < \mathbf{x}[i+1]$, then the two partitions of the dataset are $\{\mathbf{x}[1], \ldots, \mathbf{x}[i]\}$ and $\{\mathbf{x}[i+1], \ldots, \mathbf{x}[N]\}$ respectively since the attribute vector is sorted. The threshold $\mathbf{t}[i]$ corresponding to an attribute element $\mathbf{x}[i]$ is set to $(\mathbf{x}[i] + \mathbf{x}[i+1])/2$. However, this involves the expensive secure division operation, so we modify the test $\mathbf{x}_j < \mathbf{t}$ to $2\mathbf{x}_j < \mathbf{t}'$ such that $\mathbf{t}' = 2\mathbf{t}$ just as in [18]. $\mathbf{t}'$ would then be computed as $\mathbf{x}[i] + \mathbf{x}[i+1]$.

Protocol $\Pi_{\mathsf{ComputeGini}}$ is the same as the protocol for computing Gini index in [18] except for one optimization. We take advantage of the observation that while the Group Prefix Sum of $\mathbf{y}$ labels changes based on ordering of samples, the Group Sum remains same. In [18], the Group Sum of $\mathbf{y}$ labels is computed for each attribute (m GroupSum invocations). We instead compute Group Sum of $\mathbf{y}$ labels only once resulting in saving $25(m-1)Nhl$ bits of communication. This is only a minor optimization and does not change the asymptotic complexity of the protocol.

Let $\langle \mathbf{sy}_0 \rangle, \langle \mathbf{sy}_1 \rangle$ be Group Sum of $\mathbf{y}$ and $\neg\mathbf{y}$ respectively which are computed once and passed as arguments to $\Pi_{\mathsf{ComputeGini}}$. $\langle \mathbf{x} \rangle, \langle \mathbf{y} \rangle$ are attribute values and labels (sorted according to attribute) respectively. The protocol outputs shares of $\mathbf{t}, \mathbf{s}$ where $\mathbf{t}[i]$ is a threshold that splits the group at the i^{th} sample and $\mathbf{s}[i]$ is the Gini index resulting from this split. Note that $\mathbf{t}[N]$ is set to MinValue, the smallest representable value, such that $\mathbf{t}[N] <$ MinValue always gives a False result. This is to ensure non-zero number of samples in each split.

In case of duplicate values, we split the group at the last instance of the recurring value, as in [18]. For this, we set the threshold against other instances of this value (apart from the last) as MinValue, so that the result of comparison is always False i.e. we never split at instances of this value apart from the last. If $\mathbf{x}[i] = \mathbf{x}[i+1]$, for $1 \leq \alpha \leq i < \beta < N$, we set $\mathbf{t}[i] =$ MinValue for $\alpha \leq i < \beta$ and $\mathbf{t}[\beta] = x[\beta] + x[\beta+1]$.

The correctness follows from Expr. 1 which is derived in [3]. $\Pi_{\mathsf{ComputeGini}}$ requires 2 calls to $\Pi_{\mathsf{GroupPrefixSum}}$, $11N$ calls to Π_{MULT} and N calls to Π_{EQ}.

4.4.2 Test Selection. Functionality $\mathsf{TestSelection}$ takes as input the database $\langle \mathbf{D}_k \rangle$, group flag vector $\langle \mathbf{g}_k \rangle$ and sorting permutations $\left\{ \left\langle \mathbf{P}_k^i \right\rangle \right\}_i$, and outputs the splitting attributes $\langle \mathbf{A} \rangle$ and thresholds $\langle \mathbf{T} \rangle$ where $\mathbf{A}[j], \mathbf{T}[j]$ are the attribute and threshold with minimum Gini index for the node that i belongs to. Protocol $\Pi_{\mathsf{TestSelection}}$ (Protocol 8) computes the functionality.

$\Pi_{\mathsf{TestSelection}}$ invokes $\Pi_{\mathsf{ComputeGini}}$ as a subprotocol to compute the Gini Index for all possible attributes and thresholds, $\{\langle \mathbf{s}_i \rangle, \langle \mathbf{t}_i \rangle\}_{i \in [m]}$ where $\mathbf{t}_i[j]$ for $j \in [N]$ is the threshold for splitting the dataset on the i^{th} attribute and $\mathbf{s}_i[j]$ is the resulting Gini index from this split.

In [18], Test Selection is done by invoking m instances of Group Max over N-length vectors $\langle \mathbf{S}'_i \rangle, \langle \mathbf{T}'_i \rangle = \Pi_{\mathsf{GroupMax}}(\mathbf{g}, \mathbf{s}_i; \mathbf{t}_i)$ to

Protocol 7: $\Pi_{\mathsf{ComputeGini}}$

```
Input: $\langle \mathbf{g} \rangle, \langle \mathbf{x} \rangle, \langle \mathbf{y} \rangle, \langle \mathbf{sy}_0 \rangle, \langle \mathbf{sy}_1 \rangle$
Output: $\langle \mathbf{s} \rangle, \langle \mathbf{t} \rangle$ of length $N$.
Cost: $O(N)$
$\langle \mathbf{u_0} \rangle = \text{GroupPrefixSum}(\langle \mathbf{y} \rangle, \langle \mathbf{g} \rangle)$
$\langle \mathbf{u_1} \rangle = \text{GroupPrefixSum}(\langle \neg\mathbf{y} \rangle, \langle \mathbf{g} \rangle)$
$\langle \mathbf{w_b} \rangle = \langle \mathbf{sy}_b \rangle - \langle \mathbf{u_b} \rangle$ for $b \in \{0,1\}$
$\langle \mathbf{w} \rangle = \langle \mathbf{w_0} \rangle + \langle \mathbf{w_1} \rangle$ and $\langle \mathbf{u} \rangle = \langle \mathbf{u_0} \rangle + \langle \mathbf{u_1} \rangle$
$\langle \mathbf{p} \rangle = \langle \mathbf{w} \rangle \times \left(\langle \mathbf{u_0} \rangle^2 + \langle \mathbf{u_1} \rangle^2\right) + \langle \mathbf{u} \rangle \times \left(\langle \mathbf{w_0} \rangle^2 + \langle \mathbf{w_1} \rangle^2\right)$
$\langle \mathbf{q} \rangle = \langle \mathbf{u} \rangle \times \langle \mathbf{w} \rangle$
$\langle \mathbf{s} \rangle = (\langle \mathbf{p} \rangle, \langle \mathbf{q} \rangle)$
$\langle \mathbf{t}[i] \rangle = \langle \mathbf{x}[i] \rangle + \langle \mathbf{x}[i+1] \rangle$ for all $i \in [N-1]$,
    $\langle \mathbf{t}[N] \rangle = \text{MinValue}$
$\langle \mathbf{r}[i] \rangle = \langle \mathbf{g}[i+1] \rangle$ OR $\left\langle \mathbf{x}[i] \overset{?}{=} \mathbf{x}[i+1] \right\rangle$ for $i \in [N-1]$,
    $\langle \mathbf{r}[N] \rangle = 1$
$\langle \mathbf{s} \rangle, \langle \mathbf{t} \rangle = \text{IfElse}(\langle \mathbf{r} \rangle; \text{MinValue}, \text{MinValue}; \langle \mathbf{s} \rangle, \langle \mathbf{t} \rangle)$
```

compute the best threshold for the i^{th} attribute for all $i \in [m]$ followed by N invocations of Π_{Max} ($\forall j \in [N]$) over m-length vectors to compute the best attribute and corresponding threshold

$$\mathbf{A}[j], \mathbf{T}[j] = \Pi_{\mathsf{Max}}\left(\{\mathbf{S'}_i[j]; i, \mathbf{T'}_i[j]\}_{i \in [m]}\right)$$

We note that the order of operations can be reversed, reducing the number of comparisons from $mN \log N + mN$ to $N \log N + mN$. Note that this optimization alone does not reduce the asymptotic cost of [18] and is only useful when combined with our new training protocol that uses secret shared permutations instead of repeated secure sorting.

We know that $\{\mathbf{s}_i, \mathbf{t}_i\}_{i \in [m]}$ are correctly computed from the correctness of $\Pi_{\mathsf{ComputeGini}}$. To prove correctness of $\Pi_{\mathsf{TestSelection}}$, we have to show that $\mathbf{A}, \mathbf{T}$ computed in steps $6-9$ of Protocol 8 are correct. Step 10 merely checks if the group of a node is already homogenous (all labels are equal). If it is homogeneous, no more splitting is required and therefore, the threshold is set to MinValue. We show correctness of $\Pi_{\mathsf{TestSelection}}$ in Lemma 5.1 where we prove that reversing the order of Π_{GroupMax} and Π_{Max} does not affect the output.

$\Pi_{\mathsf{TestSelection}}$ requires $2m$ calls to $\Pi_{\mathsf{ApplyPerm}}$, m calls to $\Pi_{\mathsf{ComputeGini}}$, N calls to Π_{VectMax}, 1 call to Π_{GroupMax}, $3N$ calls to Π_{MULT} and $2N$ calls to Π_{EQ}.

4.4.3 *Applying Tests.*

Once parties have $\langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle$; parties compute the partition of nodes based on comparison of selected attributes with their corresponding thresholds.

The cost of $\Pi_{\mathsf{ApplyTest}}$ is $O(hmN)$, which is insignificant in the total communication and we directly use the subprotocol from [18]. If $\mathbf{A}[j] = i^*$, the i^{*th} attribute value has to be selected for the j^{th} sample (i.e. $\mathbf{x}_{i^*}[j]$) to compare with the threshold $\mathbf{T}[j]$. To fetch $\langle \mathbf{x}_{i^*}[j] \rangle$ in an oblivious manner, parties have to compute $\langle \mathbf{e}_{i^*} \rangle \in \{0,1\}^m$ from $\langle i^* \rangle$ which is its one hot encoding. We compute the one hot vector using m comparisons as:

$$\langle \mathbf{e}_{i^*} \rangle = \left(\langle i^* \rangle \overset{?}{=} 1, \langle i^* \rangle \overset{?}{=} 2, \ldots, \langle i^* \rangle \overset{?}{=} m\right)$$

Protocol 8: $\Pi_{\mathsf{TestSelection}}$

```
Input: $\langle \mathbf{D}_k \rangle, \langle \mathbf{g}_k \rangle, \left\{\left\langle \mathbf{P}^i_k \right\rangle\right\}_i, \langle \mathbf{sy}_0 \rangle, \langle \mathbf{sy}_1 \rangle$
Output: $\langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle$ of size $N$.
Cost: $O(mn + nN)$
1  for $i \in [m]$ do
2      $\langle \mathbf{u}_i \rangle = \Pi_{\mathsf{ApplyPerm}}\left(\langle \mathbf{x}_i \rangle, \left\langle \mathbf{P}^i_k \right\rangle\right)$
3      $\langle \mathbf{v}_i \rangle = \Pi_{\mathsf{ApplyPerm}}\left(\langle \mathbf{y} \rangle, \left\langle \mathbf{P}^i_k \right\rangle\right)$
4      $\langle \mathbf{s}_i \rangle, \langle \mathbf{t}_i \rangle = \Pi_{\mathsf{ComputeGini}}(\langle \mathbf{g}_k \rangle, \langle \mathbf{u}_i \rangle, \langle \mathbf{v}_i \rangle)$
5  end
6  for $j \in [N]$ do
7      $\langle \mathbf{s}[j] \rangle, \langle \mathbf{t}[j] \rangle, \langle \mathbf{a}[j] \rangle =$
           $\Pi_{\mathsf{Max}}\left(\{\langle \mathbf{s}_i[j] \rangle; \langle \mathbf{t}_i[j] \rangle, i\}_{i \in [m]}\right)$
8  end
9  $\langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle = \Pi_{\mathsf{GroupMax}}(\langle \mathbf{g}_k \rangle, \langle \mathbf{s} \rangle; \langle \mathbf{a} \rangle, \langle \mathbf{t} \rangle)$
10 $\langle \mathbf{f} \rangle = \left(\langle \mathbf{sy}_0 \rangle \overset{?}{=} 0\right)$ OR $\left(\langle \mathbf{sy}_1 \rangle \overset{?}{=} 0\right)$
11 $\langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle = \text{IfElse}(\langle \mathbf{f} \rangle; 1, \text{MinValue}; \langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle)$
```

Then the parties can compute $\mathbf{x}_{i^*}$ with dot product $\mathbf{x} \cdot \mathbf{e}_{i^*}$ and compare it with the threshold. $\Pi_{\mathsf{ApplyTest}}$ (Protocol 9) requires N calls to Π_{MULT}, N calls to Π_{LT} and mN calls to Π_{EQ}.

Protocol 9: $\Pi_{\mathsf{ApplyTest}}$

```
Input: $\langle \mathbf{D}_k \rangle, \langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle$
Output: $\langle \mathbf{b} \rangle \in \{0,1\}^N$
Cost: $O(mN)$
for $i \in [N]$ do
    $\langle \mathbf{e}_{i^*} \rangle = \text{OneHotEnc}(\langle \mathbf{A}[i] \rangle)$
    $\langle \mathbf{b}[i] \rangle = \left(2.\langle \mathbf{D}_k[i] \cdot \mathbf{e}_{i^*} \rangle \overset{?}{<} \langle \mathbf{T}[i] \rangle\right)$
end
Output $\langle \mathbf{b} \rangle$.
```

4.4.4 *Updating flag vector and state.*

After applying tests, each group is divided into two new groups based on whether the comparison result $\mathbf{b}$ is equal to 0 or 1. To avoid sorting the dataset at each layer, parties have to update $\text{State}^{(k+1)}$ such that the new groups satisfy the invariant described in Section 4.4. Functionality UpdateState takes shares of $\text{State}^{(k)}$ and $\mathbf{b}$ as input and outputs $\text{State}^{(k+1)}$ such that $\text{State}^{(k+1)}$ satisfies the invariant. In [18], the parties only update the group flag vector and rearrange the dataset so that entries of the same group appear together. UpdateState has to additionally update the sorting permutations so that we can avoid sorting and use the output $\text{State}^{(k+1)}$ to train the next layer of the decision tree. $\Pi_{\mathsf{UpdateState}}$ (Protocol 10) securely computes this functionality.

Updating sorting permutations results in additional $O(mN)$ communication for a layer. The cost of update in [18] is $O(hmN)$ so the asymptotic cost does not increase with this change. Moreover, we are able to eliminate repeated calls to secure sorting, eliminating the efficiency bottleneck.

We create a new flag vector $\mathbf{g}$ that marks the first i such that $\mathbf{b}[i] = 0$ and $\mathbf{b}[i] = 1$ in each group using the $\Pi_{\text{GroupFirstOne}}$ protocol from [18] (described in section 3.3). We do so as the first instance of 0 and 1 in each group from the comparison results indicates the starting of new groups now. We sort $\mathbf{D}_k$ and $\mathbf{g}$ based on values of $\mathbf{b}$ to obtain $\langle \mathbf{D}_{k+1}\rangle, \langle \mathbf{g}_{k+1}\rangle$ We compute the updated permutations $\left\{\mathbf{P}^i_{k+1}\right\}_{i\in[m]}$.

$\Pi_{\text{UpdateState}}$ requires $3m + 3$ calls to $\Pi_{\text{ApplyPerm}}$ and m calls to Π_{PermComp}, 2 calls to $\Pi_{\text{GroupFirstOne}}$ and $m+1$ calls to $\Pi_{\text{SortPermBit}}$. We provide a correctness proof for the UpdateState protocol in Lemma 5.3.

Protocol 10: $\Pi_{\text{UpdateState}}$

```
Input: $\left\langle \text{State}^{(k)} \right\rangle = \left( \langle \mathbf{D}_k \rangle, \langle \mathbf{g}_k \rangle, \left\{ \left\langle \mathbf{P}^i_k \right\rangle \right\}_i, \langle \mathbf{NID} \rangle \right), \langle \mathbf{b} \rangle, k$
Output: $\left\langle \text{State}^{(k+1)} \right\rangle$
Cost: $O(mN)$
$\langle \mathbf{Q} \rangle = \Pi_{\text{SortPermBit}}(\langle \mathbf{b} \rangle)$
$\langle \mathbf{D}_{k+1} \rangle = \Pi_{\text{ApplyPerm}}(\langle \mathbf{D}_k \rangle, \langle \mathbf{Q} \rangle)$
$\langle \mathbf{NID} \rangle = 2^{k-1} \langle \mathbf{b} \rangle + \langle \mathbf{NID} \rangle$
$\left\langle \mathbf{NID}' \right\rangle = \Pi_{\text{ApplyPerm}}(\langle \mathbf{NID} \rangle, \langle \mathbf{Q} \rangle)$
$\langle \mathbf{g} \rangle = \Pi_{\text{GroupFirstOne}}(\langle \mathbf{g}_k \rangle, \langle \mathbf{b} \rangle)$
      $+\Pi_{\text{GroupFirstOne}}(\langle \mathbf{g}_k \rangle, \langle \neg\mathbf{b} \rangle)$
$\langle \mathbf{g}_{k+1} \rangle = \Pi_{\text{ApplyPerm}}(\langle \mathbf{g} \rangle, \langle \mathbf{Q} \rangle)$
for $i \in [m]$ do
    $\left\langle \mathbf{b}^i \right\rangle = \Pi_{\text{SortPermBit}}\left(\langle \mathbf{b} \rangle, \left\langle \mathbf{P}^i_k \right\rangle\right)$
    $\left\langle \mathbf{Q}^i \right\rangle = \Pi_{\text{ApplyPerm}}\left(\left\langle \mathbf{P}^i_k \right\rangle, \langle \mathbf{Q} \rangle\right)$
    $\left\langle \mathbf{R}^i \right\rangle = \Pi_{\text{ApplyPerm}}\left(\left\langle \mathbf{b}^i \right\rangle\right)$
    $\left\langle \mathbf{P}^i_{k+1} \right\rangle = \Pi_{\text{PermComp}}\left(\left\langle \mathbf{R}^i \right\rangle, \left\langle \mathbf{Q}^i \right\rangle\right)$
end
Output $\left\langle \text{State}^{(k+1)} \right\rangle =$
  $\left( \langle \mathbf{D}_{k+1} \rangle, \langle \mathbf{g}_{k+1} \rangle, \left\{ \left\langle \mathbf{P}^i_{k+1} \right\rangle \right\}_{i\in[m]}, \left\langle \mathbf{NID}' \right\rangle \right)$.
```

4.4.5 Storing values of splitting nodes. Training each layer also outputs the splitting attribute and threshold for all nodes in the layer (stored in tuple $\text{Layer}^{(k)}$). Subprotocol $\Pi_{\text{StoreLayer}}$ (Protocol 11) which outputs $\text{Layer}^{(k)}$ tuple is not a bottleneck in communication efficiency ($O(hmN)$) and we use it as is from [18].

Parties have $\langle \mathbf{NID} \rangle, \langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle$ containing node id, attribute and threshold but the values repeat (same value for all elements in one group). We want to store the output of training of internal nodes in $\langle \mathbf{NID}_k \rangle, \langle \mathbf{A}_k \rangle, \langle \mathbf{T}_k \rangle$ where $\mathbf{NID}_k[i], \mathbf{A}_k[i], \mathbf{T}_k[i]$ stores the node id, attribute and threshold of i^{th} node. For the leaf node layer, parties output $\langle \mathbf{NID}_{h+1} \rangle, \langle \mathbf{L}_{h+1} \rangle$.

$\Pi_{\text{StoreLayer}}$ requires 1 call to $\Pi_{\text{SortPermBit}}$ and 3 calls to $\Pi_{\text{ApplyPerm}}$.

Protocol 11: $\Pi_{\text{StoreLayer}}$

```
Input: $\langle \mathbf{NID} \rangle, \langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle, \langle \mathbf{g}_k \rangle$ of length $N$
Output: $\langle \mathbf{NID}_k \rangle, \langle \mathbf{A}_k \rangle, \langle \mathbf{T}_k \rangle$ of length $n_k = \min(2^{k-1}, N)$
Cost: $O(N)$
$\langle \mathbf{P} \rangle = \Pi_{\text{SortPermBit}}(\neg \langle \mathbf{g}_k \rangle)$
$\langle \mathbf{NID}_k \rangle = \Pi_{\text{ApplyPerm}}(\langle \mathbf{NID} \rangle, \langle \mathbf{P} \rangle)$     //First $n_k$ terms
$\langle \mathbf{A}_k \rangle = \Pi_{\text{ApplyPerm}}(\langle \mathbf{A} \rangle, \langle \mathbf{P} \rangle)$     //First $n_k$ terms
$\langle \mathbf{T}_k \rangle = \Pi_{\text{ApplyPerm}}(\langle \mathbf{T} \rangle, \langle \mathbf{P} \rangle)$     //First $n_k$ terms
Output $\left\langle \text{Layer}^{(k)} \right\rangle = (\langle \mathbf{NID}_k \rangle, \langle \mathbf{A}_k \rangle, \langle \mathbf{T}_k \rangle)$.
```

4.4.6 End to end internal layer training. With the subprotocols constructed in Sections 4.4.1 to 4.4.5, we can describe the complete internal training protocol in Protocol 12. Training one internal layer requires 2 calls to Π_{GroupSum}, 1 call to $\Pi_{\text{TestSelection}}$, 1 call to $\Pi_{\text{ApplyTests}}$, 1 call to $\Pi_{\text{StoreLayer}}$ and 1 call to $\Pi_{\text{UpdateState}}$.

Protocol 12: $\Pi_{\text{TrainInternalLayer}}$

```
Input: $\left\langle \text{State}^{(k)} \right\rangle = \left( \langle \mathbf{D}_k \rangle, \langle \mathbf{g}_k \rangle, \left\{ \left\langle \mathbf{P}^i_k \right\rangle \right\}_i, \langle \mathbf{NID} \rangle \right), k$
Output: $\left\langle \text{Layer}^{(k)} \right\rangle, \left\langle \text{State}^{(k+1)} \right\rangle$
Cost: $O(mN + nN)$
$\langle \mathbf{sy}_0 \rangle = \text{GroupSum}(\langle \mathbf{y}_k \rangle, \langle \mathbf{g}_k \rangle)$
$\langle \mathbf{sy}_1 \rangle = \text{GroupSum}(\langle \neg\mathbf{y}_k \rangle, \langle \mathbf{g}_k \rangle)$
$\langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle = \Pi_{\text{TestSelection}}\left( \langle \mathbf{D}_k \rangle, \langle \mathbf{g}_k \rangle, \left\{ \left\langle \mathbf{P}^i_k \right\rangle \right\}_i \right)$
$\left\langle \text{Layer}^{(k)} \right\rangle = \Pi_{\text{StoreLayer}}(\langle \mathbf{NID} \rangle, \langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle, \langle \mathbf{g}_k \rangle, k)$
$\langle \mathbf{b} \rangle = \Pi_{\text{ApplyTest}}(\langle \mathbf{D}_k \rangle, \langle \mathbf{A} \rangle, \langle \mathbf{T} \rangle)$
$\left\langle \text{State}^{(k+1)} \right\rangle = \Pi_{\text{UpdateState}}\left( \left\langle \text{State}^{(k)} \right\rangle, \langle \mathbf{b} \rangle, k \right)$
Output $\left\langle \text{Layer}^{(k)} \right\rangle, \left\langle \text{State}^{(k+1)} \right\rangle$
```

4.5 Training Leaf Layer

After training all internal layers, we have to train the final leaf node layer. The decision tree training halts on the leaf node and outputs the label corresponding to the leaf label as the sample's classification result. The label of a leaf node is the value $\in \{0, 1\}$ that occurs most frequently in $\mathbf{y}$ for samples in the training dataset belonging to this node. This can be computed using Π_{GroupSum}. The final layer labels are stored in two vectors $\mathbf{NID}, \mathbf{L}$ of length n_{h+1} where $\mathbf{NID}$ stores the node id and $\mathbf{L}$ stores the corresponding labels. We can modify the $\Pi_{\text{StoreLayer}}$ accordingly to store the leaf layer output. $\Pi_{\text{TrainLeafLayer}}$ (Protocol 13) requires 2 calls to Π_{GroupSum}, 1 call to $\Pi_{\text{StoreLayer}}$ and N calls to Π_{LT}.

4.6 End to end training

Finally, we combine sub-protocols $\Pi_{\text{SetupPerm}}$, $\Pi_{\text{TrainInternalLayer}}$ and $\Pi_{\text{TrainLeafLayer}}$ to obtain an end to end privacy preserving decision tree training protocol Π_{Train} (Protocol 15 in Appendix C).

The communication cost of $\Pi_{\text{SetupPerm}}$, $\Pi_{\text{TrainInternalLayer}}$ and $\Pi_{\text{TrainLeafLayer}}$ are $O(mN \log N)$, $O(mN + N \log N)$ and $O(N)$ respectively. The total communication cost of $\Pi_{\text{TrainDecisionTree}}$ is $O(mN \log N + h(mN + N \log N))$ which improves by a factor of $\min(h, m, \log N)$ over the state-of-the-art [18].

```
Protocol 13: $\Pi_{\mathsf{TrainLeafLayer}}$
    Input: $\left\langle \mathsf{State}^{(k)} \right\rangle$
    Output: $\left\langle \mathsf{Layer}^{(k)} \right\rangle = (\langle \mathbf{NID}_k \rangle, \langle \mathbf{L}_k \rangle)$
    Cost: $O(N)$
    $\langle \mathbf{L} \rangle = \mathrm{GroupSum}(\langle \mathbf{y}_k \rangle, \langle \mathbf{g}_k \rangle) > \mathrm{GroupSum}(\langle \neg \mathbf{y}_k \rangle, \langle \mathbf{g}_k \rangle)$
    $\langle \mathbf{NID}_k \rangle, \langle \mathbf{L}_k \rangle = \Pi_{\mathsf{StoreLayer}}(\langle \mathbf{NID} \rangle, \langle \mathbf{L} \rangle, \langle \mathbf{g}_k \rangle, k)$
    Output $\left\langle \mathsf{Layer}^{(k)} \right\rangle = (\langle \mathbf{NID}_k \rangle, \langle \mathbf{L}_k \rangle)$.
```

5 Security Proof

Let Train be the functionality that takes as input the shares of labelled dataset $\langle \mathbf{D} \rangle$ and height h and outputs shares of trained decision tree of height h i.e. $\left\{ \mathsf{Layer}^{(i)} \right\}_i$. To prove the security of our training protocol, we show that the view of a semi-honest adversary corrupting one of the parties in the real execution of protocol Π_{Train} can be simulated by a simulator with access to functionality Train in the ideal world.

The protocols for building blocks $\Pi_{\mathsf{MULT}}, \Pi_{\mathsf{A2B}}, \Pi_{\mathsf{LT}}, \Pi_{\mathsf{EQ}}$ and Π_{Shuffle}, $\Pi_{\mathsf{ApplyPerm}}, \Pi_{\mathsf{PermComp}}$ were already proven UC-secure in [6, 30] and [10, 23] respectively. The groupwise operations also inherit the security of their underlying subprotocols and are UC-secure [18]. Therefore, our end-to-end decision tree training protocol is also secure under the Universal Composability (UC) framework [12] as it is just a series of concurrent and sequential compositions of these building blocks.

The correctness of subprotocols $\Pi_{\mathsf{ComputeGini}}$, $\Pi_{\mathsf{ApplyTest}}$ and $\Pi_{\mathsf{StoreLayer}}$ was shown in [18]. We show the correctness of new protocols $\Pi_{\mathsf{TestSelection}}$ and $\Pi_{\mathsf{UpdateState}}$.

5.1 Correctness of TestSelection

Lemma 5.1. *Protocol* $\Pi_{\mathsf{TestSelection}}$ *defined in Protocol 8 correctly computes the* TestSelection *functionality from Section 4.4.2.*

Proof. To demonstrate correctness of $\Pi_{\mathsf{TestSelection}}$, we prove (Lemma 5.2) that reversing the order of invocations of Π_{GroupMax} and Π_{Max} does not change the output in steps $6-9$ in $\Pi_{\mathsf{TestSelection}}$ (Protocol 8). $\{\mathbf{s}_i\}_{i \in [m]}$ are the length N vectors for Gini index for all attributes which are correctly computed from the correctness of $\Pi_{\mathsf{ComputeGini}}$ protocol. □

Suppose $\{\mathbf{s}_i\}_{i \in [m]}$ are m vectors each of length N, $\mathbf{g}$ is group flag vector denoting groups in $\mathbf{s}$ and $j \in [N]$ is an index. l_j, r_j are the leftmost and rightmost indices of the group that j belongs to (see Section 3.3). GroupMax and Max operations are defined as follows:

$$\mathbf{a} = \Pi_{\mathsf{Max}}(\mathbf{s}_1, \ldots, \mathbf{s}_m) \quad \text{and} \quad \mathbf{b}_i = \Pi_{\mathsf{GroupMax}}(\mathbf{g}, \mathbf{s}_i)$$

where

$$\mathbf{a}[j] = \mathrm{Max}_{i \in [m]}(\mathbf{s}_i[j]) \quad \text{and} \quad \mathbf{b}_i[j] \quad = \mathrm{Max}_{l_j \le k \le r_j}(\mathbf{s}_i[k])$$

Lemma 5.2. *Borrowing notations from above, we have*

$$\Pi_{\mathsf{GroupMax}}(\mathbf{g}, \mathbf{a}) = \Pi_{\mathsf{Max}}(\mathbf{b}_1, \ldots, \mathbf{b}_m)$$

(n, m, h)	Comm. ([18])	Comm. (Ours)	IF
(13, 11, 4)	14.3	3.6	4.0
(13, 20, 4)	25.9	6.1	4.2
(13, 11, 10)	35.8	6.5	5.5
(13, 11, 20)	71.6	11.4	6.3
(16, 11, 10)	310.8	53.2	5.8
(16, 20, 20)	1, 123.7	151.6	7.4
(16, 40, 50)	5, 599.8	631.4	8.9
(18, 20, 20)	4, 730.4	612.7	7.7
(18, 50, 50)	29, 453.8	3, 117.7	9.4
(18, 100, 100)	119, 282.2	11, 539.6	10.3
(19, 11, 10)	2, 680.5	434.9	6.2
(19, 20, 50)	24, 239.6	2, 793.4	8.7
(19, 40, 40)	38, 667.5	4, 169.5	9.3

Table 3: Communication (GB) of Decision Tree Training for a dataset containing $N = 2^n$ training samples, m attributes to construct a decision tree of height h. IF represents the Improvement Factor i.e. Cost ([18]) / Cost (Ours)

Proof. Let LHS = $\mathbf{c}$ and RHS = $\mathbf{d}$ where $\mathbf{c}, \mathbf{d}$ are vectors of length N. We show that for all $j \in [N]$, $\mathbf{c}[j] = \mathbf{d}[j]$. For any $j \in [N]$:

$$\begin{aligned} \mathbf{c}[j] &= \mathrm{Max}_{l_j \le k \le r_j}(\mathbf{a}[k]) = \mathrm{Max}_{l_j \le k \le r_j}\left(\mathrm{Max}_{i \in [m]}(\mathbf{s}_i[k])\right) \\ &= \mathrm{Max}_{\substack{i \in [m] \\ l_j \le k \le r_j}}(\mathbf{s}_i[k]) \end{aligned}$$

Similarly we have

$$\begin{aligned} \mathbf{d}[j] &= \mathrm{Max}_{i \in [m]}(\mathbf{b}_i[j]) = \mathrm{Max}_{i \in [m]}\left(\mathrm{Max}_{l_j \le k \le r_j}(\mathbf{s}_i[k])\right) \\ &= \mathrm{Max}_{\substack{i \in [m] \\ l_j \le k \le r_j}}(\mathbf{s}_i[k]) \end{aligned}$$

Therefore, LHS = RHS. □

5.2 Correctness of UpdateState

Lemma 5.3. *Protocol* $\Pi_{\mathsf{UpdateState}}$ *defined in Algorithm 10 correctly computes the* UpdateState *functionality from Section 4.4.4.*

We prove this Lemma in Appendix F.

Applying the composability theorem [12], we can then show:

Theorem 5.4. *Protocol* Π_{Train} *defined in Algorithm 15 securely realizes the* Train *functionality.*

6 Experiments

Hamada *et al.* [18] is the state-of-the-art for secure decision tree training and we use its implementation[3] provided in the MP-SPDZ framework [1, 2, 22] as the baseline. We consider the setting of three parties with one semi-honest corruption. We implement [4] our secure decision tree training protocols in the MP-SPDZ framework and use the same implementations of the building blocks (Section 3) as the baseline for a fair comparison. We make use

[3] Code from the original paper is unavailable.

[4] Link to the code.

(n, m, h)	Initial Sort		Internal Sort per layer (total)		Compute Max Gini per layer (total)		Rest per layer (total)		Total	
	[18]	Ours	[18]	Ours	[18]	Ours	[18]	Ours	[18]	Ours
$(13, 11, 4)$	0	1.7	2.4(9.7)	0(0)	1.1(4.3)	0.1(0.4)	0.1(0.3)	0.4(1.5)	14.3	3.6
$(16, 40, 50)$	0	50.5	75.6(3781.1)	0(0)	35.3(1, 766.9)	1.7(83.5)	1.1(51.8)	10.0(497.4)	5599.8	631.4
$(19, 20, 50)$	0	200.3	321.7(16085.6)	0(0)	157.4(7868.9)	10.6(529.0)	5.7(285.0)	41.3(2064.1)	24, 239.5	2, 793.4

Table 4: Communication (GB) breakup across subprotocols for [18] and our training protocol for a dataset containing $N = 2^n$ training samples, m attributes to construct a decision tree of height h.

(n, m, h)	Runtime ([18])	Runtime (Ours)	IF
$(13, 11, 4)$	15.5	5.0	3.1
$(13, 20, 4)$	26.1	7.5	3.5
$(13, 11, 10)$	39.0	9.9	3.9
$(13, 11, 20)$	78.1	17.9	4.4
$(16, 11, 10)$	313.8	73.0	4.3
$(16, 20, 20)$	1, 036.7	186.6	5.6
$(16, 40, 50)$	5, 118.1	725.3	7.1
$(18, 20, 20)$	4, 627.4	826.0	5.6
$(18, 50, 50)$	26, 283.8	3, 496.3	7.5
$(18, 100, 100)$	105, 473.0	12, 009.5	8.8
$(19, 11, 10)$	2, 892.7	615.4	4.7
$(19, 20, 50)$	23, 461.4	3, 590.1	6.5
$(19, 40, 40)$	39, 394.0	5, 347.7	7.4

Table 5: Runtime (s) of Decision Tree Training for a dataset containing $N = 2^n$ training samples, m attributes to construct a decision tree of height h in LAN setup. IF represents the Improvement Factor i.e. Runtime([18]) / Runtime (Ours)

of (3, 2)-replicated secret-sharing based protocols over $\mathbb{Z}_{2^{128}}$ ring[5] and use the oblivious radix sort protocol from [8] to implement Π_{SortPerm}. In the following, we compare the communication and performance of secure training using [18] and our protocols for various values of (n, m, h), where $N = 2^n$ is the number of data points, m denotes the number of attributes, and h is the height of the tree that needs to be trained.

System Details. We evaluate on three Standard F16s v2 Azure instances in the LAN and WAN settings. Each server has 16 vCPUs and 64 GB RAM. Both the baseline and our code used 4 threads. Our network bandwidth and ping latency are 9.5 Gbps and 1.2 ms in the LAN setting, and 287 Mbps and 61 ms in the WAN setting, respectively.

6.1 Communication and Rounds

In this section, we compare the concrete communication cost of our training protocol with [18]. While our asymptotic improvement over [18] matches $min(h, m, \log N)$, a better approximation for the improvement factor based on the building block implementations in the MP-SPDZ library is

$$\frac{h(3712mnN + 99531mN + 16862N)}{h(12806mN + 2286Nn + 19568N) + 81331mN}$$

that can (potentially) help understand our empirical findings better. Next, we first compare the communication incurred in end-to-end training followed by a communication breakup for a few choices of (n, m, h). We use the communication breakup to highlight the bottlenecks in [18] and also discuss how our protocol design mitigates the same. Finally, we comment on the improvement in round complexity of our protocol over [18].

End-to-end communication. For various values of n, m and h, Table 3 compares the communication cost incurred by our protocol vs [18]. As can be observed from the table, our protocol is between $4 - 10\times$ more communication efficient than [18], with larger improvements on larger values of n, m and h.

Communication breakup. We present the breakup of communication cost across critical subcomputations/subprotocols for both [18] and our training protocol in Table 4. We pick three choices of parameters (n, m, h) to represent small, medium and large problem sizes. For all instances, for [18], it is clear that the bulk of the communication is incurred in sorting and computing max Gini index, and the rest of the cost is minimal. Our protocol addresses both of these bottlenecks. First, our protocol design avoids sorting per layer and only does sorting once. In doing this, it slightly increases the other cost, but this increase is much lower than sorting itself. Our initial sort is also cheaper than the per-layer sort used in [18] as explained in Section 4. Second, we compute the max Gini index much more efficiently by using a single call to Π_{GroupMax} compared to [18] that makes m calls to the same protocol.

Rounds. Our training protocol has roughly $1.5\times$ lower rounds compared to [18].

6.2 Performance

We compare the runtimes of training protocol in [18] and our work in both the LAN and WAN settings in Tables 5 and 6 respectively. We observe that our protocol is $3.1 - 9\times$ faster than [18] in the LAN setting. In the WAN setting, we only ran the smaller benchmarks (as [18] had an incredibly high run time exceeding nearly 13 hours); even then our improvements range between $2.9 - 7.3\times$.

In the LAN setting, our improvement over [18] is dominated by communication. In the WAN setting, where ping latency is higher and bandwidth is lower than LAN, overall runtime depends on both the rounds and the communication. Hence, we observe a slight difference in WAN improvements compared to LAN improvements.

[5] Even though the bit length of the cleartext attribute values is 32 bits, we secret share all values in the protocol over $\mathbb{Z}_{2^{128}}$. This is because we require multiplications (without truncating the values) to compute Gini indices without overflow (see Equation 1). Since MP-SPDZ does not allow different variables to be secret shared over different bit lengths, we are restricted to secret sharing all values over a large ring.

(n, m, h)	Runtime ([18])	Runtime (Ours)	IF
(13, 11, 4)	438.7	152.5	2.9
(13, 20, 4)	710.5	219.0	3.2
(13, 11, 10)	1,089.8	309.0	3.5
(13, 11, 20)	2,177.3	571.2	3.8
(16, 11, 10)	4,218.6	999.1	4.2
(16, 20, 20)	14,576.4	2,621.6	5.6
(16, 40, 50)	71,740.0	10,126.5	7.1
(18, 20, 20)	68,133.4	10,283.6	6.6
(19, 11, 10)	39,076.6	7,581.7	5.2

Table 6: Runtime (s) of Decision Tree Training for a dataset containing $N = 2^n$ training samples, m attributes to construct a decision tree of height h in WAN setup. IF represents the Improvement Factor i.e. Runtime([18]) / Runtime (Ours)

Recall that the round complexity of our protocol is lower than [18] (see Section 6.1).

6.3 Comparison with ENTS [17]

ENTS [17] protocol performs two optimizations over Hamada *et al.* [18]. First, they reduce the number of calls to the sorting protocol similar to our protocol. However, the asymptotic communication of their overall protocol remains $O(hmN \log N)$ as the cost of computing the minimum Gini index is $O(mN \log N)$. In contrast, our work reduces this cost to $O(mN + N \log N)$. Second, they secret share cleartext values over the small ring ($\mathbb{Z}_{2^{32}}$) and convert these values to shares over large rings ($\mathbb{Z}_{2^{128}}$) only when computing the Gini index. This technique is orthogonal and compatible with our work and can be used directly to further reduce our protocol communication. Without this optimization incorporated into our protocol, our protocol has ≈ 1.4× less communication than ENTS [17]. For example, on the Adult and Skin Segmentation datasets (see [17] for details on these), our protocol communicates 34.2GB and 68.3GB compared to the ENTS protocol that communicates 50.5GB and 88.2GB respectively.

7 Extensions

7.1 Extension to other threat models

Our secure training protocol, as presented in Section 4, makes use of existing sub-protocols for functionalities such as multiplication, bit decomposition, comparison, equality, sort, and shuffle, while minimizing the number of calls to the expensive functionalities. In this work, we instantiated these protocols concretely in the setting of 3-party seure computation tolerating 1 semi-honest corruption. However, our protocol can easily be extended to a larger number of parties with varying corruption thresholds as well as to the case of malicious security by instantiating the underlying functionalities with appropriate protocols.

7.2 Extension to Multi-Class Classification

Although this work focuses on binary classification, our training protocol can be extended to handle class labels from a larger domain $\mathbb{Z}_c, c > 2$. To do this, instead of a binary label y, we use $\mathbf{y}_1, \ldots, \mathbf{y}_c \in \{0,1\}^N$ where $(\mathbf{y}_1[i], \ldots, \mathbf{y}_c[i])$ is a one-hot encoding of the label of the i^{th} sample. The modified Gini index expression in this case will be: $\text{Gini}(\mathbf{D}) = 1 - \sum_{b \in c}(|\mathbf{D}_{y_b=1}|^2/|\mathbf{D}|^2)$. The subprotocol that needs to be altered is $\Pi_{\text{ComputeGini}}$ which would incur $O(cN)$ communication cost with this extension. The total communication cost of the training protocol would become $O(mN \log N + chmN + hN \log N)$.

7.3 Extension to more complex ML Models

In real world scenarios, a combination of random forests and gradient boosted decision (GBD) trees is used for training on datasets. A random forest is a collection of decision trees independently generated as a result of training on different subsets of the dataset. GBD trees are also a collection of decision trees, where each tree is built iteratively based on the decision trees previously generated, thereby correcting the errors associated with the previous fits. Our decision tree training protocol can be directly used in the training process of random forests and GBD trees. Consider the case of random forests, where we have q different subsets of the dataset on which we want to generate q independent decision trees. This involves q invocations of our training protocol, with different input attributes and labels. Similarly, GBD trees can be thought of as repeatedly training a single decision tree, with a modified metric to split features instead of Gini index, that takes into account the error correction factor.

References

[1] 2019. MP-SPDZ: Versatile framework for multi-party computation. https://github.com/data61/MP-SPDZ.

[2] 2023. Decision Tree. https://github.com/data61/MP-SPDZ/blob/master/Compiler/decision_tree.py.

[3] Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. 2021. Secure training of decision trees with continuous attributes. In *PoPETS*.

[4] Adi Akavia, Max Leibovich, Yehezkel S. Resheff, Roey Ron, Moni Shahar, and Margarita Vald. 2022. Privacy-Preserving Decision Trees Training and Prediction. In *ECML-PKDD*.

[5] Rasoul Akhavan Mahdavi, Haoyan Ni, Dimitry Linkov, and Florian Kerschbaum. 2023. Level Up: Private Non-Interactive Decision Tree Evaluation using Levelled Homomorphic Encryption. In *CCS*.

[6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-throughput semi-honest secure three-party computation with an honest majority. In *CCS*.

[7] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *CCS*.

[8] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *CCS*.

[9] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *STOC*.

[10] Dan Bogdanov, Sven Laur, and Riivo Talviste. 2014. A Practical Analysis of Oblivious Sorting Algorithms for Secure Multi-party Computation. In *NordSec*.

[11] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. 2007. Privacy-preserving remote diagnostics. In *CCS*.

[12] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*.

[13] Nan Cheng, Naman Gupta, Aikaterini Mitrokotsa, Hiraku Morita, and Kazunari Tozawa. 2024. Constant-Round Private Decision Tree Evaluation for Secret Shared Data. In *PoPETS*.

[14] Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. 2014. Practical Secure Decision Tree Learning in a Teletreatment Application. In *FCDS*.

[15] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *STOC*.

[16] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*.

[17] Wenqiang Ruan Ruisheng Zhou Lushan Song Bingshuai Li Yunfeng Shao Guopeng Lin, Weili Han. 2024. Ents: An Efficient Three-party Training Framework for Decision Trees by Communication Optimization. In *CCS*.

[18] Koki Hamada, Dai Ikarashi, Ryo Kikuchi, and Koji Chida. 2023. Efficient decision tree training with new data structure for secure multi-party computation. In *PoPETS*.

[19] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2013. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*.

[20] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. In *Electronics and Communications in Japan*.

[21] Keyu Ji, Bingsheng Zhang, Tianpei Lu, Lichun Li, and Kui Ren. 2023. UC Secure Private Branching Program and Decision Tree Evaluation. In *IEEE DSC*.

[22] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS*.

[23] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-Efficient Oblivious Database Manipulation. In *ISC*.

[24] Yehuda Lindell. 2017. How to Simulate It – A Tutorial on the Simulation Proof Technique. In *Tutorials on the Foundations of Cryptography*.

[25] Yehuda Lindell and Ariel Nof. 2017. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *CCS*.

[26] Yehuda Lindell and Benny Pinkas. 2000. Privacy Preserving Data Mining. In *CRYPTO*.

[27] Wen-jie Lu, Zhicong Huang, Qizhi Zhang, Yuchen Wang, and Cheng Hong. 2023. Squirrel: a scalable secure two-party computation framework for training gradient boosting decision tree. In *USENIX*.

[28] Jack Ma, Raymond Tai, Yongjun Zhao, and Sherman Chow. 2020. Let's Stride Blindfolded in a Forest: Sublinear Multi-Client Decision Trees Evaluation. In *NDSS*.

[29] Qingkai Ma and Ping Deng. 2008. Secure Multi-party Protocols for Privacy Preserving Data Mining. In *WASA*.

[30] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *CCS*.

[31] Saeed Samet and Ali Miri. 2008. Privacy preserving ID3 using Gini Index over horizontally partitioned data. In *AICCSA*.

[32] Adi Shamir. 1979. How to Share a Secret. In *Communication of ACM*.

[33] Raymond Tai, Jack Ma, Yongjun Zhao, and Sherman Chow. 2017. Privacy-Preserving Decision Trees Evaluation via Linear Functions. In *ESORICS*.

[34] Hikaru Tsuchida, Takashi Nishide, and Yusaku Maeda. 2020. Private Decision Tree Evaluation with Constant Rounds via (Only) SS-3PC over Ring. In *ProvSec*.

[35] Jaideep Vaidya, Chris Clifton, Murat Kantarcioglu, and A. Scott Patterson. 2005. Privacy-preserving decision trees over vertically partitioned data. In *DBSec*.

[36] Ke Wang, Yabo Xu, Rong She, and Philip S. Yu. 2006. Classification Spanning Private Databases. In *AAAI*.

[37] David Wu, Tony Feng, Michael Naehrig, and Kristin Lauter. 2016. Privately Evaluating Decision Trees and Random Forests. In *PoPETS*.

[38] Ming-Jun Xiao, Liu-Sheng Huang, Yong-Long Luo, and Hong Shen. 2005. Privacy Preserving ID3 Algorithm over Horizontally Partitioned Data. In *PDCAT*.

[39] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *FOCS*.

A Plaintext Decision Tree Training

The algorithm for cleartext decision tree training is presented in Algorithm 14.

Protocol 14: Plaintext decision tree training

```
Input: Labelled training dataset $\mathbf{D}$, maximum height $h$.
Output: Decision tree $\mathcal{T}$
if $h = 0$ then
    Let $v$ be a leaf node.
    Label($v$) is the most occuring label in $\mathbf{D}$. Output tree with $v$ as root node.
else
    Find $j, t$ such that $x_j < t$ minimizes Gini index of the split.
    Let $\mathbf{D}_l = \mathbf{D}_{x_j < t}$ and $\mathbf{D}_r = \mathbf{D}_{x_j \geq t}$ be partition of $\mathbf{D}$ based on test.
    Compute $\mathcal{T}_l = \mathsf{Train}(\mathbf{D}_l, h-1)$ and $\mathcal{T}_r = \mathsf{Train}(\mathbf{D}_r, h-1)$.
    Let $v$ be an internal node with test $x_j < t$ and with left child $\mathcal{T}_l$, right child $\mathcal{T}_r$.
    Output tree with $v$ as root node.
end
```

B Training protocol for discrete attributes

Our protocol can be easily extended to handle discrete attributes as well. Using ideas of [14], we compute the Gini index for discrete attributes. Suppose dataset $\mathbf{D}$ has attribute values for a binary attribute $d \in \{0, 1\}$. Then the Gini index of splitting from test $d \overset{?}{=} 0$ (or $d \overset{?}{=} 1$) is defined as

$$G_d = \frac{|\mathbf{D}_{d=0}|}{|\mathbf{D}|} \operatorname{Gini}(\mathbf{D}_{d=0}) + \frac{|\mathbf{D}_{d=1}|}{|\mathbf{D}|} \operatorname{Gini}(\mathbf{D}_{d=1})$$

where $\operatorname{Gini}(\mathbf{D}) = 1 - \sum_{b \in \{0,1\}} (|\mathbf{D}_{y=b}|^2 / |\mathbf{D}|^2)$. Minimizing G_d is same as maximizing G'_d:

$$\mathrm{G}'_d(\mathbf{D}) = \Big(|\mathbf{D}_{d=1}| \left(\left|\mathbf{D}_{d=0 \wedge y=0}\right|^2 + \left|\mathbf{D}_{d=0 \wedge y=1}\right|^2 \right) + |\mathbf{D}_{d=0}| \left(\left|\mathbf{D}_{d=1 \wedge y=0}\right|^2 + \left|\mathbf{D}_{d=1 \wedge y=1}\right|^2 \right) \Big) / (|\mathbf{D}_{d=0}| \, |\mathbf{D}_{d=1}|)$$

where $\mathbf{D}_{d=b' \wedge y=b} = \{(\mathbf{d}, y) \in \mathbf{D} \mid d = b' \wedge y = b\}$.

This expression is similar to that of the Gini index of continuous variables and can be computed securely by simply modifying $\Pi_{\mathsf{ComputeGini}}$. Once we have the Gini index for both continuous and discrete attributes, simply choose the splitting test, which has the maximum Gini index, and the rest of the training protocol proceeds as described earlier.

If the attribute is discrete but non-binary i.e. domain$(d) = \mathbb{Z}_k$, then each value $i \in$ domain(d) induces a partition of dataset as $\{\mathbf{D}_{d=i}, \mathbf{D}_{d \neq i}\}$. We can then proceed to compute the Gini index of all such possible splits using the expression above and select the split associated with the minimum Gini index.

C End-to-end secure training protocol

Protocol 15: Π_{Train}

```
Input: dataset $\langle \mathbf{D} \rangle$, height $h$
Output: $\left\{ \left\langle \mathsf{Layer}^{(i)} \right\rangle \right\}_{i \in [h+1]}$
Cost: $O(mN \log N + h(mN + N \log N))$
$\langle \mathbf{D}_1 \rangle = \langle \mathbf{D} \rangle$
$\langle \mathbf{g}_1[1] \rangle = 1$ and $\langle \mathbf{g}_1[i] \rangle = 0$ for $i \in [2, N]$
$\langle \mathbf{NID}[i] \rangle = 1$ for all $i \in [N]$
$\left\{ \left\langle \mathbf{P}_1^i \right\rangle \right\} = \Pi_{\mathsf{SetupPerm}} \left( \langle \mathbf{D}_1 \rangle \right)$
$\left\langle \mathsf{State}^{(1)} \right\rangle = \left( \langle \mathbf{D}_1 \rangle, \langle \mathbf{g}_1 \rangle, \left\{ \left\langle \mathbf{P}_1^i \right\rangle \right\}_{i \in [m]}, \langle \mathbf{NID} \rangle \right)$
for $k = 1$ to $h$ do
    $\left\langle \mathsf{Layer}^{(k)} \right\rangle, \left\langle \mathsf{State}^{(k+1)} \right\rangle = \Pi_{\mathsf{TrainInternalLayer}} \left( \left\langle \mathsf{State}^{(k)} \right\rangle, k \right)$
end
$\left\langle \mathsf{Layer}^{(h+1)} \right\rangle = \Pi_{\mathsf{TrainLeafLayer}} \left( \left\langle \mathsf{State}^{(h+1)} \right\rangle \right)$
Output $\left\{ \left\langle \mathsf{Layer}^{(i)} \right\rangle \right\}_{i \in [h+1]}$.
```

D Protocols for Element wise operations

The protocols realizing element wise operations are as follows:

- **Secure Multiplication:** denoted by $\langle z \rangle = \Pi_{\mathsf{MULT}}(\langle u \rangle, \langle v \rangle)$. We use techniques from [6]. Parties have shares $\langle u \rangle, \langle v \rangle \in \mathbb{Z}_{2^\ell}$ and want to compute shares $\langle z \rangle \in \mathbb{Z}_{2^\ell}$ where $z = u \cdot v$. Then
$$\begin{aligned} z &= (u_0 + u_1 + u_2) \cdot (v_0 + v_1 + v_2) \\ &= (u_0 v_0 + u_0 v_1 + u_1 v_0) + (u_1 v_1 + u_1 v_2 + u_2 v_1) \\ &\quad + (u_2 v_2 + u_2 v_0 + u_0 v_2) \\ &= z_0 + z_1 + z_2 \end{aligned}$$
where party S_i can compute z_i locally. Parties mask this z_i using correlated randomness generated non-interactively (party S_i holds share α_i such that $\alpha_0 + \alpha_1 + \alpha_2 = 0$ which is added to z_i). Each party can obtain replicated shares of z with communication of 3ℓ bits (each party sends its masked z_i to the next party). An extension of the Secure Multiplication protocol to the malicious adversary setting is provided in [25].
- **Secure Bit Decomposition:** denoted by $\langle u \rangle^{\mathbf{B}} = \Pi_{\mathsf{A2B}}(\langle u \rangle)$. Parties have RSS $\langle u \rangle \in \mathbb{Z}_{2^\ell}$ and want to compute shares of bit decomposition $\langle u \rangle^{\mathbf{B}} = \left(\langle u_{\ell-1} \rangle^{\mathbf{B}}, \ldots, \langle u_0 \rangle^{\mathbf{B}} \right)$ where $u = \sum_{i=0}^{\ell-1} 2^i u_i$. We use the method described in [30] which uses $\ell \log \ell$ AND gates requiring $1 + \log \ell$ rounds. The concrete cost of bit decomposition is $\approx 212\ell$ bits.
- **Secure Comparison:** denoted by $\langle b \rangle = \Pi_{\mathsf{LT}}(\langle u \rangle, \langle v \rangle)$. Parties have RSS $\langle u \rangle, \langle v \rangle \in \mathbb{Z}_{2^\ell}$ and want to compute $\langle b \rangle \in \mathbb{Z}_{2^\ell}$ where $b = (u < v)$. Parties compute $\langle z \rangle = \langle u - v \rangle$ locally and compute $b = (z < 0)$ by extracting most significant bit of z using techniques from [30] with 2ℓ AND gates in $\log \ell$ rounds. Note that $\mathsf{MSB}(z) = 1$ if $z < 0$ and $\mathsf{MSB}(z) = 0$ otherwise. The concrete cost of secure comparison using $(3, 2)$ RSS is $\approx 15\ell$ bits.
- **Secure Equality:** denoted by $\langle b \rangle = \Pi_{\mathsf{EQ}}(\langle u \rangle, \langle v \rangle)$. Parties have RSS $\langle u \rangle, \langle v \rangle \in \mathbb{Z}_{2^\ell}$ and want to compute $\langle b \rangle \in \mathbb{Z}_{2^\ell}$ where $b = \left(u \overset{?}{=} v \right) = \left((u - v) \overset{?}{=} 0 \right)$. Parties compute $\langle z \rangle = \langle u - v \rangle$ locally, compute its bit decomposition ([30]) and then check if z is zero using a binary circuit. The concrete cost of secure equality using $(3, 2)$ RSS is $\approx 20\ell$ bits.

E Protocols for oblivious permutations

As discussed in Section 3.2, we present the protocols to – a) apply a secret permutation to a secret vector (Protocol 16); b) compose secret permutations (Protocol 17); c) compute stable sorting permutation for binary key (Protocol 18); and d) compute stable sorting permutation for arbitrary key (Protocol 19).

F Correctness Lemma

In $\Pi_{\text{UpdateState}}$, parties compute the shares of the updated database $\mathbf{D}_{k+1}$ by stably sorting $\mathbf{D}_k$ based on $\mathbf{b}$. We prove that $\mathbf{D}_{k+1}$ has correctly grouped elements.

Lemma F.1. *Entries of the same group appear consecutively in $\mathbf{D}_{k+1}$ where $\mathbf{D}_{k+1}$ is obtained by sorting $\mathbf{D}_k$ based on values of $\mathbf{b}$.*

Proof. We prove by contradiction. Suppose the theorem statement is not true. This means that there exist $d, e, f \in [N]$ such that $d < e < f$ and $\mathbf{D}_{k+1}[d]$, $\mathbf{D}_{k+1}[f]$ belong to the same group (call it (x, y)) and $\mathbf{D}_{k+1}[e]$ belongs to a different group (call it (x', y')). This implies that either

(1) $y \neq y'$: This is a contradiction as $\mathbf{D}_{k+1}$ is obtained by sorting based on y so for all $e \in [d, f]$, $y' = y$.

Protocol 16: $\Pi_{\text{ApplyPerm}}$ [8]

```
Input: Vector $\langle \mathbf{X} \rangle$, permutation $\langle \mathbf{P} \rangle$ of length $N$.
Output: Vector $\langle \mathbf{X}^{\mathbf{P}} \rangle$.
Parties compute $\langle \mathbf{X}^{\pi} \rangle, \mathbf{P}^{\pi} = \text{Shuffle}(\langle \mathbf{X} \rangle, \langle \mathbf{P} \rangle)$.
for $i \in [N]$ do
    $\langle \mathbf{X}^{\mathbf{P}}[\mathbf{P}^{\pi}[i]] \rangle = \langle \mathbf{X}^{\pi}[i] \rangle$
end
Output $\langle \mathbf{X}^{\mathbf{P}} \rangle$
```

Protocol 17: Π_{PermComp} [8]

```
Input: Permutations $\langle \mathbf{P} \rangle$ and $\langle \mathbf{Q} \rangle$ of length $N$.
Output: Permutation $\langle \mathbf{G} \rangle$ where $\mathbf{G} = \mathbf{P} \circ \mathbf{Q}$.
Parties compute $\langle \mathbf{Q}^{\pi} \rangle = \text{Shuffle}(\langle \mathbf{Q} \rangle)$ and reconstruct $\mathbf{Q}^{\pi}$.
Let $\langle \mathbf{G}^{\pi}[i] \rangle = \langle \mathbf{P}[\mathbf{Q}^{\pi}[i]] \rangle$ for all $i \in [N]$.
Parties compute $\langle \mathbf{G} \rangle = \text{Unshuffle}(\langle \mathbf{G}^{\pi} \rangle)$.
Parties output $\langle \mathbf{G} \rangle$.
```

Protocol 18: $\Pi_{\text{SortPermBit}}$ [8]

```
Input: Vector $\langle \mathbf{b} \rangle \in \{0, 1\}^N$.
Output: Permutation $\langle \mathbf{P} \rangle$ that stably sorts $\mathbf{b}$
$\langle nz \rangle = \text{Sum}(\neg \langle \mathbf{b} \rangle)$
$\langle \mathbf{r} \rangle = \text{PrefixSum}(\langle \mathbf{b} \rangle) - \langle \mathbf{b} \rangle$            //Local compute
for $i \in \{1, \ldots, N\}$ do
    $\langle \mathbf{P}[i] \rangle = i - (i-1) \cdot \langle \mathbf{b}[i] \rangle - \langle \mathbf{r}[i] \rangle + \langle \mathbf{b}[i] \rangle \cdot \langle nz + 2\mathbf{r}[i] \rangle$
end
Output $\langle \mathbf{P} \rangle$.
```

Protocol 19: Π_{SortPerm} [8]

```
Input: Vector $\langle \mathbf{x} \rangle \in \mathbb{Z}_{2^\ell}^N$.
Output: Permutation $\langle \mathbf{P} \rangle$ that stably sorts $\mathbf{x}$
for $i = 1$ to $N$ do
    Let $\langle x_i \rangle = \langle \mathbf{x}[i] \rangle$
    $(\langle x_i[\ell-1] \rangle^{\mathbf{B}}, \ldots, \langle x_i[0] \rangle^{\mathbf{B}}) = \Pi_{\text{A2B}}(\langle x_i \rangle)$
end
Let $\langle \mathbf{P} \rangle = \langle [1, \ldots, N] \rangle$.
for $j = 0$ to $\ell - 1$ do
    Let $\langle \mathbf{y}_j \rangle = [\langle x_1[j] \rangle^{\mathbf{B}}, \ldots, \langle x_N[j] \rangle^{\mathbf{B}}]$
    $\langle \mathbf{y}_j^{\mathbf{P}} \rangle = \Pi_{\text{ApplyPerm}}(\langle \mathbf{y}_j \rangle, \langle \mathbf{P} \rangle)$
    $\langle \sigma \rangle = \Pi_{\text{SortPermBit}}(\langle \mathbf{y}_j^{\mathbf{P}} \rangle)$
    $\langle \mathbf{P} \rangle = \Pi_{\text{PermComp}}(\langle \sigma \rangle, \langle \mathbf{P} \rangle)$
end
Output $\langle \mathbf{P} \rangle$.
```

(2) $x' \neq x$: We already know that $y' = y$. Suppose we have d', e', f' such that

$$\mathbf{D}_k[d'] = \mathbf{D}_{k+1}[d]$$
$$\mathbf{D}_k[e'] = \mathbf{D}_{k+1}[e]$$
$$\mathbf{D}_k[f'] = \mathbf{D}_{k+1}[f]$$

Then $d' < e' < f'$ (stable sorting) and for all $e' \in [d', f']$, $x' = x$ (all entries of a group appear consecutively in $\mathbf{D}_k$). This is also a contradiction.

Hence, all entries in the same group appear consecutively in $\mathbf{D}_{k+1}$. □

Lemma 5.3. *Protocol* $\Pi_{\mathsf{UpdateState}}$ *defined in Algorithm 10 correctly computes the* UpdateState *functionality from Section 4.4.4.*

Proof. In the UpdateState protocol, parties update the shares of dataset, group flag and node id vectors. Parties also update the permutations that sort the updated dataset according to each attribute within groups. To prove correctness of $\Pi_{\mathsf{UpdateState}}$, we show that these updates are computed correctly and updated vectors satisfy the invariant described in Section 4.4.

Correctness of $\mathbf{D}_{k+1}$. Suppose $\mathbf{D}_k$ has n_k groups and satisfies the invariant. Each group will be further subdivided into 2 new groups based on output of comparison i.e. $\mathbf{b} = 0$ or 1. We can denote the new groups of samples as a tuple (x, y), $x \in [1, n_k]$ and $y \in \{0, 1\}$ where x is the sample's group in $\mathbf{D}_k$ and y is the output of comparison.

To obtain $\mathbf{D}_{k+1}$, $\Pi_{\mathsf{UpdateState}}$ stably sorts $\mathbf{D}_k$ based on values of $\mathbf{b}$. The order in which groups appear in $\mathbf{D}_k$ is $1, 2, \ldots, n_k$ i.e. in $\mathbf{D}_k$ we first have elements of group 1 followed by group 2 and so on. The order in which groups appear in $\mathbf{D}_{k+1}$ is $(1, 0), \ldots, (n_k, 0)$, $(1, 1), \ldots, (n_k, 1)$. The ordering of groups is irrelevant as long as entries of the same group appear consecutively in $\mathbf{D}_{k+1}$. However, from Lemma F.1, we know that sorting $\mathbf{D}_k$ according to $\mathbf{b}$ indeed gives us $\mathbf{D}_{k+1}$ with correctly grouped entries.

Correctness of $\mathbf{g}_{k+1}$. We compute $\mathbf{g}$ from $\mathbf{g}_k$ in step 5 in $\Pi_{\mathsf{UpdateState}}$ using $\Pi_{\mathsf{GroupFirstOne}}$ protocol. From the description of $\Pi_{\mathsf{GroupFirstOne}}$ (Section 3.3), we can see that $\mathbf{g}[i] = 1$ if $\mathbf{b}[i]$ is the first instance of a 0 or 1 in any group. This means $\mathbf{g}[i] = 1$ if i is the first element of group $(j, 0)$ or $(j, 1)$ for any $j \in [1, n_k]$.

Finally $\Pi_{\mathsf{UpdateState}}$ sorts $\mathbf{g}$ according to $\mathbf{b}$ to obtain $\mathbf{g}_{k+1}$ in step 6. As we are stably sorting, the first element of any new group in $\mathbf{D}_k$ will be the first element of the group in $\mathbf{D}_{k+1}$. Hence, $\mathbf{g}_{k+1}[i] = 1$ if i is the first element of a group in $\mathbf{D}_{k+1}$.

Correctness of NID. Suppose we have a node with $NID = d$. Then the left child will be numbered d and the right child will be numbered $d + 2^{k-1}$. This is computed in step 3 in $\Pi_{\mathsf{UpdateState}}$ as $\mathbf{NID} = \mathbf{NID} + 2^{k-1} \cdot \mathbf{b}$. This numbering is consistent with the ordering of groups in $\mathbf{D}_{k+1}$ as the left children of all the nodes are numbered followed by all the right children.

Finally we apply permutation $\mathbf{Q}$ in step 4 to keep the ordering of $\mathbf{NID}$ and $\mathbf{D}_{k+1}$ same. Same ordering is necessary so that $\mathbf{NID}[i]$ corresponds to the node id of sample $\mathbf{D}_{k+1}[i]$. Hence, the updated vector $\mathbf{NID}$ satisfies the invariant.

Correctness of sorting permutations $\mathbf{P}^i_{k+1}$. We introduce a few more notations as follows:

(1) $\mathbf{D}_k$ is the dataset grouped by nodes in k^{th} layer. $\mathbf{P}^i_k$ is the permutation that sorts $\mathbf{D}_k$ according to i^{th} attribute. Let $\mathbf{D}^i_k = \text{Permute}\left(\mathbf{D}_k, \mathbf{P}^i_k\right)$.
(2) $\mathbf{D}_{k+1}$ is the new dataset grouped by nodes in $(k+1)^{th}$ layer. $\mathbf{P}^i_{k+1}$ is the new permutation that sorts $\mathbf{D}_{k+1}$ according to i^{th} attribute. Let $\mathbf{D}^i_{k+1} = \text{Permute}\left(\mathbf{D}_{k+1}, \mathbf{P}^i_{k+1}\right)$.

Suppose we are given $\mathbf{P}^i_k$, $\mathbf{D}_{k+1}$ and have to compute $\mathbf{P}^i_{k+1}$. From the definition of $\mathbf{D}^i_k, \mathbf{D}^i_{k+1}$, we have

$$\mathbf{D}^i_k\left[\mathbf{P}^i_k[j]\right] = \mathbf{D}_k[j] \quad \forall i \in [m], j \in [N] \tag{2}$$
$$\mathbf{D}^i_{k+1}\left[\mathbf{P}^i_{k+1}[j]\right] = \mathbf{D}_{k+1}[j] \quad \forall i \in [m], j \in [N] \tag{3}$$

Just like how we obtain $\mathbf{D}_{k+1}$ from $\mathbf{D}_k$ by sorting it based on $\mathbf{b}$, we can obtain $\mathbf{D}^i_{k+1}$ by sorting $\mathbf{D}^i_k$ based on $\mathbf{b}^i$ (can be proved similar to lemma F.1) where $\mathbf{b}^i = \text{Permute}(\mathbf{b}, \mathbf{P}^i_k)$. Let $\mathbf{R}^i$ be the permutation that sorts $\mathbf{b}^i$. Then, we have

$$\mathbf{b}^i\left[\mathbf{P}^i_k[j]\right] = \mathbf{b}[j] \quad \forall i \in [m], j \in [N] \tag{4}$$
$$\mathbf{D}^i_{k+1}\left[\mathbf{R}^i[j]\right] = \mathbf{D}^i_k[j] \quad \forall i \in [m], j \in [N] \tag{5}$$

Let $\mathbf{Q}$ be the permutation that sorts $\mathbf{b}$. Then we have

$$\mathbf{D}_{k+1}[\mathbf{Q}[j]] = \mathbf{D}_k[j] \quad \forall j \in [N] \tag{6}$$

Let $\mathbf{Q}^i$ be the permutation obtained by applying $\mathbf{Q}$ to $\mathbf{P}^i_k$ i.e.

$$\mathbf{Q}^i[\mathbf{Q}[j]] = \mathbf{P}^i_k[j] \quad \forall i \in [m], j \in [N] \tag{7}$$

Using these equations, we have

$$\begin{aligned} & \mathbf{D}^i_{k+1}\left[\mathbf{R}^i[j]\right] = \mathbf{D}^i_k[j] && \text{from (5)} \\ \implies & \mathbf{D}^i_{k+1}\left[\mathbf{R}^i\left[\mathbf{P}^i_k[j]\right]\right] = \mathbf{D}^i_k\left[\mathbf{P}^i_k[j]\right] \\ \implies & \mathbf{D}^i_{k+1}\left[\mathbf{R}^i\left[\mathbf{P}^i_k[j]\right]\right] = \mathbf{D}_k[j] && \text{from (2)} \\ \implies & \mathbf{D}^i_{k+1}\left[\mathbf{R}^i\left[\mathbf{Q}^i[\mathbf{Q}[j]]\right]\right] = \mathbf{D}_{k+1}[\mathbf{Q}[j]] && \text{from (6, 7)} \\ \implies & \mathbf{D}^i_{k+1}\left[\mathbf{R}^i\left[\mathbf{Q}^i[j]\right]\right] = \mathbf{D}_{k+1}[j] \end{aligned}$$

Therefore, $\mathbf{P}^i_{k+1} = \mathbf{R}^i \circ \mathbf{Q}^i$ where $\mathbf{R}^i$ is the permutation that sorts $\mathbf{b}^i$ and $\mathbf{Q}^i$ is the permutation obtained by applying $\mathbf{Q}$ to $\mathbf{P}^i_k$. $\Pi_{\mathsf{UpdateState}}$ computes $\mathbf{R}^i$ and $\mathbf{Q}^i$ using $\Pi_{\mathsf{ApplyPerm}}$ in steps 9, 10 and $\mathbf{P}^i_{k+1}$ using Π_{PermComp} in step 11.

This concludes our correctness proof as all components of updated $\text{State}^{(k+1)}$ satisfy the invariant. □