Securely Training Decision Trees Efficiently

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

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

Sandhya Saravanan  
Microsoft Research  
Bengaluru, India  
t-ssaravanan@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 10x less communication and is 9x faster than [18].

CCS Concepts

- Security and privacy → Cryptography: Privacy-preserving protocols.

Keywords

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

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 andPinkas [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 Abspool [3] showed how to build a concretely efficient MPC protocol. This protocol required a total communication 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 $O(hmN \log N)$.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CCS ’24, October 14–18, 2024, Salt Lake City, UT, USA © 2024 Copyright held by the owner/author(s).
ACM ISBN 978-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670268
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 \(O(mN \log N + hN + hN \log N)\) 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 our protocol and show that our protocol is concretely 10x more communication frugal and 9x 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 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) partitioned datasets 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-
We consider 3 parties $S$ as well as its applicability in training scenarios such as random forests.

One party, there exists an ideal world PPT simulation of set $A$ for $S$. For security, we require that the view of adversary $A$ is indistinguishable from the view and output of a simulator in an ideal world which has access to the ideal functionality $F$ and the outputs of honest parties in the real world. For any information. Suppose $S$ is a ring of integers modulo $\ell$ for $\ell \in \mathbb{N}$. Subset $B$ of set $A$ is denoted by $B \subseteq A$. We use $n$ to denote $\log_2 N$.

### 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 group wise 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 $2^\ell$ 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 $A$ 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 $\Pi$ is the 3-party protocol that computes the functionality $F$. $\mathcal{A}$ 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 $F$. $F$ 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 $\Pi$ such that for any input inp to the protocol $\Pi$ that computes $F$, we have $\Pi(\Pi, \mathcal{A}, \text{inp}) \approx_c \Pi(\Pi, \mathcal{A}, \text{Sim}, \text{inp})$ where $\Pi$ is the joint distribution of the view of adversary $\mathcal{A}$ and output of honest parties in the real world, and $\Pi$ is the joint distribution of the view of the simulator $\Pi$ 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 $D$ with $m$ attributes $x = (x_1, \ldots, x_m)$ and label $y$. Here, $D(\bar{i}) = (x_i[1], \ldots, x_i[m], y[i])$, 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 $x$.

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 $x$ 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 $D$ using test $x_j < t$ is defined as:

$$G_{x_j < t}(D) = \frac{|D_{x_j < t}|}{|D|} \text{Gini}(D_{x_j < t}) + \frac{|D_{x_j \geq t}|}{|D|} \text{Gini}(D_{x_j \geq t})$$

where $\text{Gini}(D) = 1 - \sum_{k \in \{0,1\}} (|D_{y=k}|^2/|D|)^2$. The best split has the minimum Gini index and Abspeel et al. [3] showed that minimizing
the Gini index is same as maximizing the following expression:
\[
G'_{X < t}(D) = \left[ |D_{X < t}| \left( |D_{X < t \land y = 0}|^2 + |D_{X < t \land y = 1}|^2 \right) \right] + |D_{X < t}| \left( |D_{X \geq t \land y = 0}|^2 + |D_{X \geq t \land y = 1}|^2 \right) / (|D_{X < t}| |D_{X \geq t}|) ,
\]

where \(D_{X < t \land y = b} = \{ (x, y) \in D \mid x < t \land y = b \} \). The algorithm selects the test \(x_{j} < t\) that maximizes this expression.

Once the test is selected, the algorithm partitions the training dataset into \(D_{X < t}\) and \(D_{X \geq t}\). The left subtree \(T_{l}\) is trained on the partition \(D_{X < t}\) and right subtree \(T_{r}\) is trained on the partition \(D_{X \geq t}\). The algorithm outputs a decision tree with the current node as root node, \(T_{l}\) as left child and \(T_{r}\) as right child.

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

\[
\text{Label} = \sum_{i \in D_{k}} y[i] \cdot \sum_{i \in D_{k}} (1 - y[i])
\]

**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 recon-struct 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 \(v_0 + v_1 + v_2 = 0 \pmod{2^{\ell}}\) and output the shares \((x_0, x_1, x_2)\) where \(x_0 = (v_0, v_1), x_1 = (v_1, v_2), x_2 = (v_2, x_0)\).

- **Reconstruct:** On input \(x_i = (u_0, u_1), x_j = (v_1, v_2) \in \mathbb{Z}_{2^{\ell}}\) where \(i \neq j, i, j \in \{1, 2, 3\}\), output \(v' = (u_0 + v_1 + v_2) \pmod{2^{\ell}}\).

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 \(\langle v \rangle^{B}\). The share of party \(S_i\) is \((u_i, u_{i+1})\).

### 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 \(\langle z \rangle = \Pi_{\text{MULT}}(\langle u \rangle, \langle v \rangle)\).
- **Secure Bit Decomposition:** denoted by \(\langle b \rangle = \Pi_{\text{BD}}(\langle u \rangle)\).
- **Secure Comparison:** denoted by \(\langle b \rangle = \Pi_{\text{TC}}(\langle u \rangle, \langle v \rangle)\).
- **Secure Equality:** denoted by \(\langle b \rangle = \Pi_{\text{EQ}}(\langle u \rangle, \langle v \rangle)\).

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 \(P\) where \(P[i]\) denotes the position of the \(i^{th}\) element in the permuted order. Let \(A \in \mathbb{Z}_{2^{3\ell}}^N\) be a vector and \(P\) be a permutation. Let \(A^P\) be the rearrangement of \(A\) according to \(P\), i.e., \(A^P = \text{Permute}(A, P)\). Since \(P[i]\) denotes the new position of the \(i^{th}\) element, we have \(A^P[\Pi[i]] = A[i]\) for all \(i \in [N]\).

The secret share of a permutation vector is defined as \(\langle P \rangle = (\langle P[1] \rangle, \ldots, \langle P[N] \rangle)\).

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 \(X\) in \(\mathbb{Z}_{2^{\ell}}^N\) and a pair of permutations \((\pi_i, \pi_{i+1})\) \(\pmod{3}\). The functionality permutes \(X\) according to \(\pi = \pi_2 \circ \pi_1 \circ \pi_0\) and outputs replicated secret shares of the permuted vector \(Y\) to parties. Protocol \(\Pi_{\text{shuffle}}\) 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 \(\pi^{-1}\) by applying the permutations in reverse order i.e. apply \(\pi_2^{-1}\), then \(\pi_1^{-1}\) and finally \(\pi_0^{-1}\) as \(\pi^{-1} = \pi_2^{-1} \circ \pi_1^{-1} \circ \pi_0^{-1}\). A typical use of shuffling is to hide data dependent memory accesses. To do this, parties shuffle a vector using \(\pi\), perform the memory accesses and then shuffle by \(\pi^{-1}\) to get back the same ordering.

**Input:** Party \(S_i, i \in \{0, 1, 2\}\) inputs \(\langle X \rangle\) and \((\pi_i, \pi_{i+1})\).

**Output:** Parties receive \(Y\) where \(Y\) is a rearrangement of \(X\) according to \(\pi = \pi_2 \circ \pi_1 \circ \pi_0\).

1. Reconstruct \(X\) and \(\pi = \pi_2 \circ \pi_1 \circ \pi_0\).
2. Compute \(Y = \text{Permute}(X, \pi)\).
3. Output \(Y\).

**Apply permutation.** Functionality 2 takes shares of a vector \(X\) and permutation \(P\) and outputs shares of \(X^P\) where \(X^P = \text{Permute}(X, P)\).
Protocol \( \Pi_{\text{ApplyPerm}} \) (Algorithm 16) in [8] realizes ApplyPerm with \( 8Nt \) bits of communication in 2 rounds.

### Functionality 2: \( \text{ApplyPerm} \)

**Input:** Parties input \( (X) \) and \( (P) \)

**Output:** Parties receive \( \langle X^P \rangle \) where \( X^P = \text{Permute}(X, P) \).

1. Reconstruct \( X \) and \( P \).
2. Compute \( X^P \) where \( X^P = \text{Permute}(X, P) \).
3. Output \( \langle X^P \rangle \).

### Composing permutations

Composition of 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 \( kt \times 8Nt \) bits using \( \text{ApplyPerm} \). Instead we use \( \Pi_{\text{PermComp}} \) to compute the final permutation to be applied to vectors and apply this permutation to all \( t \) vectors which costs \( (k + t) \times 8Nt \) bits.

### Functionality 3: \( \text{PermComp} \)

**Input:** Parties input \( (P) \) and \( (Q) \)

**Output:** Parties receive \( \langle R \rangle \) where \( R = P \circ Q \).

1. Reconstruct \( P \) and \( Q \).
2. Compute \( R \) where \( R[i] = P[Q[i]] \) \( \forall i \in [N] \).
3. Output \( \langle R \rangle \).

### Sorting permutation for binary key

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

**Definition 3.1.** Permutation \( P \) stably sorts a list \( b \) if \( \forall i,j \in [N] \)

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

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

### Functionality 4: \( \text{SortPermBit} \)

**Input:** Parties input \( (b) \) where \( b \in \{0, 1\}^N \).

**Output:** Parties receive \( \langle P \rangle \) where \( P \) stably sorts \( b \).

1. Reconstruct \( b \).
2. Compute permutation \( P \) that stably sorts \( b \).
3. Output \( \langle P \rangle \).

### Sorting permutation for arbitrary key

Functionality 5 takes a vector \( x \in \mathbb{Z}_2^N \) as input and outputs shares of permutation \( P \) that stably sorts \( x \). Protocol \( \Pi_{\text{SortPerm}} \) (Algorithm 19) in [8] securely computes \( \text{SortPerm} \) with \( N \) calls to \( \Pi_{\text{A2B}} \), \( l \) calls to \( \Pi_{\text{ApplyPerm}} \), \( l \) calls to \( \Pi_{\text{SortPermBit}} \) and \( l \) calls to \( \Pi_{\text{PermComp}} \). Note that [18] considers \( \Pi_{\text{SortPerm}} \) realized through a comparison based sorting protocol, such as the one in [19]. The implementation of [18] (from [2]) uses \( \Pi_{\text{SortPerm}} \) 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: \( \text{SortPerm} \)

**Input:** Parties input \( (x) \) where \( x \in \mathbb{Z}_2^N \).

**Output:** Parties receive \( \langle P \rangle \) where \( P \) stably sorts \( x \).

1. Reconstruct \( x \).
2. Compute permutation \( P \) that stably sorts \( x \).
3. Output \( \langle P \rangle \).

### 3.3 Groupwise Operations

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

#### Privately grouped vector

Let \( x \) 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 \( g \) of length \( N \) such that \( g[i] = 1 \) if \( x[i] \) is the first element of any group. For example, let \( x = [2, 5, 6, 8, 10, 13] \) and \( g = [1, 0, 0, 0, 1, 0] \). Then entries with index 1 to 4 belong to first group and entries with index 5 to 6 belong to second group as \( g[5] = 1 \) (Assume 1-indexing). Note that \( g[1] \) 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 groupwise 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 \( 1 \leq i \leq 4 \) and \( l_i = 5, r_i = 6 \) for \( 5 \leq i \leq 6 \). Then we can define the following groupwise operations:

1. \( \text{(a)} = \Pi_{\text{GroupSum}} (\langle x \rangle, \langle g \rangle) \)
2. \( \text{(b)} = \Pi_{\text{GroupPrefixSum}} (\langle x \rangle, \langle g \rangle) \)
3. \( \text{(c)} = \Pi_{\text{GroupMax}} (\langle x \rangle, \langle g \rangle) \)
4. \( \text{(d)} = \Pi_{\text{GroupFirstOne}} (\langle x \rangle, \langle g \rangle) \) defined when \( x \in \{0, 1\}^N \) where \( \forall i \in [N] \)

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

By instantiating \( \text{SortPermBit} \) and \( \text{ApplyPerm} \) 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...
Table 1: Communication (in bits) and Rounds of Protocols

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 \( D \in \begin{pmatrix} \mathbb{Z}_2^m \times \{0, 1\} \end{pmatrix}^N \) where \( m \) is the number of attributes and \( N \) is the number of samples in dataset \( D \). The height \( h \) of the decision tree to be built is public information. \( D[i] = (x_i[1], \ldots, x_i[m], y[i]) \) are the attribute and label values for the \( j^{th} \) sample. \( x_i \in \mathbb{Z}_2^N \) is a vector of all the \( N \) values of the \( i^{th} \) attribute and \( y \in \{0, 1\} \) is the vector of \( N \) labels.

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

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 \( \text{Layer}^{(k)} \), the left child has node id NID and right child has node id \( NID + 2^k \) in \( \text{Layer}^{(k+1)} \). The node id NID is set to 0 for the root node in \( \text{Layer}^{(1)} \).

Final output. The final output of the secure training protocol \( \Pi_{\text{Train}} \) (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 \( \left( \text{Layer}^{(k)} \right) \equiv (\text{NID}_k, (A_k, T_k)) \) where

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

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

For a given layer, \( \left( \text{NID}_k[i], A_k[i], T_k[i] \right) \) are the node id, attribute and threshold of \( i^{th} (1 \leq i \leq n_k) \) node in the layer.

For the leaf node layer i.e. \( k = h+1 \), parties output \( \left( \text{Layer}^{(h+1)} \right) \equiv (\text{NID}_{h+1}, (L_{h+1})) \) where \( L_{h+1} \) 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 \( \{D_k, g_k, P_k\}_{i \in \{m, \text{NID}\}} \), each of length \( N \), where

1. \( D_k \) is the training dataset grouped by nodes in layer \( k \),
2. \( g_k \) is the group flag vector encoding groups in \( D_k \),
3. \( P_k \) is a polynomial of degree \( m \) that encodes each group.
Securely Training Decision Trees Efficiently  

Securely Training Decision Trees Efficiently CCS ’24, October 14–18, 2024, Salt Lake City, UT, USA

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>height of tree</td>
</tr>
<tr>
<td>m</td>
<td>number of attributes</td>
</tr>
<tr>
<td>N</td>
<td>number of training samples</td>
</tr>
<tr>
<td>ℓ</td>
<td>bitlength of attributes</td>
</tr>
<tr>
<td>D</td>
<td>Labeled training dataset</td>
</tr>
<tr>
<td>xᵢ</td>
<td>ℓᵗ attribute vector</td>
</tr>
<tr>
<td>y</td>
<td>Label vector</td>
</tr>
<tr>
<td>g</td>
<td>Group flag vector</td>
</tr>
<tr>
<td>NID</td>
<td>node id</td>
</tr>
<tr>
<td>A</td>
<td>splitting attribute</td>
</tr>
<tr>
<td>T</td>
<td>splitting threshold</td>
</tr>
<tr>
<td>L</td>
<td>Leaf label</td>
</tr>
<tr>
<td>Layer(ᵏ)</td>
<td>𝑘ᵗ layer information</td>
</tr>
<tr>
<td>State(ᵏ)</td>
<td>Invariant state at 𝑘ᵗ layer</td>
</tr>
<tr>
<td>𝑛ₖ</td>
<td>Number of nodes at Layer(ᵏ)</td>
</tr>
</tbody>
</table>

Table 2: Variables and their description.

(3) \( Pᵢₖ \) (for all \( i \in \{ m \} \)) is the permutation that sorts \( D_k \) based on the \( ℓᵗ \) attribute value within each group; and
(4) \( NID \) is the vector of node ids of samples in \( D_k \). For example, if \( D_k[i] \) belongs to the subset of the dataset associated with the \( ℓᵗ \) node, \( NID[i] = j \).

To train the \( 𝑘ᵗ \) layer (\( k \in \{ h + 1 \} \)), parties input \( State(ᵏ) \) to \( Π_{TrainInternalLayer} \) (Protocol 12) and receive \( Layer(ᵏ) \) and \( State(ᵏ+1) \) as output where \( State(ᵏ+1) \) 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 \( 𝑘ᵗ \) layer partition the dataset which is encoded by a group flag vector \( g_k \) 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. \( g_k[i] = g_k[j] = 1 \) and \( g_k[i] = 0 \) for all \( i < l < j \), all samples from \( D_k[i] \) to \( D_k[j - 1] \) belong to the same group. As a layer is trained and new groups are formed, the partition of the dataset changes and the dataset \( D_k \) has to be rearranged to ensure that samples in the same group appear consecutively. Vectors \( g_k \) and \( P_k \) and \( NID \) are also updated accordingly. The output \( State(𝑘+1) \) 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 (\( x, y \)) 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] communication. 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 \( Π_{SortPermBit} \) (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 \( (x, y) \) 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 \( x \) to \( x_{new} \) as new groups are created due to the split.

The key observation to address this challenge is that even though \( x \) and \( x_{new} \) are different, \( x_{new} \) is simply a permutation of \( x \). Similarly, \( x'_{new} \) is a permutation of \( x' \). This means that there are permutations \( \psi, \phi \) such that \( x_{new} = Permute(x, \phi) \) and \( x'_{new} = Permute(x', \psi) \).
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 \( \Pi_{\text{GroupMax}} \) (on \( N \)-length vector) followed by \( N \) invocations of \( \Pi_{\text{Max}} \) (on \( m \)-length vector) with \( N \) invocations of \( \Pi_{\text{Max}} \) (on \( m \)-length vector) and 1 invocation of \( \Pi_{\text{GroupMax}} \) (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 occurring 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 \( P_i \) for all \( i \in [m] \) that sorts \( D \) 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 \( \Pi_{\text{SortPerm}} \) that computes the sorting permutation with \( O(N \log N) \) communication in \( O(\log N) \) rounds for a vector of length \( N \). \( \Pi_{\text{SetupPerm}} \) (Protocol 6) requires \( m \) calls to \( \Pi_{\text{SortPerm}} \).

### 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 \( \text{State}^{(k)} = \{D_k, \mathbb{G}_k, \{P'_i\}_{i \in [m]}, \text{NID} \} \) which satisfies the following invariant:

---

**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 \( x_{\text{new}} \) is the new attribute vector with new groups which is obtained by stably sorting \( x \) based on \( b \). \( x'_{\text{new}} \) is the new sorted attribute vector. As ordering of \( x \) changes due to new groups, the permutation that sorts \( x_{\text{new}} \) to \( x'_{\text{new}} \) also changes.

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

Given the old sorting permutation \( (P) \) (that sorts \( x \)), \( (\phi^{-1}) \) and \( (\psi) \), the new sorting permutation (that sorts \( x_{\text{new}} \)) is \( P' = \psi \circ P \circ \phi^{-1} \) (see Figure 3) which parties can compute with \( O(N) \) using \( \Pi_{\text{PermComp}} \). Moreover, we show that \( (\phi^{-1}) \) and \( (\psi) \) can be computed with \( O(N) \) using \( \Pi_{\text{SortPermBit}} \). Thus, given the old sorting permutation \( (P) \), parties can compute the new sorting permutation \( (P') \) 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 \((x, y)\) 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 \( \Pi_{\text{GroupMax}} \) (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 \( \Pi_{\text{Max}} \) (over \( m \)-length vectors). We make two important observations here; (1)
Protocol 6: $\Pi_{\text{SetupPerm}}$

\begin{verbatim}
Input: Dataset $\langle D \rangle = \langle x_1, \ldots, x_m \rangle$, $\langle y \rangle$
Output: Sorting permutation $\langle P_1' \rangle$ for all $i \in [m]$
Cost: $O(mn)$
1 for $i \in [m]$ do
  2 \{ $\langle P_1' \rangle = \Pi_{\text{SortPerm}}\left( \langle x_i \rangle \right)$ \}
end
4 Output $\{ \langle P_1' \rangle \}_{i \in [m]}$.
\end{verbatim}

(1) $D_k$ is the training dataset grouped by nodes in layer $k$.
(2) $g_k$ is the group flag vector that marks the starting of groups in $D_k$.
(3) $P_1'$ (for all $i \in [m]$) is the permutation that sorts $D_k$ based on $i^{th}$ attribute within each group and
(4) vector NID is the node id (or group id) of each sample in $D_k$.

For example if $D_k[i]$ belongs to group $i^{th}$ node, 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 $\langle \text{Layer}^{(k)} \rangle$
defined in Section 4 and $\langle \text{State}^{(k+1)} \rangle$ i.e. updated state required to train the next layer.

For the base case, $\langle D_1 \rangle = \langle D \rangle$, $\langle g_1 \rangle = \langle \{1, 0, \ldots, 0\} \rangle$, NID = $\langle \{1, 1, \ldots, 1\} \rangle$ as Layer 1 only has the root node. Hence, there is no partitioning of nodes. Moreover, from the setup phase, we have $\langle P_1' \rangle = \Pi_{\text{SortPerm}}(\langle x_i \rangle)$. Thus,

$\langle \text{State}^{(1)} \rangle = \langle D_1 \rangle$, $\langle g_1 \rangle$, $\langle P_1' \rangle_{i \in [m]}$, (NID)

satisfies the invariant for the base case.

The TrainInternalLayer functionality takes as input $k$ and shares of State$^{(k)}$, and outputs shares of Layer$^{(k)}$, State$^{(k+1)}$ where Layer$^{(k)}$ is the $k^{th}$ decision tree layer and 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 State$^{(k)}$ to algorithm 12 satisfies the invariant described above, the output State$^{(k+1)}$ also satisfies the invariant. Therefore by induction, State$^{(k)}$ satisfies the invariant for all $k \in [h + 1]$.

The $\Pi_{\text{TrainInternalLayer}}$ 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 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. ComputeGini takes an attribute (x) and label (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 $p_1q_2 < p_2q_1$ where $s_1 = (p_1, q_1)$.

$\Pi_{\text{ComputeGini}}$ (Protocol 7) securely computes the functionality. The sorted attribute vector allows us to compute the split for each threshold efficiently. If threshold is such that $x[i] < t < x[i + 1]$, then the two partitions of the dataset are $\{x[1], \ldots, x[i]\}$ and $\{x[i + 1], \ldots, x[N]\}$ respectively since the attribute vector is sorted. The threshold $t[i]$ corresponding to an attribute element $x[i]$ is set to $(x[i] + x[i + 1]) \div 2$. However, this involves the expensive secure division operation, so we modify the test $x[j] < t < x[j] < t'$ such that $t' = 2t$ just as in [18]. $t'$ would then be computed as $x[i] + x[i + 1]$. Protocol $\Pi_{\text{ComputeGini}}$ 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 y labels changes based on ordering of samples, the Group Sum remains same. In [18], the Group Sum of y labels is computed for each attribute (GroupSum invocations). We instead compute Group Sum of y labels only once resulting in saving $251m \div Nh$ bits of communication. This is only a minor optimization and does not change the asymptotic complexity of the protocol.

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

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 $x[i] = x[i + 1]$, for $1 \leq \alpha < i < \beta < N$, we set $t[i] = \text{MinValue}$ for $\alpha \leq i < \beta$ and $t[i] = |x[i] + x[i + 1]|$.

The correctness follows from Expf. 1 which is derived in [3]. $\Pi_{\text{ComputeGini}}$ requires 2 calls to $\Pi_{\text{GroupPrefixSum}}, 11N$ calls to $\Pi_{\text{MULT}}$ and N calls to $\Pi_{\text{EQ}}$.
We note that the order of operations can be reversed, reducing the cost of operations. For this, we have to show that reversing the order of computations has no effect on the correctness of the construction. To this end, we will show that the order of operations in Protocol 7 does not change the output.

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

\[ \text{Input: } \langle g \rangle, \langle x \rangle, \langle y \rangle, \langle sy_0 \rangle, \langle sy_1 \rangle \]
\[ \text{Output: } (s), (t) \text{ of length } N. \]
\[ \text{Cost: } O(N) \]

1. $\langle u_0 \rangle = \text{GroupPrefixSum}(\langle y \rangle, \langle g \rangle)$
2. $\langle u_1 \rangle = \text{GroupPrefixSum}(\langle -y \rangle, \langle g \rangle)$
3. $\langle w_0 \rangle = \langle sy_0 \rangle - \langle u_0 \rangle$ for $b \in \{0, 1\}$
4. $\langle w \rangle = \langle w_0 \rangle + \langle w_1 \rangle$
5. $\langle p \rangle = \langle w \rangle \times \left(\langle u_0 \rangle^2 + \langle u_1 \rangle^2\right) + \langle u \rangle \times \left(\langle w_0 \rangle^2 + \langle w_1 \rangle^2\right)$
6. $\langle q \rangle = \langle u \rangle \times \langle w \rangle$
7. $\langle s \rangle = \langle \langle p \rangle, \langle q \rangle \rangle$
8. $\langle t[i] \rangle = \langle x[i] \rangle + \langle x[i+1] \rangle$ for $i \in [N-1]$, $\text{t}[N] = \text{MinValue}$
9. $\langle r[i] \rangle = \langle g[i+1] \rangle$ OR $\langle x[i] \rangle = \langle x[i+1] \rangle$ for $i \in [N-1]$, $\text{r}[N] = 1$
10. $(s), (t) \left\{ \text{in an oblivious manner, parties have to compute \( r[i] \) for each layer, parties have to update \( t[i+1] \) \}

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

$\Pi_{\text{UpdateTest}}$ requires $2m$ calls to $\Pi_{\text{ApplyPerm}}, m$ calls to $\Pi_{\text{ComputeGini}}, N$ calls to $\Pi_{\text{GroupMax}}, 1$ call to $\Pi_{\text{GroupMax}}, 3N$ calls to $\Pi_{\text{MULT}}$ and $2N$ calls to $\Pi_{\text{EQ}}$.

4.4.3 Applying Tests. Once parties have $\langle A \rangle, \langle T \rangle$, parties compute the partition of nodes based on comparison of selected attributes with their corresponding thresholds.

The cost of $\Pi_{\text{ApplyTest}}$ is $O(hmN)$, which is insignificant in the total communication and we directly use the subprotocol from [18]. If $A[j] = i^*$, the $i^*$th attribute value has to be selected for the $i^*$th sample (i.e., $x_{i'}[j]$) to compare with the threshold $T[j]$. To fetch $\langle x_{i'}[j] \rangle$ in an oblivious manner, parties have to compute $\langle e_{i'} \rangle \in \{0, 1\}^m$ from $\langle i^* \rangle$ which is its one hot encoding. We compute the one hot vector using $m$ comparisons as:

\[ \langle e_{i'} \rangle = \left\{ \langle i^* \rangle = 1, \langle i^* \rangle = 2, \ldots, \langle i^* \rangle = 2^m \right\} \]

Then the parties can compute $x_{i'}$ with dot product $x \cdot e_{i'}$ and compare it with the threshold. $\Pi_{\text{ApplyTest}}$ (Protocol 9) requires $N$ calls to $\Pi_{\text{MULT}}, N$ calls to $\Pi_{\text{LT}}$ and $mN$ calls to $\Pi_{\text{EQ}}$.

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

\[ \text{Input: } \langle D_k \rangle, \langle g_k \rangle, \langle P_k \rangle, \langle sy_0 \rangle, \langle sy_1 \rangle \]
\[ \text{Output: } (A), (T) \text{ of size } N. \]
\[ \text{Cost: } O(mn + nN) \]

1. for $i \in [m]$
2. $\langle u_i \rangle = \Pi_{\text{ApplyPerm}}(\langle x_i \rangle, \langle P_k \rangle)$
3. $\langle v_i \rangle = \Pi_{\text{ApplyPerm}}(\langle y \rangle, \langle P_k \rangle)$
4. $\langle s_i \rangle, \langle t_i \rangle = \Pi_{\text{ComputeGini}}(\langle g_k \rangle, \langle u_i \rangle, \langle v_i \rangle)$
5. end

6. for $j \in [N]$ do
7. $\langle s[j] \rangle, \langle t[j] \rangle, \langle a[j] \rangle = \Pi_{\text{GroupMax}}(\langle s_i \rangle, \langle t[i] \rangle, \langle i \rangle)$
8. end

9. $\langle A \rangle, \langle T \rangle = \Pi_{\text{GroupMax}}(\langle g_k \rangle, \langle s \rangle, \langle a \rangle, \langle t \rangle)$
10. $\langle f \rangle = \langle sy_0 \rangle^2 \text{ OR } \langle sy_1 \rangle^2$
11. $(A), (T) = \text{IfElse}(\langle f \rangle; 1, \text{MinValue}; \langle A \rangle, \langle T \rangle)$

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 $b$ is equal to 0 or 1. To avoid sorting the dataset at each layer, parties have to update State$^{(k+1)}$ such that the new groups satisfy the invariant described in Section 4.4. Functionality $\Pi_{\text{UpdateState}}$ takes shares of State$^{(k)}$ and $b$ as input and outputs State$^{(k+1)}$ such that 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. $\Pi_{\text{UpdateState}}$ (Protocol 10) securely computes this functionality.

4.4.3.2.3 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 provide a correctness proof for the UpdateStateLayer which outputs (stored in tuple LayerInternal) the splitting attribute and threshold for all nodes in the layer. 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 $D_k$ and $g$ based on values of $b$ to obtain $(D_{k+1}), (g_{k+1})$. We compute the updated permutations $[\{p^i_{k+1}\}]_{i \in [m]}$.

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

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

**Input:** $(\text{State}^{(k)}) = ((D_k), (g_k), \{p^i_k\}_i, (\text{NID})$, $(b), k$)

**Output:** $(\text{State}^{(k+1)})$

**Cost:** $O(mN)$

1. $(Q) = \Pi_{SortPermBit}((b))$
2. $(D_{k+1}) = \Pi_{ApplyPerm}((D_k), (Q))$
3. $(\text{NID}) = 2^{k-1} (b) + (\text{NID})$
4. $(g) = \Pi_{GroupFirstOne}((g_k), (b))$
   + $\Pi_{GroupFirstOne}((g_k), (-b))$
5. $(g_{k+1}) = \Pi_{ApplyPerm}((g), (Q))$
6. for $i \in [m]$ do
5. $(b^i) = \Pi_{SortPermBit}((b), (p^i_k))$
7. $(Q^i) = \Pi_{ApplyPerm}((p^i_k), (Q))$
8. $(P^i_k) = \Pi_{PermComp}((R^i), (Q^i))$
9. end
10. Output $(\text{State}^{(k+1)}) = ((D_{k+1}), (g_{k+1}), \{p^i_{k+1}\}_i, (\text{NID})).$

#### 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 LayerInternal). Subprotocol $\Pi_{StoreLayer}$ (Protocol 11) which outputs Layer^{(k)} tuple is not a bottleneck in communication efficiency $O(bmN)$ and we use it as is from [18].

Parties have $(\text{NID}), (A), (T)$ 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 $(\text{NID}_k), (A_k), (T_k)$ where $\text{NID}_k[i], A_k[i], T_k[i]$ stores the node id, attribute and threshold of $i^{th}$ node. For the leaf node layer, parties output $(\text{NID}_{h+1}), (A_{h+1}), (T_{h+1})$.

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

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

**Input:** $(\text{NID}), (A), (T), (g_k)$ of length $N$

**Output:** $(\text{NID}_k), (A_k), (T_k)$ of length $n_k = \min(2^{k-1}, N)$

**Cost:** $O(N)$

1. $(P) = \Pi_{SortPermBit}((g_k))$
2. $(\text{NID}_k) = \Pi_{ApplyPerm}((\text{NID}), (P))$ //First $n_k$ terms
3. $(A_k) = \Pi_{ApplyPerm}((A), (P))$ //First $n_k$ terms
4. $(T_k) = \Pi_{ApplyPerm}((T), (P))$ //First $n_k$ terms
5. Output $(\text{Layer}^{(k)}) = ((\text{NID}_k), (A_k), (T_k))$.

#### 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 $\Pi_{GroupSum}$, 1 call to $\Pi_{TestSelection}$, 1 call to $\Pi_{ApplyTests}$, 1 call to $\Pi_{StoreLayer}$ and 1 call to $\Pi_{UpdateState}$.

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

**Input:** $(\text{State}^{(k)}) = ((D_k), (g_k), \{p^i_k\}_i, (\text{NID})$, $k$

**Output:** $(\text{Layer}^{(k)}) = ((\text{State}^{(k)}), (b), k)$

**Cost:** $O(mN + nN)$

1. $(sy_b) = \text{GroupSum}((y_k), (g_k))$
2. $(sy_t) = \text{GroupSum}((y_k), (g_k))$
3. $(A), (T) = \Pi_{TestSelection}((D_k), (g_k), \{p^i_k\}_i)$
4. $(\text{Layer}^{(k)}) = \Pi_{StoreLayer}((\text{NID}), (A), (T), (g_k), k)$
5. $(b) = \Pi_{ApplyTest}((D_k), (A), (T))$
6. $(\text{State}^{(k+1)}) = \Pi_{UpdateState}((\text{State}^{(k)}), (b), k)$
7. Output $(\text{Layer}^{(k)}), ((\text{State}^{(k+1)}))$.

#### 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 $y$ for samples in the training dataset belonging to this node. This can be computed using $\Pi_{GroupSum}$. The final layer labels are stored in two vectors $\text{NID}, \text{L}$ of length $n_{h+1}$ where $\text{NID}$ stores the node id and $\text{L}$ stores the corresponding labels. We can modify the $\Pi_{StoreLayer}$ accordingly to store the leaf layer output. $\Pi_{TrainLeafLayer}$ (Protocol 13) requires 2 calls to $\Pi_{GroupSum}$, 1 call to $\Pi_{StoreLayer}$ and $\Pi_{UpdateState}$.

#### 4.6 End to end training

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

The communication cost of $\Pi_{SetupPerm}, \Pi_{TrainInternalLayer}$ and $\Pi_{TrainLeafLayer}$ are $O(mN \log N), O(mN + N \log N)$ and $O(N)$ respectively. The total communication cost of $\Pi_{TrainDecisionTree}$ 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].
5 Security Proof

Let \text{Train} be the functionality that takes as input the shares of labelled dataset (D) and height \( h \) and outputs shares of trained decision tree of height \( h \) i.e. \( \text{Layer}^{(i)} \). 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 \( \Pi_{\text{Train}} \) can be simulated by a simulator with access to functionality \text{Train} in the ideal world.

The protocols for building blocks \( \Pi_{\text{MULT}}, \Pi_{\text{AB}}, \Pi_{\text{EQ}} \) and \( \Pi_{\text{shuffle}} \), \( \Pi_{\text{applyPerm}}, \Pi_{\text{permComp}} \) 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 \( \Pi_{\text{computeGini}}, \Pi_{\text{applyTest}} \) and \( \Pi_{\text{storeLayer}} \) was shown in \([18]\). We show the correctness of new protocols \( \Pi_{\text{TestSelection}} \) and \( \Pi_{\text{updateState}} \).

5.1 Correctness of TestSelection

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

**Proof.** To demonstrate correctness of \( \Pi_{\text{TestSelection}} \), we prove (Lemma 5.2) that reversing the order of invocations of \( \Pi_{\text{GroupMax}} \) and \( \Pi_{\text{Max}} \) does not change the output in steps 6–9 in Protocol 8, \( \{s_i\}_{i \in [m]} \) are the length \( N \) vectors for Gini index for all attributes which are correctly computed from the correctness of \( \Pi_{\text{computeGini}} \) protocol.

Suppose \( \{s_i\}_{i \in [m]} \) are \( m \) vectors each of length \( N \). \( g \) is group flag vector denoting groups in \( s \) 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:

\[
a = \Pi_{\text{Max}}(s_1, \ldots, s_m) \quad \text{and} \quad b_i = \Pi_{\text{GroupMax}}(g, s_i)
\]

where

\[
a[j] = \max_{i \in [m]} (s_i[j]) \quad \text{and} \quad b_i[j] = \max_{l_j \leq k \leq r_j} (s_i[k])
\]

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

\[
\Pi_{\text{GroupMax}}(g, a) = \Pi_{\text{Max}}(b_1, \ldots, b_m)
\]

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

<table>
<thead>
<tr>
<th>( (n, m, h) )</th>
<th>Comm. (([18]))</th>
<th>Comm. (Ours)</th>
<th>IF</th>
</tr>
</thead>
<tbody>
<tr>
<td>((13, 11, 4))</td>
<td>14.3</td>
<td>3.6</td>
<td>4.0</td>
</tr>
<tr>
<td>((13, 20, 4))</td>
<td>25.9</td>
<td>6.1</td>
<td>4.2</td>
</tr>
<tr>
<td>((13, 11, 10))</td>
<td>35.8</td>
<td>6.5</td>
<td>5.5</td>
</tr>
<tr>
<td>((13, 11, 20))</td>
<td>71.6</td>
<td>11.4</td>
<td>6.3</td>
</tr>
<tr>
<td>((16, 11, 10))</td>
<td>310.8</td>
<td>53.2</td>
<td>5.8</td>
</tr>
<tr>
<td>((16, 20, 20))</td>
<td>1,123.7</td>
<td>151.6</td>
<td>7.4</td>
</tr>
<tr>
<td>((16, 40, 50))</td>
<td>5,599.8</td>
<td>631.4</td>
<td>8.9</td>
</tr>
<tr>
<td>((18, 20, 20))</td>
<td>4,730.4</td>
<td>612.7</td>
<td>7.7</td>
</tr>
<tr>
<td>((18, 50, 50))</td>
<td>29,453.8</td>
<td>3,117.7</td>
<td>9.4</td>
</tr>
<tr>
<td>((18, 100, 100))</td>
<td>119,282.2</td>
<td>11,539.6</td>
<td>10.3</td>
</tr>
<tr>
<td>((19, 11, 10))</td>
<td>2,660.5</td>
<td>434.9</td>
<td>6.2</td>
</tr>
<tr>
<td>((19, 20, 50))</td>
<td>24,236.9</td>
<td>2,793.4</td>
<td>8.7</td>
</tr>
<tr>
<td>((19, 40, 40))</td>
<td>38,667.5</td>
<td>4,169.5</td>
<td>9.3</td>
</tr>
</tbody>
</table>

6 Experiments

Hamada et al. \([18]\) is the state-of-the-art for secure decision tree training and we use its implementation\(^\text{3}\) 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\(^\text{4}\) 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

\(^\text{3}\) Code from the original paper is unavailable.

\(^\text{4}\) Link to the code.
of (3,2)-replicated secret-sharing based protocols over $\mathbb{Z}_{128}$ ring\(^5\) and use the oblivious radix sort protocol from [8] to implement $\Pi_{\text{SortPerm}}$. 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

$$h(3712mnN + 99531mN + 16862N)$$

$$h(12806mN + 2286Nn + 19568N) + 81331mN$$

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 $n, m, h$, 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 $n, m, h$.

**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 $\Pi_{\text{GroupMax}}$ 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.
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 $O(h m N \log N)$ as the cost of computing the minimum Gini index is $O(m N \log N)$. In contrast, our work reduces this cost to $O(m N + N \log N)$. Second, they secret share cleartext values over the small ring ($\mathbb{Z}_{2^{128}}$) and convert these values to shares over large rings ($\mathbb{Z}_{2^{4096}}$) 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 $\approx 1.4x$ 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 secure 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 $y_1, \ldots, y_c \in \{0, 1\}^N$ where $(y_1[l], \ldots, y_c[l])$ is a one-hot encoding of the label of the $l$th sample. The modified Gini index expression in this case will be: $\text{Gini}(D) = 1 - \sum_{k=1}^{c} (|D_{y_k}|/|D|)^2$. 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 $O(m N \log N + cm N + hN \log N)$.

### 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


Algorithm 14. (or \( d \)) The algorithm for cleartext decision tree training is presented in FOCS [39].

Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In FOCS.

Ming-Jun Xiao, Liu-Sheng Huang, Yong-Long Luo, and Hong Shen. 2005. Privacy-preserving decision trees over vertically partitioned data. In CCS.


Yehuda Lindell and Benny Pinkas. 2000. Privacy Preserving Data Mining. In CRYP'TO.

Wen-je Li, Zhicheng 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.


Qingkai Ma and Ping Deng. 2008. Secure Multi-party Protocols for Privacy Preserving Data Mining. In WISA.


Ke Wang, Yabo Xu, Rong She, and Philip S. Yu. 2006. Classification Spanning Private Databases. In AAI.


Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In FOCS.

Protocol 14: Plaintext decision tree training

**Input**: Labelled training dataset \( D \), maximum height \( h \).

**Output**: Decision tree \( T \)

1. If \( h = 0 \) then
2. Let \( o \) be a leaf node.
3. Label(\( o \)) is the most occurring label in \( D \). Output tree with \( o \) as root node.
4. Else
5. Find \( j, t \) such that \( x_j < t \) minimizes Gini index of the split.
6. Let \( D_1 = D_{x_j < t} \) and \( D_r = D_{x_j \geq t} \) be partition of \( D \) based on test.
7. Compute \( T_l = \text{Train}(D_l, h - 1) \) and \( T_r = \text{Train}(D_r, h - 1) \).
8. Let \( o \) be an internal node with test \( x_j < t \) and with left child \( T_l \), right child \( T_r \).
9. Output tree with \( o \) as root node.
10. End

where \( D_{b \land y = b} = \{ (d, y) \in D \mid d = b' \land y = b \} \).

This expression is similar to that of the Gini index of continuous variables and can be computed securely by simply modifying \( \Pi_{\text{ComputeGini}} \). 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 \( i \in \text{domain}(d) \) induces a partition of dataset as \( \{ D_{d = i} \} \). 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**: \( \Pi_{\text{Train}} \)

**Input**: dataset \( (D) \), height \( h \)

**Output**: \( \left\{ \left( \text{Layer}^{(i)} \right) \right\}_{i \in \mathbb{N}^+} \)

\[ \text{Cost} = O(mN \log N + h(mN + N \log N)) \]

1. \( (D_1) = (D) \)
2. \( (g_1(i)) = 1 \) and \( (g_1(i)) = 0 \) for \( i \in [2, N] \)
3. \( (\text{NID}(i)) = 1 \) for all \( i \in [N] \)
4. \( \left\{ \left( P_i^{(i)} \right) \right\} = \Pi_{\text{SetupPerm}} ((D_1)) \)
5. \( \left( \text{State}^{(1)} \right) = (D_1, g_1, \left\{ P_i^{(i)} \right\}_{i \in [m], \text{NID}}) \)
6. For \( k = 1 \) to \( h \) do
7. \( \left\{ \text{Layer}^{(k)} \right\}, \left( \text{State}^{(k+1)} \right) = \Pi_{\text{TrainInternalLayer}} \left( \left( \text{State}^{(k)} \right), k \right) \)
8. End
9. \( \left( \text{Layer}^{(h+1)} \right) = \Pi_{\text{TrainLeafLayer}} \left( \left( \text{State}^{(h+1)} \right) \right) \)
10. Output \( \left\{ \left( \text{Layer}^{(i)} \right) \right\}_{i \in \mathbb{N}^+} \)

A Plaintext Decision Tree Training

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

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 \( D \) has attribute values for a binary attribute \( d \in \{ 0, 1 \} \). Then the Gini index of splitting from test \( d = 0 \) (or \( d = 1 \)) is defined as

\[
G_d = \frac{|D_{d=0}|}{|D|} \text{Gini}(D_{d=0}) + \frac{|D_{d=1}|}{|D|} \text{Gini}(D_{d=1})
\]

where \( \text{Gini}(D) = 1 - \sum_{b \in \{0,1\}} |D_{y=b}|^2/|D|^2 \). Minimizing \( G_d \) is same as maximizing \( G_d' \)

\[
G_d'(D) = \left( \frac{|D_{d=0}|}{|D|} \left( \frac{|D_{d=0} \land y=0|^2 + |D_{d=0} \land y=1|^2}{|D|} \right) 
+ \frac{|D_{d=1}|}{|D|} \left( \frac{|D_{d=1} \land y=0|^2 + |D_{d=1} \land y=1|^2}{|D|} \right) \right) / (|D_{d=0}| + |D_{d=1}|)
\]
D Protocols for Element wise operations

The protocols realizing element wise operations are as follows:

- **Secure Multiplication:** denoted by \( \langle z \rangle = \Pi_{\text{MULT}}(\langle u \rangle, \langle v \rangle) \).
  We use techniques from [6]. Parties have shares \( \langle u \rangle, \langle v \rangle \in \mathbb{Z}_{z'} \) and want to compute shares \( \langle z \rangle \in \mathbb{Z}_{z} \) where \( z = u \cdot v \). Then

\[
  z = (u_0 + u_1 + u_2) \cdot (v_0 + v_1 + v_2) \\
  = (u_0v_0 + u_0v_1 + u_1v_0) + (u_1v_1 + u_2v_2) + (u_2v_2 + u_2v_0 + u_0v_2) \\
  = z_0 + z_1 + z_2
\]

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 \( z_i \) such that \( a_0 + a_1 + a_2 = 0 \) which is added to \( z_i \)). Each party can obtain replicated shares of \( z \) with communication of \( 3 \ell \) 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 Sum Decomposition:** denoted by \( \langle u \rangle^B = \Pi_{\text{ADD}}(\langle u \rangle) \).
  Parties have RSS \( \langle u \rangle \in \mathbb{Z}_{z'}^B \) and want to compute shares of bit decomposition \( \langle u \rangle^B = \left( \langle u\rangle_{1}^B, \ldots, \langle u\rangle_{\ell}^B \right) \) 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 \( \langle b \rangle = \Pi_{\text{COMPARISON}}(\langle u \rangle, \langle v \rangle) \).
  Parties have RSS \( \langle u \rangle, \langle v \rangle \in \mathbb{Z}_{z'}^B \) and want to compute \( \langle b \rangle \in \mathbb{Z}_{z'}^B \) where \( b = \langle u < v \rangle \). Parties compute \( \langle z \rangle = \langle u - v \rangle \) locally and compute \( b = \langle z < 0 \rangle \) by extracting the most significant bit of \( z \) using techniques from [30] with \( 2\ell \) AND gates in \( \log \ell \) rounds. Note that \( \text{MSB}(z) = 1 \) if \( z < 0 \) and \( \text{MSB}(z) = 0 \) otherwise. The concrete cost of secure comparison using (3,2) RSS is \( \approx 15 \ell \) bits.

- **Secure Equality:** denoted by \( \langle b \rangle = \Pi_{\text{EQUAL}}(\langle u \rangle, \langle v \rangle) \).
  Parties have RSS \( \langle u \rangle, \langle v \rangle \in \mathbb{Z}_{z'}^B \) and want to compute \( \langle b \rangle \in \mathbb{Z}_{z'}^B \) where \( b = \langle u = v \rangle = \langle u \oplus v \rangle \oplus \langle 0 \rangle \). 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 \( D_{k+1} \) by stably sorting \( D_k \) based on \( b \). We prove that \( D_{k+1} \) has correctly grouped elements.

**Lemma F.1.** Entries of the same group appear consecutively in \( D_{k+1} \) where \( D_{k+1} \) is obtained by sorting \( D_k \) based on values of \( b \).

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

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

- \( 2) \ y = y' \): This is a contradiction as \( D_{k+1} \) is obtained by sorting based on \( y \).
\textbf{Lemmas.}

Let $D_k$ be the dataset grouped by nodes in $k^{th}$ layer. $P_k^i$ is the permutation that sorts $D_k$ according to $i^{th}$ attribute. Let $D_k^i = \text{Permute} \left( D_k, P_k^i \right)$. Let $D_{k+1}$ be the dataset grouped by nodes in $(k+1)^{th}$ layer. $P_{k+1}^i$ is the new permutation that sorts $D_{k+1}$ according to $i^{th}$ attribute. Let $D_{k+1}^i = \text{Permute} \left( D_{k+1}, P_{k+1}^i \right)$. Suppose we are given $P_k^i D_{k+1}$ and have to compute $P_{k+1}^i$. From the definition of $D_k^i$, $D_{k+1}^i$, we have
\begin{align*}
D_k^i[P_k^i[j]] &= D_k[j] \quad \forall i \in [m], j \in [N] (2) \\
D_{k+1}^i[P_{k+1}^i[j]] &= D_{k+1}[j] \quad \forall i \in [m], j \in [N] (3)
\end{align*}
Just like how we obtain $D_{k+1}$ from $D_k$ by sorting it based on $b$, we can obtain $D_{k+1}^i$ by sorting $D_k^i$ based on $b^i$ (can be proved similar to lemma F.1) where $b^i = \text{Permute} (b, P_k^i)$. Let $R^i$ be the permutation that sorts $b^i$. Then, we have
\begin{align*}
b^i[P_k^i[j]] &= b[j] \quad \forall i \in [m], j \in [N] (4) \\
D_{k+1}^i[R^i[j]] &= D_{k+1}^i[j] \quad \forall i \in [m], j \in [N] (5)
\end{align*}
Let $Q^i$ be the permutation obtained by applying $Q$ to $P_k^i$ i.e.
\begin{align*}
Q^i[Q[j]] &= P_k^i[j] \quad \forall i \in [m], j \in [N] (6)
\end{align*}
Using these equations, we have
\begin{align*}
D_{k+1}^i[R^i[j]] &= D_{k+1}^i[j] \quad \text{from (5)} \\
D_{k+1}^i[R^i[P_k^i[j]]] &= D_{k+1}^i[P_k^i[j]] \quad \text{from (2)} \\
D_{k+1}^i[R^i[P_{k+1}^i[j]]] &= D_{k+1}^i[j] \quad \text{from (6, 7)} \\
D_{k+1}^i[R^i[Q^i[j]]] &= D_{k+1}^i[j] \quad \text{from (2)}
\end{align*}
Therefore, $P_{k+1}^i = R^i \circ Q^i$ where $R^i$ is the permutation that sorts $b^i$ and $Q^i$ is the permutation obtained by applying $Q$ to $P_k^i$. $\Pi_{\text{UpdateState}}$ computes $R^i$ and $Q^i$ using $\Pi_{\text{ApplyPerm}}$ in steps 9 and 10 and $P_{k+1}^i$ using $\Pi_{\text{PermComp}}$ in step 11.

This concludes our correctness proof as all components of updated State($k+1$) satisfy the invariant. \hfill \Box