Ents: An Efficient Three-party Training Framework for Decision Trees by Communication Optimization

Guopeng Lin[$], Weili Han[$], Wenqiang Ruan[$], Ruisheng Zhou[$], Lushan Song[$], Bingshuai Li[+], Yunfeng Shao[+]

[$]Laboratory for Data Security and Governance, Fudan University, [+]Huawei Technologies China

ABSTRACT

Multi-party training frameworks for decision trees based on secure multi-party computation enable multiple parties to train high-performance models on distributed private data with privacy preservation. The training process essentially involves frequent dataset splitting according to the splitting criterion (e.g. Gini impurity). However, existing multi-party training frameworks for decision trees demonstrate communication inefficiency due to the following issues: (1) They suffer from huge communication overhead in securely splitting a dataset with continuous attributes. (2) They suffer from huge communication overhead due to performing almost all the computations on a large ring to accommodate the secure computations for the splitting criterion.

In this paper, we are motivated to present an efficient three-party training framework, namely Ents, for decision trees by communication optimization. For the first issue, we present a series of training protocols based on the secure radix sort protocols [18] to efficiently and securely split a dataset with continuous attributes. For the second issue, we propose an efficient share conversion protocol to convert shares between a small ring and a large ring to reduce the communication overhead incurred by performing almost all the computations on a large ring. Experimental results from eight widely used datasets show that Ents outperforms state-of-the-art frameworks by 5.5× ∼ 9.3× in communication sizes and 3.9× ∼ 5.3× in communication rounds. In terms of training time, Ents yields an improvement of 3.5× ∼ 6.7×. To demonstrate its practicality, Ents requires less than three hours to securely train a decision tree on a widely used real-world dataset (Skin Segmentation) with more than 245,000 samples in the WAN setting.

CCS CONCEPTS

• **Security and privacy → Privacy protections**; • **Computing methodologies → Machine learning**.

KEYWORDS

Secure Multi-party Computation, Decision Tree, Privacy-preserving Machine Learning

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA.

ACM ISBN 979-8-4007-0636-3/24/10…$15.00
https://doi.org/10.1145/3658644.3670274

ACM Reference Format:
Guopeng Lin[$], Weili Han[$], Wenqiang Ruan[$], Ruisheng Zhou[$], Lushan Song[$], Bingshuai Li[+], Yunfeng Shao[+] . 2024. Ents: An Efficient Three-party Training Framework for Decision Trees by Communication Optimization. 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, 19 pages. https://doi.org/10.1145/3658644.3670274

1 INTRODUCTION

Decision trees are one of the most popular machine learning models [22, 23]. They are widely used in practical applications such as medical diagnosis [19, 34, 45] and stock forecasting [5, 7, 27] due to their interpretability, which is very important in medical and financial scenarios. In these domains, black-box machine-learning models, such as neural networks, might be limited. For example, in medical diagnosis, doctors usually prefer a predictive model that not only performs well but also can identify the key biological factors directly affecting the patient's health outcome.

The accuracy of decision trees largely depends on the number of high-quality training data, which are usually distributed among different parties. For instance, in the medical diagnosis for a rare disease, each hospital usually only has several cases. Thus, the data of a hospital is usually not enough to train an acceptable model. As a result, multiple hospitals would like to collaboratively train an accurate diagnosis decision tree. However, in this case, the multiple hospitals could not directly share their private data of the rare disease to train a decision tree due to released privacy protection regulations and laws (e.g. GDPR [47]).

Multi-party training frameworks [1, 15, 18, 31] for decision trees based on secure multi-party computation (MPC for short) are proposed to resolve the above issue. However, these frameworks suffer from practical scenarios due to two communication inefficiency issues as follows: (1) Existing multi-party training frameworks for decision trees suffer from huge communication overhead in securely splitting a dataset with continuous attributes. Continuous attributes, e.g. salary, commonly exist in real-world datasets. However, securely splitting such datasets requires a vast number of secure comparison operations, resulting in huge communication overhead. Abspoel et al. [1] propose securely generating permutations from continuous attributes at the beginning of the training process. These pre-generated permutations could be used to securely sort the training samples according to corresponding attributes, so that a lot of secure comparison operations could be saved. Nevertheless, their method requires securely training each decision tree node with a padded dataset to hide the splitting information, leading to huge communication overhead that grows exponentially as the height of the decision tree increases. To limit the communication overhead

to grow linearly as the tree height increases, Hamada et al. [18] propose group-wise protocols to securely train each decision tree node with a non-padded dataset while keeping the splitting information private. However, the group-wise protocols are incompatible with the pre-generated permutations. As a result, Hamada et al.'s method requires the repeated secure generation of permutations, again resulting in huge communication overhead. (2) Existing multi-party training frameworks for decision trees suffer from huge communication overhead due to performing almost all computations on a large ring to accommodate the secure computations for the splitting criterion (e.g. Gini impurity). The secure computations for the splitting criterion are a crucial step in the secure training process of decision trees and usually involve several sequential bit-length expansion operations, such as secure multiplication and division operations. These operations often produce intermediate or final results that require twice as many bits for representation compared to their inputs. Performing these operations sequentially several times requires a large ring to represent the intermediate or final results. Therefore, most existing multi-party training frameworks for decision trees, such as [1, 15, 18], conduct almost all computations on a large ring, leading to huge communication overhead.

In this paper, we are motivated to propose an efficient three-party training framework, namely Ents. Ents enables three parties, such as the hospitals mentioned above, to efficiently train a decision tree while preserving their data privacy. In Ents, we propose two optimizations to reduce the communication overhead led by the above issues. (1) We propose a series of training protocols based on the secure radix sort protocols [18] to efficiently and securely split a dataset with continuous attributes. One of the training protocols securely updates the pre-generated permutations to be compatible with the group-wise protocols, and the other protocols leverage the updated permutations and the group-wise protocols to securely split the dataset (train the layers of a decision tree). As a result, Ents can securely train a decision tree with only linearly growing communication overhead, while eliminating the need to repeatedly generate permutations. (2) We propose an efficient share conversion protocol to convert shares between a small ring and a large ring to reduce the communication overhead incurred by performing almost all computations on a large ring. We observe that though the secure computations for the splitting criterion usually involve bit-length expansion operations like secure multiplication and division operations, other computations in the training process just involve bit-length invariant operations, such as secure addition and comparison operations, which at most cause a single-bit increase. While the bit-length expansion operations should be performed on a large ring, the bit-length invariant operations can be performed on a small ring. Therefore, we can employ our proposed efficient share conversion protocol to convert shares to a large ring when performing the bit-length expansion operations and then convert the shares back to a small ring to reduce the communication overhead of the bit-length invariant operations.

We summarize the main contributions in Ents as follows:

- Based on the secure radix sort protocols, we propose a series of efficient training protocols for decision trees.
- We propose an efficient share conversion protocol to convert shares between a small ring and a large ring.

To evaluate the efficiency of Ents [1], we compare Ents against two state-of-the-art three-party training frameworks [1, 18] for decision trees with eight widely-used real-world datasets from the UCI repository [26]. The experimental results show that Ents outperforms these two frameworks by 5.5× ∼ 9.3× in communication sizes, 3.9× ∼ 5.3× in communication rounds, and 3.5× ∼ 6.7× in training time. Notably, Ents requires less than three hours to train a decision tree on a real-world dataset (Skin Segmentation) with more than 245,000 samples in the WAN setting. These results show that Ents is promising in the practical usage (i.e. industrial deployment) of privacy preserving training for decision trees.

2 PRELIMINARIES

2.1 Decision Tree Training

Structure of decision trees. Given a training dataset $\mathcal{D}$ containing n samples, each of which consists of m continuous attributes $a_0, \cdots, a_{m-1}$ and a label y, a decision tree built on this dataset $\mathcal{D}$ is usually a binary tree. Each internal node of the decision tree includes a split attribute index i ($i \in [0, \cdots, m-1]$) and a split threshold t. The split attribute index i and the split threshold t composite a split point $(a_i, t) : a_i < t$, which is used for splitting datasets during the training process. Each leaf node of the decision tree includes a predicted label y_{pred}.

Training process for decision trees. A decision tree is usually trained from the root node to the leaf nodes using a top-down method. For training each node, the first step is to check whether the split-stopping condition is met. In this paper, we adopt the commonly used split-stopping condition [1, 3, 18], i.e. when the current node's height reaches a predefined tree's height h. If the split-stopping condition is met, the current node becomes a leaf node. Its predicted label y_{pred} is set to the most common label in its training dataset $\mathcal{D}^{node}$, which is a subset of $\mathcal{D}$. Otherwise, the current node becomes an internal node, and its split point is computed according to the chosen splitting criterion, such as the modified Gini impurity mentioned in Section 2.2. Assuming the split point is (a_i, t), the training dataset $\mathcal{D}^{node}$ of the current node is split into two sub-datasets, $\mathcal{D}^{node}_{a_i<t}$ and $\mathcal{D}^{node}_{a_i \geq t}$. Here, $\mathcal{D}^{node}_{a_i<t}$ refers to a dataset that contains all samples whose i-th attribute values are smaller than t in $\mathcal{D}^{node}$, and $\mathcal{D}^{node}_{a_i \geq t}$ refers to a dataset that contains all samples whose i-th attribute values are greater than or equal to t in $\mathcal{D}^{node}$. Then, the left child node and right child node of the current node are trained using $\mathcal{D}^{node}_{a_i<t}$ and $\mathcal{D}^{node}_{a_i \geq t}$, respectively.

2.2 Splitting Criterion

Several splitting criteria have been proposed, such as information gain [38], information gain ratio [39], and Gini impurity [33]. In this paper, we follow the previous studies [1, 18] to use the modified Gini impurity, which is easy to compute in MPC [1, 18], as the splitting criterion to implement our proposed Ents.

Modified Gini impurity. Let v be the number of labels, and a label $y \in [0, \cdots, v-1]$, the modified Gini impurity for a split point (a_i, t) in a dataset $\mathcal{D}$ is defined as follows:

[1] We opened our implementation codes at https://github.com/FudanMPL/Garnet/tree/Ents.

$$Gini_{a_i<t}(\mathcal{D}) = \frac{\sum_{l=0}^{v-1} |\mathcal{D}_{a_i<t \wedge y=l}|^2}{|\mathcal{D}_{a_i<t}|} + \frac{\sum_{l=0}^{v-1} |\mathcal{D}_{a_i \geq t \wedge y=l}|^2}{|\mathcal{D}_{a_i \geq t}|} \quad (1)$$

$\mathcal{D}_{a_i<t \wedge y=l}$ refers to a dataset that contains all samples whose i-th attribute values are smaller than t and labels are l in $\mathcal{D}$. $\mathcal{D}_{a_i \geq t \wedge y=l}$ refers to a dataset that contains all samples whose i-th attribute values are greater than or equal to t and labels are l in $\mathcal{D}$. Besides, $|\mathcal{D}_{a_i<t}|$, $|\mathcal{D}_{a_i \geq t}|$, $|\mathcal{D}_{a_i<t \wedge y=l}|$ and $|\mathcal{D}_{a_i \geq t \wedge y=l}|$ refer to the number of the samples in $\mathcal{D}_{a_i<t}$, $\mathcal{D}_{a_i \geq t}$, $\mathcal{D}_{a_i<t \wedge y=l}$ and $\mathcal{D}_{a_i \geq t \wedge y=l}$, respectively.

During the training process, the split point (a_i, t) of an internal node should be set to the one with the maximum modified Gini impurity $Gini_{a_i<t}(\mathcal{D}^{node})$ [1, 18], where $\mathcal{D}^{node}$ refers to the training dataset of the node.

2.3 Secure Multi-party Computation

MPC enables multiple parties to cooperatively compute a function while keeping their input data private. There are several technical routes of MPC, including secret sharing-based protocols [35], homomorphic encryption-based protocols [16], and garbled circuit-based protocols [11]. In this paper, we adopt the three-party replicated secret sharing (RSS for short) as the underlying protocol of Ents for its efficiency and promising security.

In this section, we first introduce the three-party replicated secret sharing. Then, we sequentially introduce the basic operations [35, 44], the vector max protocol [18], the secure radix sort protocols [10], and the group-wise protocols [18] based on RSS, which are used in our proposed training protocols in Ents.

2.3.1 Three-party Replicated Secret Sharing. To share a secret x on a ring $\mathbf{Z}_{2^\ell}$ of size 2^ℓ using RSS to three parties P_0, P_1 and P_2, a party first generates three random numbers $[\![x]\!]^0$, $[\![x]\!]^1$ and $[\![x]\!]^2$, where $x = ([\![x]\!]^0 + [\![x]\!]^1 + [\![x]\!]^2) \ mod \ 2^\ell$. Then the party sends $[\![x]\!]^i$, $[\![x]\!]^{i+1}$ (where $[\![x]\!]^{2+1}$ refers to $[\![x]\!]^0$) to P_i. In this paper, we use the notation $\langle x \rangle$ to denote the replicated shares of x, which means P_i holds $\langle x \rangle^i = ([\![x]\!]^i, [\![x]\!]^{i+1})$.

To share a vector $\vec{x}$ using RSS, the three parties leverage the above way to share each element of $\vec{x}$. For example, given a vector $\vec{x} = [2, 3, 1]$, the secret-shared vector $\langle \vec{x} \rangle$ would be $[\langle 2 \rangle, \langle 3 \rangle, \langle 1 \rangle]$.

2.3.2 Basic Operations Based on RSS. Let c be a public constant, $\langle x \rangle$ and $\langle y \rangle$ be the replicated shares of x and y, the basic operations [35, 44] based on RSS used in this paper is as follows:

- **Secure addition with constant.** $\langle z \rangle = \langle x \rangle + c$, such that $z = x + c$.
- **Secure multiplication with constant.** $\langle z \rangle = c * \langle x \rangle$, such that $z = c * x$.
- **Secure addition.** $\langle z \rangle = \langle x \rangle + \langle y \rangle$, such that $z = x + y$.
- **Secure multiplication.** $\langle z \rangle = \langle x \rangle * \langle y \rangle$, such that $z = x * y$.
- **Secure probabilistic truncation.** $\langle z \rangle = Trunc(\langle x \rangle, c)$, such that $z = \lfloor x/2^c \rfloor + bit$, $bit \in [-1, 0, 1]$.
- **Secure comparison.** $\langle z \rangle = \langle x \rangle < \langle y \rangle$, such that if $x < y$, $z = 1$, otherwise $z = 0$.
- **Secure equality test.** $\langle z \rangle = \langle x \rangle \stackrel{?}{=} \langle y \rangle$, such that if $x = y$, $z = 1$, otherwise $z = 0$.
- **Secure division.** $\langle z \rangle = \langle x \rangle \ / \ \langle y \rangle$, such that $z = x/y$.

2.3.3 Vector Max Protocol Based on RSS. The protocol *VectMax* [18] is used to securely find an element in a vector whose position matches that of a maximum value in another vector. In this protocol, the parties input two secret-shared vectors $\langle \vec{x} \rangle$ and $\langle \vec{y} \rangle$, both of which have the same size. The parties get a secret-shared element $\langle z \rangle$ as the output, such that $z = \vec{y}[i]$ when $\vec{x}[i]$ is a maximum value in $\vec{x}$ (if there are multiple maximum values, the last one is chosen). For example, given $\vec{x} = [2, 3, 1]$ and $\vec{y} = [4, 5, 6]$, $VectMax(\langle \vec{x} \rangle, \langle \vec{y} \rangle) = \langle 5 \rangle$.

2.3.4 Secure Radix Sort Protocols Based on RSS. The secure radix sort protocols are based on a data structure called a permutation. A permutation $\vec{\pi}$ is a special vector whose elements are distinct from each other and belong to $[0, n-1]$, where n is the size of the permutation. A permutation $\vec{\pi}$ could be used to reorder elements of a vector whose size is the same as the permutation. For example, given a permutation $\vec{\pi} = [3, 2, 4, 0, 1]$ and a vector $\vec{x} = [4, 9, 2, 9, 3]$, we can reorder the elements of $\vec{x}$ according to $\vec{\pi}$ to get a new vector $\vec{z}$ ($\vec{z}[\vec{\pi}[i]] = \vec{x}[i]$). Consequently, we obtain $\vec{z} = [9, 3, 9, 4, 2]$.

In this paper, we adopt the secure radix sort protocols proposed by Chida et al. [10]. To securely radix sort a secret-shared vector $\langle \vec{x} \rangle$, the parties use two protocols: *GenPerm* and *ApplyPerm*. In the protocol *GenPerm*, the parties input a secret-shared vector $\langle \vec{x} \rangle$, and get a secret-shared permutation $\langle \vec{\pi} \rangle$ as the output, where $\vec{\pi}[i]$ represents the index of $\vec{x}[i]$ in ascending order. Note that this ascending order is stable. For example, given $\langle \vec{x} \rangle = [\langle 4 \rangle, \langle 9 \rangle, \langle 2 \rangle, \langle 9 \rangle, \langle 3 \rangle]$, $\langle \vec{\pi} \rangle = GenPerm(\langle \vec{x} \rangle) = [\langle 2 \rangle, \langle 3 \rangle, \langle 0 \rangle, \langle 4 \rangle, \langle 1 \rangle]$. In the protocol *ApplyPerm*, the parties input a secret-shared permutation $\langle \vec{\pi} \rangle$ and a secret-shared vector $\langle \vec{x} \rangle$, and they get a secret-shared vector $\langle \vec{z} \rangle$ that is created by securely reordering $\langle \vec{x} \rangle$ according to $\langle \vec{\pi} \rangle$ as the output. For example, given $\langle \vec{\pi} \rangle = [\langle 2 \rangle, \langle 3 \rangle, \langle 0 \rangle, \langle 4 \rangle, \langle 1 \rangle]$ and $\langle \vec{x} \rangle = [\langle 4 \rangle, \langle 9 \rangle, \langle 2 \rangle, \langle 9 \rangle, \langle 3 \rangle]$, $\langle \vec{z} \rangle = ApplyPerm(\langle \vec{\pi} \rangle, \langle \vec{x} \rangle) = [\langle 2 \rangle, \langle 3 \rangle, \langle 4 \rangle, \langle 9 \rangle, \langle 9 \rangle]$. Thus, by first generating a secret-shared permutation $\langle \vec{\pi} \rangle$ from a secret-shared vector $\langle \vec{x} \rangle$ using the protocol *GenPerm* and then applying $\langle \vec{\pi} \rangle$ to $\langle \vec{x} \rangle$ using the protocol *ApplyPerm*, the parties can securely radix sort $\langle \vec{x} \rangle$. Additionally, if the parties apply $\langle \vec{\pi} \rangle$ to another secret-shared vector $\langle \vec{y} \rangle$ with the same size of $\langle \vec{x} \rangle$, they can securely radix sort $\langle \vec{y} \rangle$ according to $\langle \vec{x} \rangle$.

We detail the implementation of the protocol *GenPerm*, because this protocol inspires our proposed permutation updating protocol *UpdatePerms* (Protocol 3 in Section 4.2.3). *GenPerm* is implemented with four subprotocols *BitVecDec*, *ApplyPerm*, *GenPermByBit*, and *ComposePerms*. In the protocol *BitVecDec*, the parties input a secret-shared vector $\langle \vec{x} \rangle$, and they get ℓ secret-shared bit vectors $\langle \vec{b_0} \rangle, \cdots, \langle \vec{b}_{\ell-1} \rangle$ as the output, such that $\vec{x}[i] = \sum_{j=0}^{\ell-1} 2^j * \vec{b_j}[i]$. Here, ℓ is the bit length of the ring $\mathbb{Z}_{2^\ell}$ on which the protocol *BitVecDec* is performed. In the protocol *GenPermByBit*, the parties input a secret-shared bit vector $\langle \vec{b} \rangle$, and they get a secret-shared permutation $\langle \vec{\pi} \rangle$ as the output, such that $\vec{\pi}[i]$ represents the index of $\vec{b}[i]$ in ascending order. Note that this ascending order is also stable. In the protocol *ComposePerms*, the parties input two secret-shared permutations $\langle \vec{\alpha} \rangle$ and $\langle \vec{\beta} \rangle$, and get a secret-shared permutation $\langle \vec{\pi} \rangle$

as the output, such that $\langle\vec{\pi}\rangle$ compose the effects of both $\langle\vec{\alpha}\rangle$ and $\langle\vec{\beta}\rangle$, i.e. $ApplyPerm(\langle\vec{\pi}\rangle, \langle\vec{x}\rangle) = ApplyPerm\ (\langle\vec{\beta}\rangle, ApplyPerm(\langle\vec{\alpha}\rangle, \langle\vec{x}\rangle))$.

As is shown in Protocol 1, the protocol *GenPerm* consists of three stages. The first stage (Line 1): the parties use the protocol *BitVecDec* to decompose the input $\langle\vec{x}\rangle$ into ℓ secret-shared bit vectors, $\langle\vec{b_0}\rangle, \cdots, \langle\vec{b}_{\ell-1}\rangle$. The second stage (Line 2): the parties generate a secret-shared permutation $\langle\vec{\pi}\rangle$ from the least significant bit vector $\langle\vec{b_0}\rangle$. The third stage (Line 3-7): the parties update $\langle\vec{\pi}\rangle$ with each remaining secret-shared bit vector. To update $\langle\vec{\pi}\rangle$ with the i-th secret-shared bit vector $\langle\vec{b_i}\rangle$, the parties first apply $\langle\vec{\pi}\rangle$ to $\langle\vec{b_i}\rangle$ to make the i-th bits sorted according to the least significant $i-1$ bits, producing a new secret-shared bit vector $\langle\vec{b'}_i\rangle$ (Line 4). Next, the parties generate a new secret-shared permutation $\langle\vec{\alpha}\rangle$ from $\langle\vec{b'}_i\rangle$ (Line 5), such that $\langle\vec{\alpha}\rangle$ can be used to sort a secret-shared vector that has been sorted according to the least significant $i-1$ bits further according to the i-th bits. Finally, the parties use the protocol *ComposePerms* compose the effect of $\langle\vec{\pi}\rangle$ and $\langle\vec{\alpha}\rangle$ to obtain the updated $\langle\vec{\pi}\rangle$, which can be used to directly sort a secret-shared vector according to the least significant i bits (Line 6).

Besides, we introduce another protocol *UnApplyPerm*, which is also used in our proposed training protocols. In this protocol, the parties input a secret-shared permutation $\langle\vec{\pi}\rangle$ and a secret-shared vector $\langle\vec{x}\rangle$, and they get a secret-shared vector $\langle\vec{z}\rangle$, which is generated by securely reversing the effect of applying $\langle\vec{\pi}\rangle$ to $\langle\vec{x}\rangle$, as the output. For example, given $\langle\vec{\pi}\rangle = [\langle 2\rangle, \langle 3\rangle, \langle 0\rangle, \langle 4\rangle, \langle 1\rangle]$ and $\langle\vec{x}\rangle = [\langle 2\rangle, \langle 3\rangle, \langle 4\rangle, \langle 9\rangle, \langle 9\rangle]$, $\langle\vec{z}\rangle = UnApplyPerm(\langle\vec{\pi}\rangle, \langle\vec{x}\rangle) = [\langle 4\rangle, \langle 9\rangle, \langle 2\rangle, \langle 9\rangle, \langle 3\rangle]$.

Protocol 1 $GenPerm(\langle\vec{x}\rangle)$

Input: A secret-shared vector $\langle\vec{x}\rangle$.
Output: A secret-shared permutation $\langle\vec{\pi}\rangle$.

1: $\langle\vec{b_0}\rangle, \cdots, \langle\vec{b}_{\ell-1}\rangle = BitVecDec(\langle\vec{x}\rangle)$.
2: $\langle\vec{\pi}\rangle = GenPermByBit(\langle\vec{b_0}\rangle)$.
3: **for** $i = 1$ *to* $\ell - 1$ **do**
4: $\quad\langle\vec{b'}_i\rangle = ApplyPerm(\langle\vec{\pi}\rangle, \langle\vec{b_i}\rangle)$.
5: $\quad\langle\vec{\alpha}\rangle = GenPermByBit(\langle\vec{b'}_i\rangle)$.
6: $\quad\langle\vec{\pi}\rangle = ComposePerms(\langle\vec{\pi}\rangle, \langle\vec{\alpha}\rangle)$.
7: **end for**

2.3.5 Group-Wise Protocols Based on RSS. We present the design of group-wise protocols proposed by Hamada et al. [18]. The fundamental data structure of the group-wise protocols consists of a secret-shared element vector $\langle\vec{x}\rangle$ and a secret-shared group flag vector $\langle\vec{g}\rangle$. The secret-shared group flag vector $\langle\vec{g}\rangle$ privately indicates the group boundaries of $\langle\vec{x}\rangle$. Concretely, $\langle\vec{g}[i]\rangle = \langle 1\rangle$ if the i-th element of $\langle\vec{x}\rangle$ is the first element of a group, and $\langle\vec{g}[i]\rangle = \langle 0\rangle$ otherwise. According to this definition, $\langle\vec{g}[0]\rangle$ always equals to $\langle 1\rangle$. For example, $\langle\vec{g}\rangle = [\langle 1\rangle, \langle 0\rangle, \langle 1\rangle, \langle 1\rangle, \langle 0\rangle, \langle 0\rangle]$ and $\langle\vec{x}\rangle = [\langle 4\rangle, \langle 3\rangle, \langle 2\rangle, \langle 8\rangle, \langle 9\rangle, \langle 0\rangle]$ mean that the six elements of $\langle\vec{x}\rangle$ are divided into three groups: $\{\langle 4\rangle, \langle 3\rangle\}$, $\{\langle 2\rangle\}$, $\{\langle 8\rangle, \langle 9\rangle, \langle 0\rangle\}$. With this data structure, the elements of a secret-shared vector $\langle\vec{x}\rangle$ can be divided into several groups without revealing any information about group membership.

The group-wise protocols based on the above data structure include *GroupSum*, *GroupPrefixSum*, and *GroupMax*. In all these protocols, the parties input a secret-shared group flag vector $\langle\vec{g}\rangle$ and a secret-shared element vector $\langle\vec{x}\rangle$, and get a secret-shared vector $\langle\vec{z}\rangle$ as the output. The secret-shared vector $\langle\vec{g}\rangle$, $\langle\vec{x}\rangle$ and $\langle\vec{z}\rangle$ are of the same size. Here, we define $f(i)$ as the first element index of the group that $\vec{x}[i]$ belongs to, and $l(i)$ as the last element index of the group that $\vec{x}[i]$ belongs to. As is shown in Figure 1, the protocol *GroupSum* securely assigns $\vec{z}[i]$ as the sum of the group that $\vec{x}[i]$ belongs to, i.e. $\vec{z}[i] = \sum_{j=f(i)}^{l(i)} \vec{x}[j]$. The protocol *GroupPrefixSum* securely assigns $\vec{z}[i]$ as the prefix sum of the group that $\vec{x}[i]$ belongs to, i.e. $\vec{z}[i] = \sum_{j=f(i)}^{i} \vec{x}[j]$. The protocol *GroupMax* securely assigns $\vec{z}[i]$ as the maximum of the group that $\vec{x}[i]$ belongs to, i.e. $\vec{z}[i] = \max_{j\in[f(i),l(i)]} \vec{x}[j]$. Besides, we introduce a extension version of the protocol *GroupMax*, which are denoted as $GroupMax(\langle\vec{g}\rangle, \langle\vec{x}\rangle, \langle\vec{y}\rangle)$. In this protocol, the parties input a secret-shared group flag vector $\langle\vec{g}\rangle$, a secret-shared vector $\langle\vec{x}\rangle$, and a secret-shared vector $\langle\vec{y}\rangle$, and they get two secret-shared vectors $\langle\vec{z}_1\rangle, \langle\vec{z}_2\rangle$ as the output, where $\vec{z}_1[i] = \vec{x}[k]$ and $\vec{z}_2[i] = \vec{y}[k]$ when $k = \arg\max_{j\in[f(i),l(i)]} \vec{x}[j]$. The secret-shared vectors $\langle\vec{g}\rangle$, $\langle\vec{x}\rangle$, $\langle\vec{y}\rangle$, $\langle\vec{z}_1\rangle$ and $\langle\vec{z}_2\rangle$ are also of the same size.

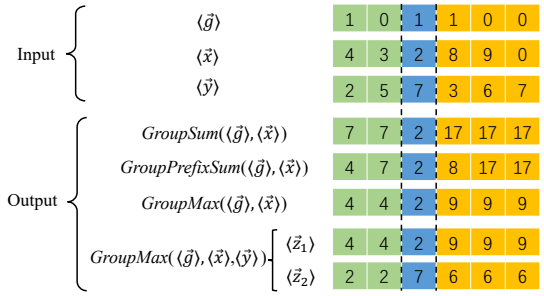

Figure 1: Examples of group-wise protocols. We use both dashed lines and colors to distinctly delineate the group boundaries.

3 OVERVIEW OF ENTS

3.1 Architecture and Security Model

The architecture of Ents involves three parties, each of which could be a corporation, organization, or even individual. They seek to collaboratively train a decision tree on their own data while preserving the privacy of their data. The secure training process of Ents is as follows: (1) the parties securely share their data with each other using RSS, ensuring that each party holds the shares of the entire training dataset. (2) the parties perform secure training protocols on this secret-shared training dataset to obtain the shares of a trained decision tree. (3) the parties have the option to either send the shares of the trained decision tree to a designated party for tree reconstruction or utilize the shares for secure prediction on other samples.

Ents employs a three-party semi-honest security model with an honest majority, which is much more practical and widely used in the field of privacy preserving machine learning [1, 15, 18, 41, 44], due to its efficiency and promising security. In this security model,

the parties mentioned above will correctly execute the training protocols without colluding with each other. Besides, we also present the technical routines in Section 6 for the two-party semi-honest security model with a dishonest majority.

3.2 Data Representation

In our proposed training protocols, we add a subscript M to $\langle x \rangle$, producing $\langle x \rangle_M$, to indicate the size of the ring on which a secret x is shared, where $M = 2$ or 2^k or 2^ℓ ($2 < k < \ell$).

For a training dataset consisting of n samples, each of which consists of m continuous attributes and a label, we represent the dataset using m secret-shared attribute vectors $\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}$ and a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$ for sharing. That is, for the i-th sample, its attributes and label are $\vec{a}_0[i], \cdots, \vec{a}_{m-1}[i]$, and $\vec{y}[i]$.

Besides, we assign each node a node ID (*nid* for short) to reflect the relationship among the nodes. That is, for the node whose *nid* is j in the k-th layer, its left child node is the node whose *nid* is $2 * j + 1$ in the $(k+1)$-th layer. Its right child node is the node whose *nid* is $2 * j + 2$ in the $(k+1)$-th layer.

Furthermore, we leverage a secret-shared sample-node vector $\langle \overrightarrow{spnd}^{(k)} \rangle_{2^k}$ to privately indicate which node the samples belong to in the k-th layer of a decision tree. Concretely, the equation $\overrightarrow{spnd}^{(k)}[i] = j$ indicates the i-th sample belongs to the node whose *nid* is j in the k-th layer.

4 DESIGN OF ENTS

4.1 Protocol for Training a Decision Tree

As is shown in Protocol 2, the protocol *TrainDecisionTree* is used to securely train a decision tree. The parties in this protocol input m secret-shared attribute vectors $\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}$, a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$, and an integer h (representing the desired tree's height). They get $h+1$ secret-shared layers $\langle layer^{(0)} \rangle_{2^k}, \cdots, \langle layer^{(h)} \rangle_{2^k}$ of a trained decision tree as the output. If the k-th layer is an internal layer, the secret-shared layer $\langle layer^{(k)} \rangle_{2^k}$ contains the secret-shared *nid*, split attribute index, and split threshold of each node in the k-th layer. If the k-th layer is a leaf layer, the secret-shared layer $\langle layer^{(k)} \rangle_{2^k}$ contains the secret-shared *nid* and predicted label of each node in the k-th layer.

The protocol *TrainDecisionTree* consists of three stages. (1) The first stage (Line 1-4): the parties generate m secret-shared permutations from the attributes by calling the protocol *GenPerm* (Protocol 1) and initial a secret-shared sample-node vector $\langle \overrightarrow{spnd}^{(0)} \rangle_{2^k}$. These permutations are used to securely sort the training samples to save secure comparison operations. The parties set all elements of $\langle \overrightarrow{spnd}^{(0)} \rangle_{2^k}$ to $\langle 0 \rangle$, since all samples belong to the root node whose *nid* is 0 in the 0-th layer. (2) The second stage (Line 5-10): the parties securely train each internal layer of the decision tree, compute the secret-shared sample-node vector for the next layer, and update the pre-generated permutations. To securely train an internal layer, the parties call the protocol *TrainInternalLayer* (Protocol 4), which outputs a secret-shared layer $\langle layer^{(k)} \rangle_{2^k}$, a secret-shared sample-attribute vector $\langle \overrightarrow{spat}^{(k)} \rangle_{2^k}$, and a secret-shared sample-threshold vector $\langle \overrightarrow{spth}^{(k)} \rangle_{2^k}$ (Line 6). $\overrightarrow{spat}^{(k)}$ and $\overrightarrow{spth}^{(k)}$ contain the split attribute index and split threshold of the node to which the i-th sample belongs in the k-th layer, respectively. For example, the equations $\overrightarrow{spat}^{(2)}[0] = 1$ and $\overrightarrow{spth}^{(2)}[0] = 5$ represents that the split attribute index and split threshold of the node to which the 0-th sample belongs in the 2-th layer are 1 and 5, respectively. The parties then obtain the secret-shared comparison results $\langle \vec{b} \rangle_{2^k}$ between the samples and the thresholds by calling the protocol *TestSamples* (Protocol 8 in Appendix B) (Line 7). These secret-shared comparison results are used to compute the secret-shared vector $\langle \overrightarrow{spnd}^{(k+1)} \rangle_{2^k}$, and update the pre-generated secret-shared permutations $\langle \vec{\pi}_0 \rangle_{2^k}, \cdots, \langle \vec{\pi}_{m-1} \rangle_{2^k}$ (Line 8-9). The idea for computing $\langle \overrightarrow{spnd}^{(k+1)} \rangle_{2^k}$ is as follows: if a sample in the k-th layer belongs to a node whose *nid* is j and has a comparison result of '0', it will be assigned to the left child node (*nid* is $2 * j + 1$) in the $(k+1)$-th layer. Conversely, if the result is '1', the sample is assigned to the right child node (*nid* is $2 * j + 2$) in the $(k+1)$-th layer. To update the pre-generated permutations, the parties call the protocol *UpdatePerms* (Protocol 3). (3) The third stage (Line 11): the parties securely train the leaf layer by calling the protocol *TrainLeafLayer* (Protocol 9 in Appendix B).

Protocol 2 *TrainDecisionTree*

Input: m secret-shared attribute vectors, $\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}$, a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$, and an integet h.

Output: $h+1$ secret-shared layers,$\langle layer^{(0)} \rangle_{2^k}, \cdots, \langle layer^{(h)} \rangle_{2^k}$, of a decision tree.

1: **for** each $i \in [0, m-1]$ in parallel **do**
2: $\quad \langle \vec{\pi}_i \rangle_{2^k} = GenPerm\ (\langle \vec{a}_i \rangle_{2^k})$.
3: **end for**
4: $\langle \overrightarrow{spnd}^{(0)}[j] \rangle_{2^k} = \langle 0 \rangle$ for each $j \in [0, n-1]$.
5: **for** each $k \in [0, h-1]$ **do**
6: $\quad \langle layer^{(k)} \rangle_{2^k}$, $\langle \overrightarrow{spat}^{(k)} \rangle_{2^k}$, $\langle \overrightarrow{spth}^{(k)} \rangle_{2^k} = TrainInternalLayer($ k, $\langle \overrightarrow{spnd}^{(k)} \rangle_{2^k}$, $\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}$, $\langle \vec{y} \rangle_{2^k}$, $\langle \vec{\pi}_0 \rangle_{2^k}, \cdots,$ $\langle \vec{\pi}_{m-1} \rangle_{2^k})$.
7: $\quad \langle \vec{b} \rangle_{2^k} = TestSamples(\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}, \langle \overrightarrow{spat}^{(k)} \rangle_{2^k},$ $\langle \overrightarrow{spth}^{(k)} \rangle_{2^k})$.
8: $\quad \langle \overrightarrow{spnd}^{(k+1)} \rangle_{2^k} = 2 * \langle \overrightarrow{spnd}^{(k)} \rangle_{2^k} + 1 + \langle \vec{b} \rangle_{2^k}$.
9: $\quad \langle \vec{\pi}_0 \rangle_{2^k}, \cdots, \langle \vec{\pi}_{m-1} \rangle_{2^k} = UpdatePerms\ (\langle \vec{b} \rangle_{2^k}, \langle \vec{\pi}_0 \rangle_{2^k}, \cdots,$ $\langle \vec{\pi}_{m-1} \rangle_{2^k})$.
10: **end for**
11: $\langle layer^{(h)} \rangle_{2^k} = TrainLeafLayer(h, \langle \vec{\pi}_0 \rangle_{2^k}, \langle \overrightarrow{spnd}^{(h)} \rangle_{2^k}, \langle \vec{y} \rangle_{2^k})$.

To optimize the communication overhead for securely training a decision tree, we propose two communication optimizations: One is based on the secure radix sort protocols [10] and mainly implemented in our proposed protocols *UpdatePerms* (Protocol 3 in Section 4.2.3), *TrainInternalLayer* (Protocol 4 in Section 4.2.4), and *TrainLeafLayer* (Protocol 9 in Appendix B). The other is based on a novel share conversion protocol proposed by us and mainly implemented in our proposed protocols *ComputeModifiedGini* (Protocol 5 in Section 4.3.2) and *ConvertShare* (Protocol 6 in Section 4.3.4).

Note that we present an example to show the key steps of the training process in Figure 3 in Appendix C for understanding.

4.2 Communication Optimization Based on Secure Radix Sort Protocols

4.2.1 Brief Overview of Existing Methods for Secure Dataset Splitting. Securely splitting a dataset with continuous attributes usually

requires numerous secure comparison operations, leading to huge communication overhead. To resolve this issue, Abspoel et al.[1] propose securely generating secret-shared permutations from the secret-shared continuous attribute vectors at the beginning of the training process. These pre-generated secret-shared permutations are used to securely sort the training samples according to the attribute vectors, so that lots of secure comparison operations can be saved. However, to hide the splitting information of the training dataset, their method requires securely training each node with a padded dataset of the same size as the entire training dataset. The secure training process with padded datasets results in huge communication overhead that grows exponentially as the height of the decision tree increases.

To achieve linear growth in communication overhead as the height of the decision tree increases, Hamada et al. [18] propose a series of group-wise protocols. These protocols treat the samples of each node as a group and enable securely training all nodes of one layer simultaneously with non-padded datasets, thereby avoiding the exponential growth in communication overhead. However, the group-wise protocols are incompatible with pre-generated secret-shared permutations. Specifically, to use the group-wise protocols, the samples belonging to the same node must be stored consecutively. Nevertheless, after using the pre-generated secret-shared permutations to securely sort the samples, the samples belonging to the same node will no longer be stored consecutively. To resolve this issue, Hamada et al. [18] propose securely sorting the training samples first according to the attribute vectors and then according to the sample-node vector in each layer. Due to the stability of the sort, this method can consecutively store samples belonging to the same node and securely sort the samples according to the attribute vectors within each node. Nevertheless, this method requires repeatedly generating secret-shared permutations from each attribute and the sample-node vector for training each internal layer of the decision tree. The repeated generation process also results in significant communication overhead.

4.2.2 Main Idea. To optimize the communication overhead, we employ the secure radix sort protocols [10] to securely update the pre-generated secret-shared permutations, so that the pre-generated secret-shared permutations become compatible with the group-wise protocols after the updating process. More specifically, once updated, the pre-generated secret-shared permutations can be employed to consecutively store samples belonging to the same node, and securely sort these samples according to their attribute vectors within each node. This method combines the advantages of the methods proposed by Abspoel et al. [1] and Hamada et al. [18]: this method only requires securely generating the secret-shared permutations once, and the communication overhead of this method for training a decision tree only grows linearly as the height of the decision tree increases.

The principle behind securely updating the pre-generated secret-shared permutations based on the secure radix sort protocols [10] is as follows: the comparison result between a sample and the split threshold of a node $\mathcal{N}$ that the sample belongs to determines which node the sample should be assigned to. A comparison result of '0' indicates this sample should be assigned to the left child node of $\mathcal{N}$, while a comparison result of '1' indicates this sample should be assigned to the right child node of $\mathcal{N}$. Thus, if the samples are sorted according to the comparison result vector, the samples belonging to the same node will be stored consecutively. Furthermore, if the pre-generated permutations are updated with the comparison result vector, due to the stability of radix sort, the updated permutations can then be used to store the samples belonging to the same node consecutively, and meanwhile, sort the samples according to their attribute vectors within each node.

Protocol 3 *UpdatePerms*

Input: A secret-shared comparison result vector $\langle \vec{b} \rangle_{2^k}$ and m secret-shared permutations $\langle \vec{\pi}_0 \rangle_{2^k}, \cdots, \langle \vec{\pi}_{m-1} \rangle_{2^k}$.
Output: m secret-shared permutations $\langle \vec{\pi}_0 \rangle_{2^k}, \cdots, \langle \vec{\pi}_{m-1} \rangle_{2^k}$.

1: **for** each $i \in [0, m-1]$ in parallel **do**
2: $\quad \langle \vec{b'}_i \rangle_{2^k} = ApplyPerm(\langle \pi_i \rangle_{2^k}, \langle \vec{b} \rangle_{2^k})$.
3: $\quad \langle \vec{\alpha} \rangle_{2^k} = GenPermByBit(\langle \vec{b'}_i \rangle_{2^k})$.
4: $\quad \langle \vec{\pi}_i \rangle_{2^k} = ComposePerms(\langle \vec{\pi}_i \rangle_{2^k}, \langle \vec{\alpha} \rangle_{2^k})$.
5: **end for**

4.2.3 Protocol for Updating Permutations. As is shown in Protocol 3, the protocol *UpdatePerms* securely updates the pre-generated permutations with the comparison result vector that contains the comparison results between the samples and the thresholds. The parties in this protocol input a secret-shared comparison result vector $\langle \vec{b} \rangle_{2^k}$ and m secret-shared permutations $\langle \vec{\pi}_0 \rangle_{2^k}, \cdots, \langle \vec{\pi}_{m-1} \rangle_{2^k}$. They get m updated secret-shared permutations as the output.

The protocol *UpdatePerms* follows the updating step (Line 4-6) of the protocol *GenPerm* (Protocol 1) to update each secret-shared permutations $\langle \vec{\pi}_i \rangle_{2^k}$: the parties first apply $\langle \vec{\pi}_i \rangle_{2^k}$ to $\langle \vec{b} \rangle_{2^k}$ to generate a new secret-shared vector $\langle \vec{b'}_i \rangle_{2^k}$ (Line 2). Next, the parties generate a secret-shared permutation $\langle \vec{\alpha} \rangle_{2^k}$ from $\langle \vec{b'}_i \rangle_{2^k}$ by calling the protocol *GenPermByBit* (introduced in Section 2.3.4) (Line 3), such that $\langle \vec{\alpha} \rangle_{2^k}$ can be used to sort a secret-shared vector that has been sorted with $\langle \vec{\pi}_i \rangle_{2^k}$ further according to $\langle \vec{b} \rangle_{2^k}$. Finally, the parties compose the effect of $\langle \vec{\pi}_i \rangle_{2^k}$ and $\langle \vec{\alpha} \rangle_{2^k}$ by calling the protocol *ComposePerms* (introduced in Section 2.3.4), resulting in an updated $\langle \vec{\pi}_i \rangle_{2^k}$ (Line 4).

4.2.4 Protocol for Training Internal Layer. As is shown in Protocol 4, the protocol *TrainInternalLayer* is used to securely train an internal layer of a decision tree with the secret-shared permutations. The parties in this protocol input an integer k, representing the height of the internal layer, a secret-shared sample-node vector $\langle \overrightarrow{spnd} \rangle_{2^k}$, m secret-shared attribute vectors $\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}$, a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$, and m secret-shared permutations $\langle \vec{\pi}_0 \rangle_{2^k}, \cdots, \langle \vec{\pi}_{m-1} \rangle_{2^k}$. They get a secret-shared layer $\langle layer \rangle_{2^k}$, a secret-shared sample-attribute vector $\langle \overrightarrow{spat} \rangle_{2^k}$, and a secret-shared sample-threshold vector $\langle \overrightarrow{spth} \rangle_{2^k}$ as the output.

The protocol *TrainInternalLayer* consists of four stages. (1) The first stage (Line 1-3): the parties compute a secret-shared group flag vector $\langle \vec{g} \rangle_{2^k}$ that privately indicates the boundaries of the nodes. To compute $\langle \vec{g} \rangle_{2^k}$, the parties first apply one of the permutations (without loss of generality, $\langle \vec{\pi}_0 \rangle_{2^k}$ is chosen) to $\langle \overrightarrow{spnd} \rangle_{2^k}$ to obtain a new secret-shared sample-node vector $\langle \overrightarrow{spnd'} \rangle_{2^k}$ (Line 1). This process ensures that the elements with the same values in $\langle \overrightarrow{spnd'} \rangle_{2^k}$

Protocol 4 *TrainInternalLayer*

Input: An integer k that represents the height of the internal layer, a secret-shared sample-node vector $\langle \overrightarrow{spnd} \rangle_{2^k}$, m secret-shared attribute vectors, $\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}$, a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$, and m secret-shared permutations $\langle \vec{\pi}_0 \rangle_{2^k}, \cdots, \langle \vec{\pi}_{m-1} \rangle_{2^k}$.
Output: a secret-shared layer $\langle layer \rangle_{2^k}$, a secret-shared sample-attribute vector $\langle \overrightarrow{spat} \rangle_{2^k}$, and a secret-shared sample-threshold vector $\langle \overrightarrow{spth} \rangle_{2^k}$.

1: $\langle \overrightarrow{spnd'} \rangle_{2^k} = ApplyPerm(\langle \vec{\pi}_0 \rangle_{2^k}, \langle \overrightarrow{spnd} \rangle_{2^k})$.
2: $\langle \vec{g}[0] \rangle_{2^k} = \langle 1 \rangle_{2^k}$.
3: $\langle \vec{g}[j] \rangle_{2^k} = (\langle \overrightarrow{spnd'}[j-1] \rangle_{2^k} \neq \langle \overrightarrow{spnd'}[j] \rangle_{2^k})$ for each $j \in [1, n-1]$.
4: **for** each $i \in [0, m-1]$ in parallel **do**
5: $\langle \vec{y'} \rangle_{2^k} = ApplyPerm(\langle \vec{\pi}_i \rangle_{2^k}, \langle \vec{y} \rangle_{2^k})$.
6: $\langle \vec{a'}_i \rangle_{2^k} = ApplyPerm(\langle \vec{\pi}_i \rangle_{2^k}, \langle \vec{a}_i \rangle_{2^k})$.
7: $\langle \overrightarrow{spth_i} \rangle_{2^k}, \langle \overrightarrow{gini_i} \rangle_{2^k} = AttributeWiseSplitSelection$ $(\langle \vec{g} \rangle_{2^k}, \langle \vec{a'}_i \rangle_{2^k}, \langle \vec{y'} \rangle_{2^k})$.
8: **end for**
9: **for** each $j \in [0, n-1]$ in parallel **do**
10: $\langle \overrightarrow{ginilist} \rangle_{2^k} = [\langle \overrightarrow{gini}_0[j] \rangle_{2^k}, \cdots, \langle \overrightarrow{gini}_{m-1}[j] \rangle_{2^k}]$.
11: $\langle \overrightarrow{aidxlist} \rangle_{2^k} = [\langle 0 \rangle_{2^k}, \cdots, \langle m-1 \rangle_{2^k}]$.
12: $\langle \overrightarrow{stlist} \rangle_{2^k} = [\langle \overrightarrow{spth}_0[j] \rangle_{2^k}, \cdots, \langle \overrightarrow{spth}_{m-1}[j] \rangle_{2^k}]$.
13: $\langle \overrightarrow{spat}[j] \rangle_{2^k} = VectMax(\langle \overrightarrow{ginilist} \rangle_{2^k}, \langle \overrightarrow{aidxlist} \rangle_{2^k})$.
14: $\langle \overrightarrow{spth}[j] \rangle_{2^k} = VectMax(\langle \overrightarrow{ginilist} \rangle_{2^k}, \langle \overrightarrow{stlist} \rangle_{2^k})$.
15: **end for**
16: $\langle layer \rangle_{2^k} = FormatLayer\ (k, \langle \vec{g} \rangle_{2^k}, \langle \overrightarrow{spnd'} \rangle_{2^k}, \langle \overrightarrow{spat} \rangle_{2^k}, \langle \overrightarrow{spth} \rangle_{2^k})$.
17: $\langle \overrightarrow{spat} \rangle_{2^k} = UnApplyPerm(\langle \vec{\pi}_0 \rangle_{2^k}, \langle \overrightarrow{spat} \rangle_{2^k})$.
18: $\langle \overrightarrow{spth} \rangle_{2^k} = UnApplyPerm(\langle \vec{\pi}_0 \rangle_{2^k}, \langle \overrightarrow{spth} \rangle_{2^k})$.

are stored consecutively. Thus, if $\overrightarrow{spnd'}[j] \neq \overrightarrow{spnd'}[j-1]$, the j-th sample should be the first sample of a node. Then, the parties set $\langle \vec{g}[0] \rangle_{2^k}$ to $\langle 1 \rangle$ and compute the remaining elements of $\langle \vec{g} \rangle_{2^k}$ based on $\langle \overrightarrow{spnd'} \rangle_{2^k}$ (Line 2-3). (2) The second stage (Line 4-8): the parties securely compute the split thresholds with each attribute and the corresponding modified Gini impurities. To achieve this, the parties first apply each secret-shared permutation $\langle \vec{\pi}_i \rangle_{2^k}$ to $\langle \vec{y} \rangle_{2^k}$ and $\langle \vec{a}_i \rangle_{2^k}$ to generate a new secret-shared label vector $\langle \vec{y'} \rangle_{2^k}$ and a new secret-shared attribute vector $\langle \vec{a'}_i \rangle_{2^k}$ (Line 5-6). This process makes the elements of $\langle \vec{y'} \rangle_{2^k}$ and $\langle \vec{a'}_i \rangle_{2^k}$ belonging to the same node stored consecutively and sorted according to the corresponding attribute vector within each node, such that the group-wise protocols [18] can be applied on $\langle \vec{y'} \rangle_{2^k}$ and $\langle \vec{a'}_i \rangle_{2^k}$. Subsequently, the parties call the protocol *AttributeWiseSplitSelection* (Protocol 7 in Appendix B) to obtain a secret-shared sample-threshold vector $\langle \overrightarrow{spth_i} \rangle_{2^k}$ and a secret-shared Gini impurity vector $\langle \overrightarrow{gini_i} \rangle_{2^k}$ (Line 7). $\overrightarrow{spth_i}[j]$ is the split threshold of the node to which the j-sample belongs and is computed with only the i-th attribute. $\overrightarrow{gini_i}[j]$ is the modified Gini impurity for the split point $(a_i, \overrightarrow{spth_i}[j])$. (3) The third stage (Line 9-15): the parties securely find the split attribute index and the split threshold with a maximum modified Gini impurity across different attributes. This is achieved by calling the protocol *VectMax* (introduced in Section 2.3.3). (4) The fourth stage (Line 16-18): the parties first compute the secret-shared layer $\langle layer \rangle_{2^k}$ by calling the protocol *FormatLayer* (Protocol 10 in Appendix B) (Line 16). The protocol *FormatLayer* is used to remove the redundant values from the input vectors to leave only one *nid*, split attribute index, and split threshold for each node. Then the parties reverse the effect of $\langle \vec{\pi}_0 \rangle_{2^k}$ on the secret-shared sample-attribute vector $\langle \overrightarrow{spat} \rangle_{2^k}$ and sample-threshold vector $\langle \overrightarrow{spth} \rangle_{2^k}$ to align their element positions with the original secret-shared attribute vectors (Line 17-18). So that, $\langle \overrightarrow{spat} \rangle_{2^k}$ and $\langle \overrightarrow{spth} \rangle_{2^k}$ can be used in the comparison between the samples and the split thresholds (Line 7 of Protocol 2).

4.3 Communication Optimization Based on Share Conversion Protocol

4.3.1 Brief Overview of Existing Methods for Secure Splitting Criterion Computation. The existing multi-party training frameworks [1, 15, 18] for decision trees perform almost all computations on a large ring to accommodate the secure computations for the splitting criterion, e.g. the modified Gini impurity. The secure computations for the splitting criterion usually involve several sequential bit-length expansion operations, such as secure multiplication and division operations. These operations usually produce intermediate or final results that require twice as many bits for representation compared to their inputs. For instance, a secure multiplication operation with two 10-bit numbers as input produces a 20-bit result. Performing such operations sequentially several times requires a large ring to represent the intermediate or final results. For example, the framework proposed by Abspoel et al. [1] requires a ring with bit length at least $5\lceil \log \frac{n}{2} \rceil$ to accommodate the secure computation for the modified Gini impurity, where n is the number of samples in the training dataset. Performing almost all computations on a large ring incurs huge communication overhead to the existing multi-party training frameworks.

4.3.2 Main Idea. To optimize the communication overhead incurred by performing almost all computations on a large ring, we propose an efficient share conversion protocol to convert shares between a small ring and a large ring. We observe that excluding the secure computations for the splitting criterion, other computations in the training process only involve bit-length invariant operations, such as secure addition and comparison operations, which cause at most a single-bit increase and can be performed on a small ring whose bit length is just two bits large than $\lceil \log n \rceil$ (one additional bit for representing the sign and the other additional bit for avoiding overflow when performing secure addition operations). With our proposed share conversion protocol, the parties in Ents can convert shares to a large ring when performing the bit-length extension operations and then convert the shares back to a small ring once these operations are completed. As a result, the communication overhead of the bit-length invariant operations can be significantly reduced.

4.3.3 Protocol for Computing Modified Gini Impurity. As is shown in Protocol 5, the protocol *ComputeModifiedGini* is based on our proposed share conversion protocol *ConvertShare* (Protocol 6) to convert shares to a large ring $\mathbb{Z}_{2^\ell}$ for accommodating the bit-length expansion operations. Besides, the computed modified Gini impurity vector is finally converted back to a small ring $\mathbb{Z}_{2^k}$ to save the communication overhead of the further bit-length invariant operations. The parties in this protocol input a secret-shared group flag vector $\langle \vec{g} \rangle_{2^k}$ and a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$, whose elements

belonging to the same nodes are stored consecutively and sorted according to an attribute vector $\vec{a}_i$ within each node. The parties get a secret-shared modified Gini impurity vector $\langle \overrightarrow{gini} \rangle_{2^\kappa}$ as the output, where $\overrightarrow{gini}[j]$ is the modified Gini impurity of the split point (a_i, t_j), where $t_j = (\vec{a}_i[j] + \vec{a}_i[j+1])/2$ for each $j \in [0, n-2]$ and $t_{n-1} = \vec{a}_i[n-1]$.

Protocol 5 *ComputeModifiedGini*

Input: A secret-shared group flag vector $\langle \vec{g} \rangle_{2^\kappa}$, and a secret-shared label vectors $\langle \vec{y} \rangle_{2^\kappa}$.

Output: A secret-shared modified Gini impurity vector $\langle \overrightarrow{gini} \rangle_{2^\kappa}$.

1: $\langle \overrightarrow{one}[j] \rangle_{2^\kappa} = \langle 1 \rangle_{2^\kappa}$ for each $j \in [0, n-1]$.
2: $\langle \overrightarrow{ts} \rangle_{2^\kappa} = GroupSum(\langle \vec{g} \rangle_{2^\kappa}, \langle \overrightarrow{one} \rangle_{2^\kappa})$.
3: $\langle \overrightarrow{ps} \rangle_{2^\kappa} = GroupPrefixSum(\langle \vec{g} \rangle_{2^\kappa}, \langle \overrightarrow{one} \rangle_{2^\kappa})$.
4: $\langle \overrightarrow{ss} \rangle_{2^\kappa} = \langle \overrightarrow{ts} \rangle_{2^\kappa} - \langle \overrightarrow{ps} \rangle_{2^\kappa}$.
5: $\langle \overrightarrow{ps} \rangle_{2^\ell} = ConvertShare(\langle \overrightarrow{ps} \rangle_{2^\kappa}, \kappa, \ell)$.
6: $\langle \overrightarrow{ss} \rangle_{2^\ell} = ConvertShare(\langle \overrightarrow{ss} \rangle_{2^\kappa}, \kappa, \ell)$.
7: **for** each $l \in [0, v-1]$ in parallel **do**
8: $\quad \langle \vec{y}_l \rangle_{2^\kappa} = \langle \vec{y} \rangle_{2^\kappa} \stackrel{?}{=} l$.
9: $\quad \langle \vec{c}_l \rangle_{2^\kappa} = GroupSum(\langle \vec{g} \rangle_{2^\kappa}, \langle \vec{y}_l \rangle_{2^\kappa})$.
10: $\quad \langle \overrightarrow{pre}_l \rangle_{2^\kappa} = GroupPrefixSum(\langle \vec{g} \rangle_{2^\kappa}, \langle \vec{y}_l \rangle_{2^\kappa})$.
11: $\quad \langle \overrightarrow{suf}_l \rangle_{2^\kappa} = \langle \vec{c}_l \rangle_{2^\kappa} - \langle \overrightarrow{pre}_l \rangle_{2^\kappa}$.
12: $\quad \langle \overrightarrow{pre}_l \rangle_{2^\ell} = ConvertShare(\langle \overrightarrow{pre}_l \rangle_{2^\kappa}, \kappa, \ell)$.
13: $\quad \langle \overrightarrow{suf}_l \rangle_{2^\ell} = ConvertShare(\langle \overrightarrow{suf}_l \rangle_{2^\kappa}, \kappa, \ell)$.
14: $\quad \langle \overrightarrow{presq}_l \rangle_{2^\ell} = \langle \overrightarrow{pre}_l \rangle_{2^\ell} * \langle \overrightarrow{pre}_l \rangle_{2^\ell}$.
15: $\quad \langle \overrightarrow{sufsq}_l \rangle_{2^\ell} = \langle \overrightarrow{suf}_l \rangle_{2^\ell} * \langle \overrightarrow{suf}_l \rangle_{2^\ell}$.
16: **end for**
17: $\langle \overrightarrow{presqs} \rangle_{2^\ell} = \sum_{l=0}^{v-1} \langle \overrightarrow{presq}_l \rangle_{2^\ell}$.
18: $\langle \overrightarrow{sufsqs} \rangle_{2^\ell} = \sum_{l=0}^{v-1} \langle \overrightarrow{sufsq}_l \rangle_{2^\ell}$.
19: $\langle \overrightarrow{gini} \rangle_{2^\ell} = \langle \overrightarrow{presqs} \rangle_{2^\ell} / \langle \overrightarrow{ps} \rangle_{2^\ell} + \langle \overrightarrow{sufsqs} \rangle_{2^\ell} / \langle \overrightarrow{ss} \rangle_{2^\ell}$.
20: **if** $f + \lceil \log n \rceil > \kappa - 1$ **then**
21: $\quad \langle \overrightarrow{gini} \rangle_{2^\ell} = Trunc(\langle \overrightarrow{gini} \rangle_{2^\ell}, f + \lceil \log n \rceil - (\kappa - 1))$.
22: **end if**
23: $\langle \overrightarrow{gini} \rangle_{2^\kappa} = ConvertShare(\langle \overrightarrow{gini} \rangle_{2^\ell}, \ell, \kappa)$.

The protocol *ComputeModifiedGini* consists of three stages. (1) The first stage (Line 1-6): the parties securely compute the values of $|\mathcal{D}^{node}_{a_i<t_j}|$ and $|\mathcal{D}^{node}_{a_i \geq t_j}|$ for each split point (a_i, t_j) simultaneously. To do this, the parties first create a secret-shared one vector $\langle \overrightarrow{one} \rangle_{2^\kappa}$ (Line 1), and compute the secret-shared group-wise prefix sum $\langle \overrightarrow{ps} \rangle_{2^\kappa}$ and group-wise suffix sum $\langle \overrightarrow{ss} \rangle_{2^\kappa}$ of $\langle \overrightarrow{one} \rangle_{2^\kappa}$(Line 2-4). $\overrightarrow{ps}$ and $\overrightarrow{ss}$ contain all the values of $|\mathcal{D}^{node}_{a_i<t_j}|$ and $|\mathcal{D}^{node}_{a_i \geq t_j}|$, because the samples have been sorted according to the attribute vector $\vec{a}_i$. The shares of $\overrightarrow{ps}$ and $\overrightarrow{ss}$ are then converted to a large ring $\mathbb{Z}_{2^\ell}$ (Line 5-6) to accommodate the secure division operation. (2) The second stage (Line 7-18): the parties securely compute the values of $\sum_{l=0}^{v-1} |\mathcal{D}^{node}_{a_i<t_j \wedge y=l}|^2$ and $\sum_{l=0}^{v-1} |\mathcal{D}^{node}_{a_i \geq t_j \wedge y=l}|^2$ for each split point (a_i, t_j) simultaneously. To do this, the parties first compute a secret-shared vector $\langle \vec{y}_l \rangle_{2^\kappa}$ that privately indicates whether the samples' labels are l (Line 8). The parties then compute the corresponding secret-shared group-wise prefix sum $\langle \overrightarrow{pre}_l \rangle_{2^\kappa}$ and group-wise suffix sum $\langle \overrightarrow{suf}_l \rangle_{2^\kappa}$ for the label l (Line 9-11). The shares of these values are then converted to the large ring $\mathbb{Z}_{2^\ell}$ for computing their squared values $\langle \overrightarrow{presq}_l \rangle_{2^\ell}$ and $\langle \overrightarrow{sufsq}_l \rangle_{2^\ell}$ (Line 12-15). Subsequently, the parties sum up the squared values $\langle \overrightarrow{presq}_l \rangle_{2^\ell}$ and $\langle \overrightarrow{sufsq}_l \rangle_{2^\ell}$ of each label (Line 17-18). (3) The third stage (Line 19-23): the parties first compute the secret-share modified Gini impurity vector $\langle \overrightarrow{gini} \rangle_{2^\ell}$, then truncate $\langle \overrightarrow{gini} \rangle_{2^\ell}$ to leave the most significant $\kappa - 1$ bits, and finally convert the truncated results to a small ring $\mathbb{Z}_{2^\kappa}$. Note that before truncation, a modified Gini impurity is a value in $[1, n)$ and has at most $\lceil \log n \rceil + f$ bits, where f is the bit length to represent the decimal part. Thus, only if $\lceil \log n \rceil + f > \kappa - 1$, does the truncation operation need to be performed.

4.3.4 Design of Share Conversion Protocol. To efficiently and accurately perform the protocol *ComputeModifiedGini*, a share conversion protocol that is both efficient and correct is necessary. Though several share conversion protocols have been proposed [4, 17, 20, 24, 32], these existing protocols either lack efficiency or have a high failure probability. Therefore, in this section, we present an efficient share conversion protocol, *ConvertShare*, which only requires an online communication size of $4\ell + 4$ bits in a single communication round and ensures correctness.

Note that converting the shares of a secret x ($x \in [0, 2^{\kappa-1})$) from a large ring $\mathbb{Z}_{2^\ell}$ to a small ring $\mathbb{Z}_{2^\kappa}$ can be achieved just by a local modular operation [40] [2]. Therefore, we only present the principle and protocol of converting the shares from $\mathbb{Z}_{2^\kappa}$ to $\mathbb{Z}_{2^\ell}$ in this paper.

To convert the shares of a secret x from a small ring $\mathbb{Z}_{2^\kappa}$ to a large ring $\mathbb{Z}_{2^\ell}$, the key step is to compute the shares of d_0, d_1 and $ovfl$ on $\mathbb{Z}_{2^\ell}$ (i.e. $\langle d_0 \rangle_{2^\ell}$,$\langle d_1 \rangle_{2^\ell}$, and $\langle ovfl \rangle_{2^\ell}$), where $d_0 = (\llbracket x \rrbracket^0_{2^\kappa} + \llbracket x \rrbracket^1_{2^\kappa}) \mod 2^\kappa$, $d_1 = \llbracket x \rrbracket^2_{2^\kappa}$, and $ovfl = d_0 + d_1 - x$. After obtaining $\langle d_0 \rangle_{2^\ell}$,$\langle d_1 \rangle_{2^\ell}$, and $\langle ovfl \rangle_{2^\ell}$, the parties can compute the shares of x on $\mathbb{Z}_{2^\ell}$ using $\langle x \rangle_{2^\ell} = \langle d_0 \rangle_{2^\ell} + \langle d_1 \rangle_{2^\ell} - \langle ovfl \rangle_{2^\ell}$.

Protocol 6 *ConvertShare*

Input: $\langle x \rangle_{2^\kappa}$ ($x \in [0, 2^{\kappa-1})$, i.e. P_i inputs $\langle x \rangle^i_{2^\kappa} = (\llbracket x \rrbracket^i_{2^\kappa}, \llbracket x \rrbracket^{i+1}_{2^\kappa})$, and two public integers κ and ℓ ($\kappa < \ell$).

Output: $\langle x' \rangle_{2^\ell}$, i.e. P_i gets $\langle x' \rangle^i_{2^\ell} = (\llbracket x' \rrbracket^i_{2^\ell}, \llbracket x' \rrbracket^{i+1}_{2^\ell})$, with $x' = x$.

Offline: P_0, P_1, P_2 generate dabit together, so P_i holds $\llbracket r \rrbracket^i_2$, $\llbracket r \rrbracket^{i+1}_2$, $\llbracket r \rrbracket^i_{2^\ell}$ and $\llbracket r \rrbracket^{i+1}_{2^\ell}$ ($r = 0$ *or* 1).

1: P_0 locally computes $d_0 = (\llbracket x \rrbracket^0_{2^\kappa} + \llbracket x \rrbracket^1_{2^\kappa}) \ mod \ 2^\kappa$. P_1 locally computes $d_1 = \llbracket x \rrbracket^2_{2^\kappa}$.
2: P_0 locally computes $truncd_0 = d_0 \gg (\kappa - 1)$. P_1 locally computes $truncd_1 = (-((-d_1) \gg \kappa - 1))$.
3: P_0 shares d_0 and $truncd_0$ on $\mathbf{Z}_{2^\ell}$. P_1 shares d_1 and $truncd_1$ on $\mathbf{Z}_{2^\ell}$.
4: $\langle d \rangle_{2^\ell} = \langle d_0 \rangle_{2^\ell} + \langle d_1 \rangle_{2^\ell}$.
5: $\langle truncsum \rangle_{2^\ell} = \langle truncd_0 \rangle_{2^\ell} + \langle truncd_1 \rangle_{2^\ell}$.
6: P_0 locally computes $b_0 = (truncd_0 \wedge 1) \oplus \llbracket r \rrbracket^0_2 \oplus \llbracket r \rrbracket^1_2$. P_1 locally computes $b_1 = (truncd_1 \wedge 1) \oplus \llbracket r \rrbracket^2_2$.
7: P_0 sends b_0 to P_1 and P_2. P_1 sends b_1 to P_0 and P_2.
8: The parties all locally compute $b = b_0 \oplus b_1$.
9: $\langle bit \rangle_{2^\ell} = b + \langle r \rangle_{2^\ell} - 2 * b * \langle r \rangle_{2^\ell}$.
10: $\langle ovfl \rangle_{2^\ell} = (\langle truncsum \rangle_{2^\ell} - \langle bit \rangle_{2^\ell}) * 2^{\kappa-1}$.
11: $\langle x' \rangle_{2^\ell} = \langle d \rangle_{2^\ell} - \langle ovfl \rangle_{2^\ell}$.

d_0 can be computed and shared by P_0 ,since both $\llbracket x \rrbracket^0_{2^\kappa}$ and $\llbracket x \rrbracket^1_{2^\kappa}$ are held by P_0. d_1 can be computed and shared by P_1, since $\llbracket x \rrbracket^2_{2^\kappa}$ are held by P_1. Furthermore, the computation of $\langle ovfl \rangle_{2^\ell}$ is

[2]Converting the shares of a secret x ($x \in [0, 2^{\kappa-1})$) from a large ring $\mathbb{Z}_{2^\ell}$ to a small ring $\mathbb{Z}_{2^\kappa}$ is referred to as 'Reduce' in the study [40].

based on the following observation: all the secrets whose shares require conversion in the protocol *ComputeModifiedGini* (Protocol 5) are positive, i.e. the secrets belong to the range $[0, 2^{k-1})$. Additionally, $ovfl = 0$ *or* 2^k, since $ovfl = (d_0+d_1-x)$ and $x = (d_0+d_1) \bmod 2^k$. Hence, to obtain $\langle ovfl \rangle_{2^\ell}$, the parties can first securely truncate the least significant $k-1$ bits of $d_0 + d_1$, and then multiply the secret-shared truncated result with 2^{k-1}.

To securely truncate the least significant $k-1$ bits of $d_0 + d_1$, we utilize Theorem 1, which is a variant of the Trunc Theorem proposed by Zhou et al. [48]. According to Theorem 1, truncating $k-1$ (i.e. $c = k-1$) bits from d_0 and d_1 respectively may lead to one positive bit error in the truncated sum $((d_0 \gg c) + (-((-d_1) \gg c)))$ compared to the correct truncate result $\lfloor d/2^{k-1} \rfloor$ $(d = d_0 + d_1)$. To eliminate this error, we observe that the least significant bit of $\lfloor d/2^{k-1} \rfloor$ is 0, since $d = x + ovfl$, $x \in [0, 2^{k-1})$, and $ovfl = 0$ *or* 2^k. This insight allows the parties to determine whether the truncated sum has one positive bit error based on its least significant bit. If the least significant bit is 1 after truncating $k-1$ bits, one positive bit error occurs; if it is 0, the truncated sum is correct. Thus, the parties can compute the least significant bit of the truncated sum to eliminate its impact.

In summary, to compute $\langle ovfl \rangle_{2^\ell}$, P_0 and P_1 first truncate d_0 and d_1 with $k-1$ bits, respectively. This process removes the non-overflow portion and potentially introduces one positive bit error compared to the correct truncate result $\lfloor d/2^{k-1} \rfloor$. Next, the parties securely compute the least significant bit of the truncated sum and eliminate the possible one positive bit error. Finally, the parties multiply the secret-shared truncated sum with 2^{k-1}, then get $\langle ovfl \rangle_{2^\ell}$.

Theorem 1. *Let c be an integer, satisfying $c < k < \ell - 1$. Let $d \in [0, 2^{k+1})$, d_0 and $d_1 \in [0, 2^k)$ satisfying $d_0 + d_1 = d$. Then:*
$(d_0 \gg c) + (-((-d_1) \gg c)) = \lfloor d/2^c \rfloor + bit$, *where* $bit = 0$ *or* 1.

Proof. The proof is presented in Appendix G. □

As is shown in Protocol 6, the parties in the protocol *ConvertShare* input a secret-shared value $\langle x \rangle_{2^k}$ $(x \in [0, 2^{k-1}))$ [3] on the small ring $\mathbb{Z}_{2^k}$ and two public integers k and ℓ $(k < \ell)$. The parties get a secret-shared value $\langle x' \rangle_{2^\ell}$ on the large ring $\mathbb{Z}_{2^\ell}$, such that $x = x'$, as the output.

During the offline phase of the protocol *ConvertShare*, the parties generate a daBit [13], which consists of the shares of a bit on both $\mathbb{Z}_2$ (a ring of size 2) and $\mathbb{Z}_{2^\ell}$. The daBit is used for masking the least significant bit of the truncated sum.

The online phase of the protocol *ConvertShare* consists of three stages. (1) The first stage (Line 1-5): The parties compute $\langle d \rangle_{2^\ell}$ and the secret-shared truncated sum $\langle truncsum \rangle_{2^\ell}$. To accomplish this, P_0 and P_1 first locally compute d_0 and d_1 (Line 1), and then truncate d_0 and d_1 to get $truncd_0$ and $truncd_1$ (Line 2). Next, P_0 and P_1 share these values on $\mathbb{Z}_{2^\ell}$ (Line 3). Finally, the parties compute $\langle d \rangle_{2^\ell}$ and $\langle truncsum \rangle_{2^\ell}$ by summing the secret-shared values (Line 4-5). (2) The second stage (Line 6-9): The parties compute the secret-shared one positive bit error $\langle bit \rangle_{2^\ell}$. To accomplish this, P_0 and P_1 first mask the least significant bits of $truncd_0$ and $truncd_1$ using the daBit (Line 6). Then, P_0 sends its masked bit to both P_1 and P_2. P_1 sends its masked bit to P_0 and P_2 (Line 7). Finally, the parties compute the secret-shared least significant bit $\langle bit \rangle_{2^\ell}$ by XORing the two masked bits and then removing the mask (Line 8-9). (3) The third stage (Line 10-11): the parties compute the shares of the secret x on $\mathbb{Z}_{2^\ell}$ by first computing the secret-shared overflow value $\langle ovfl \rangle_{2^\ell}$, and then subtracting $\langle ovfl \rangle_{2^\ell}$ from $\langle d \rangle_{2^\ell}$.

[3] Note that the assumption $x \in [0, 2^{k-1})$ is reasonable when we train decision trees since all the values need to be converted are all positive, although the studies [4, 13, 17, 20] adopt the assumption $x \in [0, 2^k)$.

4.4 Communication Complexity Comparison

We compare the communication complexity of Ents with two three-party frameworks [1, 18] for training a decision tree from two aspects: communication sizes and communication rounds. As shown in Table 1, Ents outperforms these two frameworks in both communication sizes and communication rounds. (1) Compared to the framework [1], Ents reduces a exponential term 2^h to a linear term h in the communication size complexity ($2^h mn \log n\ell \log \ell \rightarrow hmn \log nk \log k$). This improvement comes from the communication optimization based on the secure radix sort protocols [10] in Ents eliminates the need to train each node with a padded dataset. (2) Compared to the framework [18], Ents reduces a linear term h in both communication sizes and communication rounds ($hmn\ell^2 \rightarrow mnk^2$ and $h\ell \rightarrow k$). This improvement comes from that the communication optimization based on the secure radix sort protocols [10] eliminates the need to repeatedly generate permutations. (3) Compared to both frameworks [1, 18], Ents reduces the dependency on the parameter ℓ to a smaller parameter k ($k < \ell$) in both communication sizes and rounds. This improvement comes from the optimization based on our proposed share conversion protocol allows Ents to perform most computations on the small ring $\mathbb{Z}_{2^k}$, rather than on the large ring $\mathbb{Z}_{2^\ell}$.

Besides, we present the communication complexity of the basic primitives and the detailed communication complexity analysis of each training protocol in Ents in Appendix F.

Table 1: Communication complexity of Ents vs the frameworks [1, 18]. We assume $v < \log n \approx m < n$. The communication complexity of these three frameworks is all analyzed based on the primitive's communication complexity presented in Appendix F.

Framework	Sizes	Rounds
Ents	$O(hmn \log nk \log k + mnk^2)$	$O(h \log n \log k + k)$
[1]	$O(2^h mn \log n\ell \log \ell + mn\ell^2)$	$O(h \log n \log \ell + \ell))$
[18]	$O(hmn \log n\ell \log \ell + hmn\ell^2)$	$O(h \log n \log \ell + h\ell))$

5 PERFORMANCE EVALUATION

5.1 Implementation and Experiment Settings

Implementation. We implement Ents based on MP-SPDZ [25], which is a widely used open-source framework for evaluating multi-party machine learning frameworks in the research field [12, 28, 29, 37]. In Ents, computations are performed on a small ring $\mathbb{Z}_{2^k}$ with $k = 32$ and a large ring $\mathbb{Z}_{2^\ell}$ with $\ell = 128$. Besides, during the execution of the secure division operations, the bit length, f, for the decimal part is set to $2\lceil \log n \rceil$, where n is the number of samples in the training dataset.

Table 2: Detailed information of datasets employed in our evaluation.

Dataset	#Sample	#Attribute	#Label
Kohkiloyeh	100	5	3
Diagnosis	120	6	2
Iris	150	4	3
Wine	178	13	3
Cancer	569	32	2
Tic-tac-toe	958	9	2
Adult	48,842	14	2
Skin Segmentation	245,057	4	2

Experiment Environment: We conduct experiments on a Linux server equipped with a 32-core 2.4 GHz Intel Xeon CPU and 512GB of RAM. Each party in Ents is simulated by a separate process with four threads. As for the network setting, we consider two scenarios: one is the LAN setting with a bandwidth of 1 gigabyte per second (GBps for short) and sub-millisecond round-trip time (RTT for short) latency. The other is the WAN setting with 5 megabytes per second (MBps for short) bandwidth and 40ms RTT latency. We apply the tc tool[4] to simulate these two network settings.

Datasets: As is shown in Table 2, we employ eight widely-used real-world datasets from the UCI repository [26] in our experiments. These datasets span a broad range in size, from the relatively small dataset, Kohkiloyeh, with only 100 samples, to the moderate-size dataset in the real world, Skin Segmentation, containing over 245,000 samples.

5.2 Accuracy of Ents

Table 3: Accuracy of Ents vs. scikit-learn on eight widely used datasets. The height of decision trees is set to six.

Dataset	Ents	scikit-learn
Kohkiloyeh	0.7352	0.7352
Diagnosis	1.0	1.0
Iris	0.9960	0.9607
Wine	0.8622	0.8590
Cancer	0.9388	0.9398
Tic-tac-toe	0.8987	0.8987
Adult	0.8494	0.8526
Skin Segmentation	0.9898	0.9898

We evaluate the accuracy of Ents by comparing it with the plaintext training algorithm for decision trees in scikit-learn [36] on the datasets shown in Table 2. We divide each dataset into a training set and a test set with a ratio of $2:1$ and set the height of the decision trees to six. We conduct five runs for both Ents and scikit-learn. The average accuracy of the five runs is shown in Table 3. As is shown in Table 3, the accuracy of Ents and scikit-learn are almost the same. These accuracy results prove that Ents can accurately train a decision tree. Besides, we observe that the main reason for the small difference in the accuracy results is that when multiple split points have the same maximum modified Gini impurity, Ents and scikit-learn usually select different split points.

[4]https://man7.org/linux/man-pages/man8/tc.8.html

5.3 Efficiency of Ents

We compare the efficiency of Ents with two state-of-the-art three-party frameworks [1, 18]. These two frameworks are both based on RSS and adopt the same security model as Ents. Because these two frameworks are not open-sourced by the authors, for the framework proposed by Hamada et al. [18], we adopt the open-source implementation in MP-SPDZ, and modify the implementation to support multi-class datasets. For the framework proposed by Abspoel et al. [1], we also implement it based on MP-SPDZ. Besides, we perform the computations of these two frameworks on the large ring $\mathbb{Z}_{2^\ell}$ ($\ell = 128$), so that these two frameworks could support datasets of almost the same size as Ents.

The experimental results in Table 4 show that:

- In the LAN setting, Ents significantly outperforms the two frameworks by $3.5\times \sim 5.8\times$ in terms of training time. The improvement primarily comes from that the two communication optimizations in Ents significantly reduce communication sizes, reaching an improvement of $5.5\times \sim 9.3\times$ compared to the baselines.
- In the WAN setting, Ents significantly outperforms the baselines by $4.5\times \sim 6.7\times$ in terms of training time. The improvement primarily comes from two facts. The first one is the same as that of the LAN setting, i.e. the two optimizations in Ents significantly reduce communication sizes. The second one is that the two optimizations in Ents also significantly reduce communication rounds, reaching an improvement of $3.9\times \sim 5.3\times$ compared to the baselines.
- Especially, Ents requires less than three hours (8,736.53s) in the WAN setting to train a decision tree on the dataset Skin Segmentation, which is a moderate-size dataset in the real world. This result shows that Ents is promising in the practical usage of privacy preserving training for decision trees.

5.4 Effectiveness of Communication Optimizations

In order to further show the effectiveness of the two communication optimizations in Ents, we implement the two communication optimizations in Hamada et al.'s framework [18]. We name Hamada et al.'s framework with the communication optimization based on the secure radix sort protocols [10] as Hamada et al.'s framework-radixsort, and Hamada et al.'s framework with the communication optimization based on our proposed share conversion protocol as Hamada et al.'s framework-convert. We compare the training time, communication sizes, and communication rounds of Ents, these two frameworks, and the original Hamada et al.'s framework.

As is shown in Figure 2, both the communication optimizations significantly improve the efficiency of Hamada et al.'s framework. (1) The communication optimization based on the secure radix sort protocols [10] improves the training time of Hamada et al.'s framework by $2.9\times \sim 3.4\times$ in the LAN setting and $3.1\times \sim 3.7\times$ in the WAN setting. In terms of communication sizes and communication rounds, this communication optimization yields improvements of $3.2\times \sim 3.5\times$ and $3.5\times \sim 3.6\times$, respectively. The improvements are primarily because this communication optimization enables Hamada et al.'s framework-radixsort to securely generate permutations only once, while the original Hamada et al.'s framework

Table 4: Online training time (seconds), communication sizes (MBs), and communication rounds of Ents vs. two three-party frameworks [1, 18] to train a decision tree with height six. Note that the communication rounds, reported by the virtual machine of MP-SPDZ, are the cumulative totals across four threads. Therefore, the communication rounds exceed the actual number required.

	Framework	Dataset							
		Kohkiloyeh	Diagnosis	Iris	Wine	Cancer	Tic-tac-toe	Adult	Skin Segmentation
Online Training Time in LAN	Ents	**1.5** (5.2×)	**1.6** (5.8×)	**1.1** (5.3×)	**5.1** (4.5×)	**23.3** (5.2×)	**14.0** (4.9×)	**1,187.8** (4.1×)	**1,783.5** (3.5×)
	[1]	9.9	11.4	9.4	33.1	179.8	108.5	13,043.9	18,601.9
	[18]	7.8	9.3	5.9	23.1	122.3	68.8	4,933.2	6,271.1
Online Training Time in WAN	Ents	**128.9** (5.2×)	**129.3** (5.2×)	**78.9** (4.5×)	**302.3** (4.5×)	**580.5** (5.1×)	**238.6** (5.0×)	**5,195.3** (6.7×)	**8,736.5** (5.2×)
	[1]	679.8	682.4	420.2	1,389.2	3,397.0	1,418.9	70,168.4	85,899.1
	[18]	679.4	681.3	356.9	1,374.8	3,004.5	1,210.3	35,032.2	42,315.1
Communication Sizes (All parties)	Ents	**24.9** (9.3×)	**35.8** (9.2×)	**34.1** (9.2×)	**140.3** (8.9×)	**980.7** (9.2×)	**501.3** (9.2×)	**51,725.3** (8.7×)	**90,361.8** (5.5×)
	[1]	232.5	331.9	315.3	1,251.8	9,110.4	4,637.0	739,511.0	875,991.0
	[18]	276.3	397.0	336.3	1,288.2	9,734.3	4,947.9	450,378.0	504,693.0
Communication Rounds	Ents	**17,526** (4.6×)	**20,882** (4.5×)	**15,931** (4.3×)	**54,472** (3.9×)	**111,242** (5.1×)	**33,914** (5.1×)	**61,638** (5.3×)	**15,142** (4.8×)
	[1]	80,912	96,042	69,824	216,326	570,042	174,948	392,348	98,030
	[18]	106,286	127,352	86,300	278,077	655,562	197,498	331,412	73,488

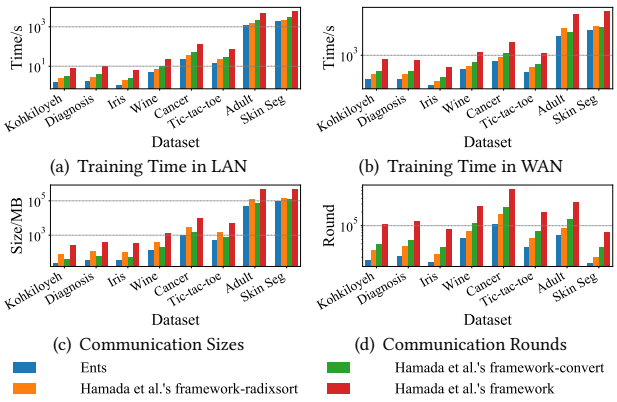

Figure 2: Online training time (seconds), communication sizes (MBs), and communication rounds of Ents, Hamada et al.'s framework-radixsort, Hamada et al.'s framework-convert, and Hamada et al.'s framework. 'Skin Seg' refers to Skin Segmentation.

requires securely generating permutations when training each internal layer of a decision tree. (2) The communication optimization based on our proposed share conversion protocol improves the training time of Hamada et al.'s framework by 1.9× ∼ 2.5× in the LAN setting and 2.3× ∼ 4.7× in the WAN setting. In terms of communication sizes and communication rounds, this communication optimization yields improvements of 4.2× ∼ 7.0× and 2.2× ∼ 2.7×, respectively. The improvements are primarily because this communication optimization enables Hamada et al.'s framework-convert to perform most computations on the small ring to reduce huge communication size. Additionally, this optimization also reduces communication sizes and communication rounds for the secure generation of permutations, because the secure generation of permutations can also be performed on the small ring.

With the two communication optimizations, Ents significantly outperforms Hamada et al.'s framework in terms of the training time by 3.5× ∼ 5.8× in the LAN setting, 4.5× ∼ 6.7× in the WAN setting. In terms of communication sizes and communication rounds, Ents significantly outperforms Hamada et al.'s framework by 5.5× ∼ 11.0× and 4.8× ∼ 6.0×, respectively.

Note that although Hamada et al.'s framework-radixsort requires more communication sizes than Hamada et al.'s framework-convert, Hamada et al.'s framework-radixsort still outperforms Hamada et al.'s framework-convert in the LAN setting. The primary reason for this phenomenon is that the communication optimization based on the secure radix sort protocols [10] also saves a lot of memory access time for Hamada et al.'s framework-radixsort. The process of generating permutations requires a lot of random memory access, which is very time-consuming. The communication optimization based on the secure radix sort protocols [10] enables Hamada et al.'s framework-radixsort to generate permutations only once. Thus, the memory access time of Hamada et al.'s framework-radixsort is much less than Hamada et al.'s framework-convert.

5.5 Efficiency of Conversion Protocol

We compare the efficiency of our proposed share conversion protocol, *ConvertShare* (Protocol 6), against three state-of-the-art conversion protocols: (1) the conversion protocol based on arithmetic and boolean share conversion (*Convert-A2B* for short) [35]. (2) the conversion protocol based on Dabits (*Convert-Dabits* for short) [4]. (3) the conversion protocol based on function secret sharing (*Convert-FSS* for short) [17]. We implement all these protocols in MP-SPDZ [25]

To convert the replicated shares of a secret from $\mathbb{Z}_{2^{32}}$ to $\mathbb{Z}_{2^{128}}$, the communication sizes and communication rounds of the above

conversion protocols are as follows: (1) our proposed protocol *ConvertShare* (Protocol 6) requires a communication size of 516 bits in one communication round. (2) the protocol *Convert-A2B* requires a communication size of 1, 506 bits in six communication rounds. (3) the protocol *Convert-Dabits* requires a communication size of 22, 860 bits in three communication rounds. (4) the protocol *Convert-FSS* requires a communication size of 512 bits in two communication rounds. Besides, to convert the replicated shares of a vector of size l from $\mathbb{Z}_{2^{32}}$ to $\mathbb{Z}_{2^{128}}$, the communication sizes of the above protocols will scale up by a factor of l. While the number of communication rounds remains constant.

We evaluate the online runtime of the above conversion protocols for converting secret-shared vectors of size 1, 10, 100, 1000, and 10000 in both LAN and WAN settings. As is shown in Table 5, our proposed protocol *ConvertShare* significantly outperforms the other protocols by 2.0× ∼ 4.8× in the LAN setting and by 2.0× ∼ 4.7× in the WAN setting. This advancement is primarily due to its lower communication sizes and fewer communication rounds. Note that although the protocol *Convert-FSS* has a marginally smaller communication size than our proposed protocol *ConvertShare*, the protocol *Convert-FSS* burdens significantly higher local computation overhead, owing to its reliance on function secret sharing. Therefore, our proposed protocol *ConvertShare* is more efficient than the protocol *Convert-FSS* in both the LAN and WAN settings.

Table 5: Online runtime (milliseconds) of conversion protocols for converting secret-shared vectors of size 1, 10, 100, 1000, and 10000 from $\mathbf{Z}_{2^{32}}$ to $\mathbf{Z}_{2^{128}}$.

	Protocol	Vector Size				
		1	10	100	1,000	10,000
LAN	*ConvertShare*	**0.58** (2.0×)	**0.59** (4.5×)	**1.2** (4.8×)	**6.2** (4.1×)	**57.0** (3.3×)
	Convert-A2B [35]	2.6	2.7	5.8	26.7	191.1
	Convert-Dabits [4]	1.2	4.5	16.2	137.4	1,310.8
	Convert-FSS [17]	5.0	19.1	113.0	1,103.9	11,004.5
WAN	*ConvertShare*	**40.6** (2.0×)	**40.8** (2.0×)	**42.0** (4.1×)	**60.4** (4.7×)	**202.4** (2.7×)
	Convert-A2B [35]	243.3	243.7	247.6	284.5	563.5
	Convert-Dabits [4]	141.7	144.7	173.1	519.3	3,394.3
	Convert-FSS [17]	84.7	84.9	180.9	1,711.9	10,979.5

6 DISCUSSION

Discrete Attributes in Training Datasets. Although the protocols (Protocol 2, 4, 5, 7, 8, 9) in our proposed Ents only process continuous attributes in training datasets, these protocols can be modified to process discrete attributes. According to the technical routine introduced by Abspoel et al. [1], the main modification to the above training protocols to process discrete attributes are as follows: (1) The procedure of generating and applying permutations should be removed, because training on discrete attributes does not require sorting the samples according to the attributes. (2) The less-than tests between the samples and split threshold should be replaced with equality tests, because the splitting test for discrete attributes requires checking whether the attribute of samples is equal to the split threshold, rather than checking whether the attribute of samples is less than the split threshold.

Ents in Two-Party Scenarios. Ents can be adapted to train a decision tree in two-party scenarios by modifying the basic MPC protocols introduced in Section 2.3 accordingly. Note that except for our proposed conversion protocol *ConvertShare*, all the other basic protocols are already supported for the two-party semi-honest security model in MP-SPDZ. Thus, we provide the modification of our proposed protocol *ConvertShare* to two-party scenarios in Appendix D.1.

To evaluate the efficiency of two-party Ents, we perform performance evaluation for two-party Ents and two state-of-the-art two-party frameworks [3, 31]. The experimental results, provided in Appendix D.2, show that two-party Ents also significantly outperforms the two-party frameworks [3, 31], with a training time improvement of 14.3× ∼ 1, 362× in the LAN setting and 4.7× ∼ 6.4× in the WAN setting.

Security of Ents. Ents is designed to be secure under a three-party semi-honest security model with an honest majority. Note that except for the protocol *ConvertShare*, other protocols in Ents are constructed using the security-proven protocols introduced in Section 2.3. To prove the security of the protocol *ConvertShare*, we provide a security analysis using the standard real/ideal world paradigm in Appendix E.

Practical Usages of Ents. Ents is promising in the practical usage of privacy preserving training for decision trees. According to the experimental results, Ents costs less than three hours to train a decision tree on a real-world dataset with more than 245,000 samples in the WAN setting. Besides, with a larger bandwidth and more powerful servers, the training time of Ents can be further reduced.

Future Work. In the future, we will support more tree-based models, such as XGBoost [8], in Ents. Because decision trees are the basic components of these tree-based models, the optimizations proposed in this paper should also be suitable for them.

7 RELATED WORK

Multi-party training frameworks for decision trees. Over the last decade, many multi-party training frameworks [1–3, 14, 18, 21, 30, 31, 43, 46] for decision trees have been proposed. These frameworks can be categorized into two types according to whether they disclose some private information during the training process. The first type frameworks [14, 21, 30, 31, 43] disclose some private information, such as the split points of decision tree nodes and Gini impurity, and rely on the disclosed private information to speed up the training process. For instance, the framework proposed by Lu et al. [21] discloses the split points of decision tree nodes to a specific party, so that some comparisons between the samples and the split thresholds can be performed locally to speed up the training process. However, Zhu et al. [49] demonstrate that the disclosed information may be exploited to infer training data. Thus, the frameworks of this type may be unsuitable in scenarios where training data must be strictly protected.

The second type frameworks [1–3, 18, 46], on the other hand, does not disclose private information during the training process but suffer from low accuracy or inefficiency. Vaidya et al.[46] and Adams et al.[2] both propose frameworks for securely training random decision trees. However, random decision trees usually have lower accuracy compared to conventional decision trees. Abspoel

et al.[1] and Hamada et al.[18] both propose three-party training frameworks based on RSS for decision trees. However, these frameworks are inefficient due to high communication overhead. Akavia et al. [3] introduced a two-party framework based on homomorphic encryption. However, this framework is also inefficient due to high computation overhead. Due to limitations in accuracy or efficiency, these frameworks of this type remain impractical.

Shares conversion protocols. Recently, several share conversion protocols have been proposed [4, 13, 17, 20, 24, 35]. Kelkar et al. [24] introduce a share conversion protocol that does not require interaction but suffers from a high failure probability, making it unsuitable for training decision trees. The arithmetic and boolean conversion protocol proposed by Mohassel and Rindal [35] can be used to implement the correct share conversion between a small ring and a large ring. However, this method is inefficient since it requires six communication rounds. Aly et al.[4] propose a share conversion protocol based on Dabits [13], but this protocol requires considerable communication sizes and three communication rounds, which make it inefficient too. Gupta et al.[17] and Jawalkar et al. [20] both propose share conversion protocols based on function secret sharing [6], requiring zero and one communication rounds, respectively. However, when combining function secret sharing with replicated secret sharing, two additional communication rounds are introduced: one for masking shares and another for resharing shares. Therefore, Gupta's method requires two communication rounds, and Jawalkar's method requires three communication rounds. Besides, these two methods also suffer from high computation overhead due to dependence on function secret sharing. In summary, the above protocols either suffer from high failure probability or lack efficiency.

Our advantages. (1) Our proposed framework `Ents` does not disclose private information during the training process, and meanwhile, it attains the same accuracy level as the plaintext training algorithm for decision trees in `scikit-learn` and significantly outperforms the existing frameworks that also do not disclose private information during the training process in terms of efficiency. (2) Our proposed conversion protocol demonstrates superior efficiency while guaranteeing its correctness.

8 CONCLUSION

In this paper, we propose `Ents`, an efficient three-party training framework for decision trees, and optimize the communication overhead from two-folds: (1) We present a series of training protocols based on the secure radix sort protocols [10] to efficiently and securely split a dataset with continuous attributes. (2) We propose an efficient share conversion protocol to convert shares between a small ring and a large ring to reduce the huge communication overhead incurred by performing almost all computations on a large ring. Experimental results from eight widely used datasets demonstrate that `Ents` outperforms state-of-the-art frameworks by $5.5\times \sim 9.3\times$ in communication sizes and $3.9\times \sim 5.3\times$ in communication rounds. In terms of training time, `Ents` yields an improvement of $3.5\times \sim 6.7\times$. Especially, `Ents` requires less than three hours to train a decision tree on a real-world dataset (Skin Segmentation) with more than 245,000 samples in the WAN setting. These results show that `Ents` is promising in the practical usage of privacy preserving training for decision trees.

ACKNOWLEDGEMENT

This paper is supported by the National Key R&D Program of China (2023YFC3304400), Natural Science Foundation of China (62172100, 92370120). We thank all anonymous reviewers for their insightful comments. Weili Han is the corresponding author.

REFERENCES

[1] Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. 2021. Secure training of decision trees with continuous attributes. In *Proceedings of the Privacy Enhancing Technologies Symposium*. 167–187.

[2] Samuel Adams, Chaitali Choudhary, Martine De Cock, Rafael Dowsley, David Melanson, Anderson Nascimento, Davis Railsback, and Jianwei Shen. 2022. Privacy-preserving training of tree ensembles over continuous data. In *Proceedings of the Privacy Enhancing Technologies Symposium*. 205–226.

[3] Adi Akavia, Max Leibovich, Yehezkel S Resheff, Roey Ron, Moni Shahar, and Margarita Vald. 2022. Privacy-preserving decision trees training and prediction. *ACM Transactions on Privacy and Security* (2022), 1–30.

[4] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P Smart, and Tim Wood. 2019. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 33–44.

[5] Monika Arya and Hanumat G Sastry. 2022. Stock indices price prediction in real time data stream using deep learning with extra-tree ensemble optimisation. *International Journal of Computational Science and Engineering* (2022), 140–151.

[6] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1292–1303.

[7] Chun-Hao Chen, Yin-Ting Lin, Shih-Ting Hung, and Mu-En Wu. 2021. Forecasting Stock Trend Based on the Constructed Anomaly-Patterns Based Decision Tree. In *Proceedings of the 13th Asian Conference on Intelligent Information and Database Systems*. 606–615.

[8] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*. 409–437.

[10] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. 2019. An efficient secure three-party sorting protocol with an honest majority. *Cryptology ePrint Archive* (2019).

[11] Michele Ciampi, Vipul Goyal, and Rafail Ostrovsky. 2021. Threshold garbled circuits and ad hoc secure computation. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 64–93.

[12] Anders Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic four: {Honest-Majority}{Four-Party} secure computation with malicious security. In *Proceedings of the 30th USENIX Security Symposium*. 2183–2200.

[13] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New primitives for actively-secure MPC over rings with applications to private machine learning. In *Proceedings of 2019 IEEE Symposium on Security and Privacy*. 1102–1120.

[14] Jayanti Dansana, Debadutta Dey, and Raghvendra Kumar. 2013. A novel approach: CART algorithm for vertically partitioned database in multi-party environment. In *Proceedings of the 2013 IEEE Conference on Information & Communication Technologies*. 829–834.

[15] Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. 2014. Practical Secure Decision Tree Learning in a Teletreatment Application. In *Proceedings of the 18th International Conference on Financial Cryptography and Data Security*.

[16] Yfke Dulek, Christian Schaffner, and Florian Speelman. 2016. Quantum homomorphic encryption for polynomial-sized circuits. In *Proceedings of the Annual International Cryptology Conference*. 3–32.

[17] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. 2022. LLAMA: A Low Latency Math Library for Secure Inference. In *Proceedings of the Privacy Enhancing Technologies Symposium*. 274–294.

[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 *Proceedings of the Privacy Enhancing Technologies Symposium*. 343–364.

[19] Mehmet Tahir Huyut and Hilal Üstündağ. 2022. Prediction of diagnosis and prognosis of COVID-19 disease by blood gas parameters using decision trees machine learning model: A retrospective observational study. *Medical Gas Research* (2022), 60.
[20] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2024. Orca: FSS-based Secure Training and Inference with GPUs. In *Proceedings of 2024 IEEE Symposium on Security and Privacy*. 63–63.
[21] 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 *Proceedings of 32nd USENIX Security Symposium*. 6435–6451.
[22] Kaggle. 2020. State of Data Science and Machine Learning 2020. https://www.kaggle.com/kaggle-survey-2020.
[23] Kaggle. 2021. State of Data Science and Machine Learning 2021. https://www.kaggle.com/kaggle-survey-2021.
[24] Mahimna Kelkar, Phi Hung Le, Mariana Raykova, and Karn Seth. 2022. Secure poisson regression. In *Proceedings of 31st USENIX Security Symposium*. 791–808.
[25] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.
[26] Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. 2023. The UCI Machine Learning Repository. https://archive.ics.uci.edu
[27] Chee Sun Lee, Peck Yeng Sharon Cheang, and Massoud Moslehpour. 2022. Predictive analytics in business analytics: decision tree. *Advances in Decision Sciences* (2022), 1–29.
[28] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. 2021. Muse: Secure inference resilient to malicious clients. In *Proceedings of 30th USENIX Security Symposium*. 2201–2218.
[29] Xiling Li, Rafael Dowsley, and Martine De Cock. 2021. Privacy-preserving feature selection with secure multiparty computation. In *Proceedings of International Conference on Machine Learning*. 6326–6336.
[30] Ye Li, Zoe L Jiang, Lin Yao, Xuan Wang, Siu-Ming Yiu, and Zhengan Huang. 2019. Outsourced privacy-preserving C4. 5 decision tree algorithm over horizontally and vertically partitioned dataset among multiple parties. *Cluster Computing* (2019), 1581–1593.
[31] Lin Liu, Rongmao Chen, Ximeng Liu, Jinshu Su, and Linbo Qiao. 2020. Towards practical privacy-preserving decision tree training and evaluation in the cloud. *IEEE Transactions on Information Forensics and Security* (2020), 2914–2929.
[32] Ziyao Liu, Ivan Tjuawinata, Chaoping Xing, and Kwok-Yan Lam. 2020. Mpc-enabled privacy-preserving neural network training against malicious attack. *arXiv preprint arXiv:2007.12557* (2020).
[33] Wei-Yin Loh. 2011. Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery* (2011), 14–23.
[34] F Manzella, G Pagliarini, G Sciavicco, and IE Stan. 2023. The voice of COVID-19: Breath and cough recording classification with temporal decision trees and random forests. *Artificial Intelligence in Medicine* (2023), 102486.
[35] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 35–52.
[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* (2011), 2825–2830.
[37] Sikha Pentyala, Rafael Dowsley, and Martine De Cock. 2021. Privacy-preserving video classification with convolutional neural networks. In *Proceedings of International conference on machine learning*. 8487–8499.
[38] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* (1986), 81–106.
[39] J Ross Quinlan. 2014. *C4. 5: programs for machine learning*.
[40] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. Sirnn: A math library for secure rnn inference. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy*. 1003–1020.
[41] Wenqiang Ruan, Mingxin Xu, Wenjing Fang, Li Wang, Lei Wang, and Weili Han. 2023. Private, efficient, and accurate: Protecting models trained by multi-party learning with differential privacy. In *Proceedings of 2023 IEEE Symposium on Security and Privacy*. 1926–1943.
[42] SEAL 2022. Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..
[43] M Antony Sheela and K Vijayalakshmi. 2013. A novel privacy preserving decision tree induction. In *Proceedings of the 2013 IEEE Conference on Information & Communication Technologies*. 1075–1079.
[44] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. 2021. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *Proceedings of 2021 IEEE Symposium on Security and Privacy*. 1021–1038.
[45] Harsh Tyagi, Aditya Agarwal, Aakash Gupta, Kanak Goel, Anand Kumar Srivastava, and Akhilesh Kumar Srivastava. 2022. Prediction and diagnosis of diabetes using machine learning classifiers. *International Journal of Forensic Software Engineering* (2022), 335–347.
[46] Jaideep Vaidya, Basit Shafiq, Wei Fan, Danish Mehmood, and David Lorenzi. 2014. A Random Decision Tree Framework for Privacy-Preserving Data Mining. *IEEE Transactions on Dependable and Secure Computing* (2014).
[47] Paul Voigt and Axel Von dem Bussche. 2017. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing* (2017).
[48] Lijing Zhou, Ziyu Wang, Hongrui Cui, Qingrui Song, and Yu Yu. 2023. Bicoptor: Two-round Secure Three-party Non-linear Computation without Preprocessing for Privacy-preserving Machine Learning. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy*. 534–551.
[49] Zutao Zhu and Wenliang Du. 2010. Understanding privacy risk of publishing decision trees. In *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*. 33–48.

A NOTATIONS USED IN THIS PAPER

As is shown in Table 6, we list the notations used in this paper for clarity purposes.

B REMAINING PROTOCOLS FOR TRAINING PROCESS

The protocol *AttributeWiseSplitSelection* used to compute the modified Gini impurity and split threshold with an attribute a_i is shown in Protocol 7. The parties in this protocol input a secret-shared group flag vector $\langle \vec{g} \rangle_{2^k}$, a secret-shared attribute vector $\langle \vec{a}_i \rangle_{2^k}$, and a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$. They get a secret-shared sample-threshold vector $\langle \overrightarrow{spth_i} \rangle_{2^k}$, which are computed with only the attribute a_i, and a secret-shared modified Gini impurity vector $\langle \overrightarrow{gini_i} \rangle_{2^k}$ as the output. The attribute vector and the label vector are assumed to have been sorted according to $\vec{a}_i$ within each node.

The protocol *AttributeWiseSplitSelection* consists of three stages: (1) The first stage (Line 1-3): the parties compute the modified Gini impurity and the split threshold for each split point $(a_i, \vec{t}[j])$, where $\vec{t}[j] = (\vec{a}_i[j] + \vec{a}_i[j+1])/2$ for each $j \in [0, n-2]$ and $\vec{t}[n-1] = \vec{a}_i[n-1]$. (2) The second stage (Line 4-6): the parties check whether the i-th sample is the final sample of a node. If it is true, the modified Gini impurity is set to the *MinValue* (*MinValue* refers to the minimum value in the ring $\mathbb{Z}_{2^k}$) to avoid splitting between the i-th sample and $(i+1)$-th sample. (3) The third stage (Line 7): the parties call the protocol *GroupMax* (introduced in Section 2.3.5) to obtain the secret-shared sample-threshold for each node and the secret-shared maximum modified Gini impurity.

Note that this protocol is the same as the protocol proposed by Hamada et al. [18].

Protocol 7 *AttributeWiseSplitSelection*

Input: A secret-shared group flag vector $\langle \vec{g} \rangle_{2^k}$, a secret-shared attribute vector $\langle \vec{a}_i \rangle_{2^k}$, and a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$.
Output: A secret-shared sample-threshold vector $\langle \overrightarrow{spth_i} \rangle_{2^k}$ and a secret-shared modified Gini impurity vector $\langle \overrightarrow{gini_i} \rangle_{2^k}$.

1: $\langle \overrightarrow{gini_i} \rangle_{2^k} = ModifiedGini(\langle \vec{g} \rangle_{2^k}, \langle \vec{y} \rangle_{2^k})$.
2: $\langle \vec{t}[j] \rangle_{2^k} = (\langle \vec{a}_i[j] \rangle_{2^k} + \langle \vec{a}_i[j+1] \rangle_{2^k})$ / 2 for each $j \in [0, n-1)$.
3: $\langle \vec{t}[n-1] \rangle_{2^k} = \langle \vec{a}_i[n-1] \rangle_{2^k}$.
4: $\langle \vec{p}[j] \rangle_{2^k} = \langle \vec{g}[j+1] \rangle_{2^k}$ or $(\langle \vec{a}_i[j] \rangle_{2^k} \stackrel{?}{=} \langle \vec{a}_i[j+1] \rangle_{2^k})$ for each $j \in [0, n-2)$.
5: $\langle \vec{p}[n-1] \rangle_{2^k} = \langle 1 \rangle_{2^k}$.
6: $\langle \overrightarrow{gini_i} \rangle_{2^k} = MinValue * \langle \vec{p} \rangle_{2^k} + \langle \overrightarrow{gini_i} \rangle_{2^k} * (1 - \langle \vec{p} \rangle_{2^k})$.
7: $\langle \overrightarrow{gini_i} \rangle_{2^k}, \langle \overrightarrow{spth_i} \rangle_{2^k} = GroupMax(\langle \vec{g} \rangle_{2^k}, \langle \overrightarrow{gini_i} \rangle_{2^k}, \langle \vec{t} \rangle_{2^k})$.

Table 6: Notations used in this paper.

Notation	Description
$\mathcal{D}$	The dataset for training a decision tree.
$\mathcal{D}_{condition}$	A subset of $\mathcal{D}$. It contains all the samples satisfying the given condition in $\mathcal{D}$.
$\mathcal{D}^{node}$	The dataset for training a node. It is a subset of $\mathcal{D}$.
n	The number of samples in the training dataset.
m	The number of attributes of a sample.
v	The number of labels.
t	The split threshold of a node.
$a_0, \cdots, a_{m-1}$	The m attributes of a sample.
y	The label of a sample.
h	The height of a decision tree.
P_i	Party i participating in the secure decision tree training, where $i \in \{0, 1, 2\}$.
$\mathbb{Z}_2, \mathbb{Z}_{2^k}, \mathbb{Z}_{2^\ell}$	A ring of size $2, 2^k, 2^\ell (2 < k < \ell)$ respectively.
f	The bit length for representing the decimal part.
$\langle x \rangle$	The replicated shares of a secret x, i.e. P_i holds $\langle x \rangle^i = (\llbracket x \rrbracket^i, \llbracket x \rrbracket^{i+1})$
$\vec{a}_i$	The i-th attribute vector that contains the i-th attribute values of all the samples, i.e. $\vec{a}_i[j]$ represents the value of i-th attribute of j-th sample.
$\vec{y}$	The label vector that contains the label of all the samples, i.e. $\vec{y}[i]$ represents the label of of i-th sample.
$\vec{\pi}, \vec{\alpha}$	Permutations used to reorder vectors.
$\vec{g}$	The group flag vector used to indicates group boundaries.
$\overrightarrow{spnd}$	A sample-node vector. The equation $\overrightarrow{spnd}[j] = i$ represents that the j-th sample belongs to the node whose *nid* is i.
$\overrightarrow{spat}$	A sample-attribute vector. The equation $\overrightarrow{spat}[j] = i$ represents that the split attribute index of the node to which the j-th sample belongs is i.
$\overrightarrow{spth}$	A sample-threshold vector . The equation $\overrightarrow{spth}[j] = i$ represents that the split threshold of the node to which the j-th sample belongs is i.
$\overrightarrow{spnd}^{(k)}$	The sample-node vector in the k-th layer.
$\overrightarrow{spat}^{(k)}$	The sample-attribute vector in the k-th layer.
$\overrightarrow{spth}^{(k)}$	The sample-threshold vector in the k-th layer.
$\overrightarrow{spth}_i$	The sample-threshold vector computed with only the i-th attribute.
MinValue	The minimum value on the ring $\mathbb{Z}_{2^k}$

The protocol *TestSamples* used to compute the comparison results between samples and the split threshold is shown in Protocol 8. The parties in this protocol input m secret-shared attribute vectors, $\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}$, a secret-shared sample-attribute vector $\langle \overrightarrow{spat} \rangle_{2^k}$, and a secret-shared sample-threshold vector $\langle \overrightarrow{spth} \rangle_{2^k}$. They get a secret-shared comparison result vector $\langle \vec{b} \rangle_{2^k}$ as the output. Initially, the parties determine the attribute for testing the samples based on $\langle \overrightarrow{spat} \rangle_{2^k}$ and place the corresponding attribute values in a secret-shared vector $\langle \vec{x} \rangle_{2^k}$ (Line 1-4). Subsequently, the parties compute the secret-shared comparison results by comparing the secret-shared vector $\langle \vec{x} \rangle_{2^k}$ with the secret-shared sample-threshold vector $\langle \overrightarrow{spth} \rangle_{2^k}$ (Line 5).

Protocol 8 *TestSamples*

Input: m secret-shared attribute vectors, $\langle \vec{a}_0 \rangle_{2^k}, \cdots, \langle \vec{a}_{m-1} \rangle_{2^k}$, a secret-shared sample-attribute vector, $\langle \overrightarrow{spat} \rangle_{2^k}$,and a secret-shared sample-threshold vector, $\langle \overrightarrow{spth} \rangle_{2^k}$.

Output: A secret-shared comparison result vector $\langle \vec{b} \rangle_{2^k}$.

1: **for** each $i \in [0, m-1]$ in parallel **do**
2: $\quad \langle \vec{eq}_i \rangle_{2^k} = (\langle \overrightarrow{spat} \rangle_{2^k} \stackrel{?}{=} i)$.
3: **end for**
4: $\langle \vec{x} \rangle_{2^k} = \sum_{i=0}^{m-1} \langle \vec{a}_i \rangle_{2^k} * \langle \vec{eq}_i \rangle_{2^k}$.
5: $\langle \vec{b} \rangle_{2^k} = \langle \vec{x} \rangle_{2^k} < \langle \overrightarrow{spth} \rangle_{2^k}$.

Note that this protocol is also the same as the protocol proposed by Hamada et al. [18].

The protocol *TrainLeafLayer* used to train a leaf layer is shown in Protocol 9. The parties in this protocol input an integer k that denotes the height of the leaf layer, a secret-shared permutation $\langle \vec{\pi} \rangle_{2^k}$, a secret-shared sample-node vector $\langle \overrightarrow{spnd} \rangle_{2^k}$, and a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$. The parties get a secret-shared leaf layer $\langle layer \rangle_{2^k}$ as the output.

The protocol *TrainLeafLayer* consists of three stages: (1) The first stage (Line 1-4): the parties compute the secret-shared group flag vector $\langle \vec{g} \rangle_{2^k}$ using the same way in the protocol *TrainInternalLayer* (Protocol 4). (2) The second stage (Line 5-9): the parties compute the secret-shared sample-label vector $\langle \overrightarrow{splb} \rangle_{2^k}$. The equation $\overrightarrow{splb}[j] = i$ represents that the predicted label of the node to which the j-th sample belongs is i. As the predicted label of a leaf node is the most common one of the samples belonging to it, the parties count the number of each label in each node by calling the protocol *GroupSum* (introduced in Section 2.3.5) (Line 5-8). Then the parties determine which label is the most common one by calling the protocol *VectMax* (introduced in Section 2.3.3) (Line 9). (3) The third stage (Line 9): the parties call the protocol *FormatLayer* (Protocol 10) to get the secret-shared layer. The protocol *FormatLayer* is used to remove the redundant values from the input vectors to leave only one nid and one predicted label for each node.

Protocol 9 *TrainLeafLayer*

Input: An integer k that represents the height of the leaf layer, a secret-shared permutation $\langle \vec{\pi} \rangle_{2^k}$, a secret-shared sample-node vector $\langle \overrightarrow{spnd} \rangle_{2^k}$, and a secret-shared label vector $\langle \vec{y} \rangle_{2^k}$.

Output: A secret-shared leaf layer $\langle layer \rangle$.

1: $\langle \vec{y'} \rangle_{2^k} = ApplyPerm(\langle \vec{\pi} \rangle_{2^k}, \langle \vec{y} \rangle_{2^k})$.
2: $\langle \overrightarrow{spnd'} \rangle_{2^k} = ApplyPerm(\langle \vec{\pi} \rangle_{2^k}, \langle \overrightarrow{spnd} \rangle_{2^k})$.
3: $\langle \vec{g}[0] \rangle_{2^k} = \langle 1 \rangle_{2^k}$
4: $\langle \vec{g}[j] \rangle_{2^k} = (\langle \overrightarrow{spnd'}[j-1] \rangle_{2^k} \neq \langle \overrightarrow{spnd'}[j] \rangle_{2^k})$ for each $j \in [1, n-1]$.
5: **for** each $i \in [0, v-1]$ in parallel **do**
6: $\quad \langle \vec{y}_i \rangle_{2^k} = \langle \vec{y'} \rangle_{2^k} \stackrel{?}{=} i$.
7: $\quad \langle \overrightarrow{cnt}_i \rangle_{2^k} = GroupSum(\langle \vec{g} \rangle_{2^k}, \langle \vec{y}_i \rangle_{2^k})$.
8: **end for**
9: $\langle \overrightarrow{splb}[j] \rangle_{2^k} = VectMax([\langle \overrightarrow{cnt}_0[j] \rangle_{2^k}, \cdots, \langle \overrightarrow{cnt}_{v-1}[j] \rangle_{2^k}], [0, \cdots, v-1])$ for each $j \in [0, n-1]$.
10: $\langle layer \rangle_{2^k} = FormatLayer(k, \langle \vec{g} \rangle_{2^k}, \langle \overrightarrow{spnd'} \rangle_{2^k}, \langle \overrightarrow{splb} \rangle_{2^k})$.

The protocol *FormatLayer* used to format a layer is shown in Protocol 10. The parties in this protocol input an integer k that

Protocol 10 *FormatLayer*

Input: An integer k that represents the height of the layer required to be formatted, a secret-shared group flag vector $\langle \vec{g} \rangle_{2^\ell}$, and c secret-shared vectors $\langle \vec{w}_0 \rangle_{2^\ell}, \cdots, \langle \vec{w}_{c-1} \rangle_{2^\ell}$ to be formatted.
Output: c formatted secret-shared vectors $\langle \vec{u}_0 \rangle_{2^\ell}, \cdots, \langle \vec{u}_{c-1} \rangle_{2^\ell}$.

1: $\langle \vec{\alpha} \rangle_{2^\ell} = GenPermFromBit(1 - \langle \vec{g} \rangle_{2^\ell})$.
2: **for** each $i \in [0, c-1]$ in parallel **do**
3: $\quad \langle \vec{u}_i \rangle_{2^\ell} = ApplyPerm(\langle \vec{\alpha} \rangle_{2^\ell}, \langle \vec{w}_i \rangle_{2^\ell})$.
4: $\quad \langle \vec{u}_i \rangle_{2^\ell} = \langle \vec{u}_i[0 : min(2^k, n)] \rangle_{2^\ell}$. // retain the first $min(2^k, n)$ elements of $\langle \vec{u}_i \rangle_{2^\ell}$.
5: **end for**

denotes the height of the current layer required to be formatted, a secret-shared group flag vector $\langle \vec{g} \rangle$, and c secret-shared vectors $\langle \vec{w}_0 \rangle, \cdots, \langle \vec{w}_{c-1} \rangle$. The parties get c formatted secret-shared vectors $\langle \vec{u}_0 \rangle, \cdots, \langle \vec{u}_{c-1} \rangle$ as the output. The protocol *FormatLayer* aims to remove redundant values from the given vectors. The given vectors include the secret-shared sample-node vector, the secret-shared sample-attribute vector, the secret-shared sample-threshold vector, and the secret-shared label vector, whose elements are identical within each node. Thus it suffices to retain only one element per node. To achieve this, the parties move the first element of each group to the front of the vectors and retain only the $min(2^k, n)$ elements, as the k-th layer has at most 2^k nodes. As a result of this operation, the output decision tree is typically a complete binary tree, except when the number of training samples is fewer than the node number in some layers of a complete binary tree.

C EXAMPLE OF TRAINING STEPS

As is shown in Figure 3, we illustrate the key steps of the training process. The example considers a training dataset containing nine samples, each with one label and two attributes.

In the initial stage: the parties generate secret-shared permutations from the attribute vectors. These permutations could be directly used to sort the attributes and labels according to the corresponding attributes within each node of the 0-th layer. That is because this layer contains only a single node, with all samples belonging to it. Additionally, the parties initialize a secret-shared sample-node vector $\langle \overrightarrow{spnd}^{(0)} \rangle$.

In the stage of training the 0-th layer: the parties first apply one of the secret-shared permutations (without loss of generality, $\langle \vec{\pi}_0 \rangle_{2^\ell}$ is chosen) to $\langle \overrightarrow{spnd}^{(0)} \rangle$, resulting in a new sample-node vector $\langle \overrightarrow{spnd'} \rangle$. This step ensures that elements with the values in $\langle \overrightarrow{spnd'} \rangle$ are stored consecutively. Note that this step could be omitted in the 0-th layer, since the 0-th layer only contains one node. The parties then use $\langle \overrightarrow{spnd'} \rangle$ to compute the secret-shared group flag vector $\langle \vec{g} \rangle$. Following this, they apply the secret-shared permutations to the secret-shared attribute vectors and the secret-shared label vector, obtaining new secret-shared attribute vectors and a new secret-shared label vector, whose elements are sorted according to the corresponding attribute vectors within each node. The parties proceed to compute the modified Gini impurity and the split thresholds of each split point for each attribute. Note that the communication optimization based on our proposed share conversion protocol is performed during the computation for the modified Gini impurity. Subsequently, the parties find a maximum modified Gini impurity and its corresponding threshold for each attribute by calling the protocol *GroupMax* (introduced in Section 2.3.5). Then, the parties find the split attribute index and the split threshold with a maximum modified Gini impurity across the attributes. Finally, the parties align the element positions of $\langle \overrightarrow{spat} \rangle$ and $\langle \overrightarrow{spth} \rangle$ with those in the original attribute vectors by calling the protocol *UnApply* (introduced in Section 2.3.4).

In the update stage: the parties compute the comparison results between the samples and the split thresholds. Based on these results, the parties compute the secret-shared sample-node vector of the next layer and update the secret-shared permutations for training the next layer.

D ENTS IN TWO-PARTY SCENARIOS

D.1 Share Conversion Protocol for Two-Party Scenarios

The protocol *ConvertShareTwoParty* designed for share conversion in two-party scenarios is shown in Protocol 11. Here, we use the notation $[\![x]\!]$ to denote the additive shares of a secret x, i.e. P_0 holds $[\![x]\!]^0$ and P_1 holds $[\![x]\!]^1$. The parties in this protocol input an additive share $[\![x]\!]_{2^k}$ on $\mathbb{Z}_{2^k}$ and two public integers k and ℓ ($k < \ell$), and they get a new additive share $[\![x']\!]_{2^\ell}$ on $\mathbb{Z}_{2^\ell}$ as the output. The implementation is similar to the three-party version, with only differences in computations of d_0, d_1, b_0, and b_1. Concretely, in this protocol, P_0 locally computes $d_0 = [\![x]\!]^0_{2^k}$ and $b_0 = (truncd_0 \wedge 1) \oplus [\![r]\!]^0_2$. P_1 locally computes $d_1 = [\![x]\!]^1_{2^k}$ and $b_1 = (truncd_1 \wedge 1) \oplus [\![r]\!]^1_2$. while in the protocol *ConvertShare*, P_0 locally computes $d_0 = ([\![x]\!]^0_{2^k} + [\![x]\!]^1_{2^k}) \bmod 2^k$ and $b_0 = (truncd_0 \wedge 1) \oplus [\![r]\!]^0_2 \oplus [\![r]\!]^1_2$. P_1 locally computes $d_1 = [\![x]\!]^2_{2^k}$ and $b_1 = (truncd_1 \wedge 1) \oplus [\![r]\!]^2_2$.

We analyze the security of this protocol in Appendix E.

Protocol 11 *ConvertShareTwoParty*

Input: $[\![x]\!]_{2^k}$ ($x \in [0, 2^{k-1})$), i.e. P_i inputs $[\![x]\!]^i_{2^k}$, and two public integers k and ℓ ($k < \ell$).
Output: $[\![x']\!]_{2^\ell}$, i.e. P_i gets $[x']^i_{2^\ell}$, with $x' = x$.
Offline: P_0, P_1 generate a daBit together, so each P_i holds $[\![r]\!]^i_2$, and $[\![r]\!]^i_{2^\ell}$ ($r = 0$ *or* 1).

1: P_0 locally computes $d_0 = [\![x]\!]^0_{2^k}$. P_1 locally computes $d_1 = [\![x]\!]^1_{2^k}$.
2: P_0 locally computes $truncd_0 = d_0 \gg (k-1)$. P_1 locally computes $truncd_1 = -((-d_1) \gg (k-1))$.
3: P_0 shares d_0 and $truncd_0$ on $\mathbb{Z}_{2^\ell}$. P_1 shares d_1 and $truncd_1$ on $\mathbb{Z}_{2^\ell}$.
4: $[\![d]\!]_{2^\ell} = [\![d_0]\!]_{2^\ell} + [\![d_1]\!]_{2^\ell}$.
5: $[\![truncsum]\!]_{2^\ell} = [\![truncd_0]\!]_{2^\ell} + [\![truncd_1]\!]_{2^\ell}$.
6: P_0 locally computes $b_0 = (truncd_0 \wedge 1) \oplus [\![r]\!]^0_2$. P_1 locally computes $b_1 = (truncd_1 \wedge 1) \oplus [\![r]\!]^1_2$.
7: P_0 sends b_0 to P_1. P_1 and sends b_1 to P_0.
8: The parties both locally compute $b = b_0 \oplus b_1$.
9: $[\![bit]\!]_{2^\ell} = b + [\![r]\!]_{2^\ell} - 2 * b * [\![r]\!]_{2^\ell}$.
10: $[\![ovfl]\!]_{2^\ell} = ([\![truncsum]\!]_{2^\ell} - [\![bit]\!]_{2^\ell}) * 2^{k-1}$.
11: $[\![x']\!]_{2^\ell} = [\![d]\!]_{2^\ell} - [\![ovfl]\!]_{2^\ell}$.

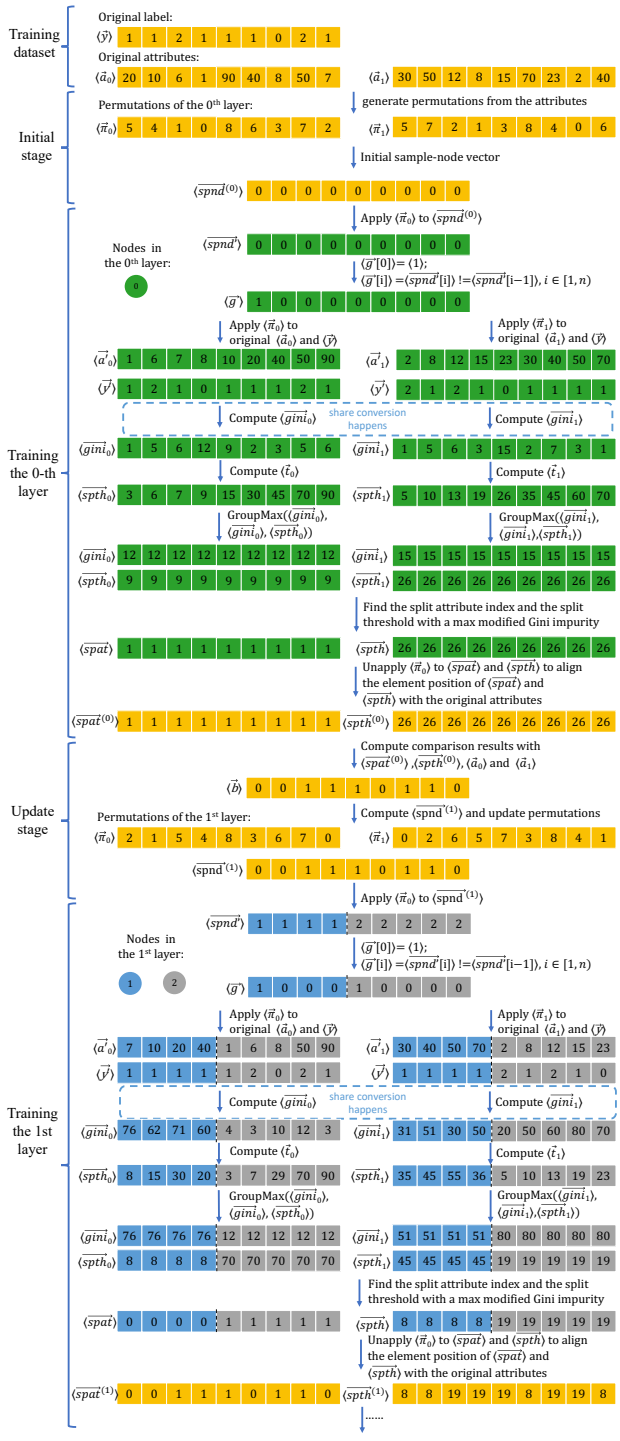

Figure 3: An example to show the key steps to train a decision tree with height more than two. Vectors colored orange indicate that elements are stored in their original positions. Vectors colored green, blue, and gray indicate that elements belonging to the same nodes are stored consecutively. Additionally, we use simulated values for the modified Gini vectors, because the real values contain too many digits.

D.2 Experimental Results of Two-Party Ents

We compare the efficiency of two-party Ents with two state-of-the-art two-party frameworks [3, 31]. (1) Liu et al.'s framework [31] is based on secret sharing and homomorphic encryption. It offers three privacy levels. Higher privacy levels provide stronger privacy guarantees. We compare two-party Ents with the framework [31] at its highest privacy level, as two-party Ents provides stronger privacy guarantees. Specifically, even at the highest privacy level, the framework [31] still leaks Gini impurity and the decision tree structure, while two-party Ents does not leak any private information during the training process. Since this framework is not open-sourced, we rely on the experimental results, which are performed on two laptops with 4-core 2.6GHz Intel Core i7-6700HQ CPUs and 8GB of RAM, reported in their paper [31]. (2) Akavia et al.'s framework [3] is based on the CKKS homomorphic encryption scheme [9] implemented in Microsoft SEAL v3.3.2 [42]. The framework involves a client and a server, with the client outsourcing encrypted data to the server to train decision trees. As this framework is also not open-sourced, we use the experimental results, which were conducted on an AWS x1.16xlarge server equipped with 32-core server 2.3GHZ and 976GB of RAM, from their paper [3].

In terms of hardware settings, our setup is worse than that used in Akavia et al.'s framework [3]. When compared to the framework by Liu et al.[31], our core usage is the same as theirs, with each party utilizing up to 4 cores. Although our machines have more memory, the actual memory usage for evaluating the three datasets (Kohkiloyeh, Diagnosis, and Tic-tac-toe) in two-party Ents does not exceed 200 MB of RAM.

Results. We show the experimental results in Table 7. Note that the literatures [3, 31] only report the experimental results obtained in the LAN setting (their concrete RTT and bandwidth are unknown). We consider the experimental results of the framework [3] in the LAN setting to be representative of the WAN setting as well, because the framework [3] is totally based on homomorphic encryption and its performance is usually unaffected by the network setting. The experimental results show that two-party Ents significantly outperforms the framework [31] by $14.3\times \sim 36.9\times$ in the LAN setting, and outperforms the framework [3] by $338.2\times \sim 1,362.3\times$ in the LAN setting and by $4.7\times \sim 6.4\times$ in the WAN setting. The improvement is primarily due to two facts: (1) the computations of two-party Ents are totally performed based on additive secret sharing, which is much more computation efficient compared to homomorphic encryption. Such a technical route makes two-party Ents more efficient than the two frameworks [3, 31], especially in the LAN setting. (2) with the two communication optimizations, the communication overhead of Ents should be low, ensuring its efficiency even in the WAN setting.

E SECURITY ANALYSIS FOR CONVERSION PROTOCOLS

We analyze the security of the protocols *ConvertShare* (Protocol 6) and *TwoPartyConvertShare* (Protocol 11) using the standard real/ideal world paradigm.

Table 7: Online training time (seconds) of two-party Ents vs. the two-party frameworks [3, 31] for training a decision tree. Since the tree height for the framework [31] is not specified, we trained a decision tree of height six for Ents on the Kohkiloyeh, Diagnosis, and Tic-tac-toe datasets. For the datasets Iris, Wine, and Cancer, we matched the tree height and sample number with those used in the literature [3]. Concretely, the tree height for the three datasets is set to four, and the sample number is set to 100, 119, and 381, respectively. The notation '-' indicates the absence of experimental results.

Dataset	LAN			WAN	
	Ents	[31]	[3]	Ents	[3]
Kohkiloyeh	**4.15** (36.99×)	153.53	-	**943.70**	-
Diagnosis	**4.26** (14.31×)	61	-	**943.91**	-
Tic-tac-toe	**56.52** (23.67×)	1,338	-	**2,056.0**	-
Iris	**2.07** (1, 362.31×)	-	2,820	**435.40** (6.47×)	2,820
Wine	**7.87** (1, 128.33×)	-	8,880	**1,674.91** (5.30×)	8,880
Cancer	**49.31** (338.26×)	-	16,680	**3,533.82** (4.72×)	16,680

Proof. Let the semi-honest adversary $\mathcal{A}$ corrupt at most one party, we now present the steps of the idea-world adversary (simulator) $\mathcal{S}$ for $\mathcal{A}$. Our simulator $\mathcal{S}$ for *ConvertShare* (Protocol 6), and *ConvertShareTwoParty* (Protocol 11) is constructed as follows:

Security for *ConvertShare* (Protocol 6): For the offline phase, we use the generation process of the daBit [13] in a black-box manner. Therefore, the simulation follows the same in the study [13]. For line 3 and 7 in Protocol 6, which are the only two steps that require interaction between parties, we analyze it case by case: (1) If $\mathcal{A}$ corrupts P_0, $\mathcal{S}$ receives $[\![d_0]\!]^1_{2^\ell}$, $[\![d_0]\!]^2_{2^\ell}$, $[\![truncd_0]\!]^1_{2^\ell}$, $[\![truncd_0]\!]^2_{2^\ell}$ and b_0, from $\mathcal{A}$ on behalf of P_1, and receives $[\![d_0]\!]^2_{2^\ell}$, $[\![d_0]\!]^0_{2^\ell}$, $[\![truncd_0]\!]^2_{2^\ell}$, $[\![truncd_0]\!]^0_{2^\ell}$ and b_0, from $\mathcal{A}$ on behalf of P_2. Then $\mathcal{S}$ selects random values to simulate $[\![d_1]\!]^0_{2^\ell}$, $[\![d_1]\!]^1_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^1_{2^\ell}$ and b_1, and sends them to $\mathcal{A}$ on behalf of P_1. (2) If $\mathcal{A}$ corrupts P_1, $\mathcal{S}$ receives $[\![d_1]\!]^0_{2^\ell}$, $[\![d_1]\!]^1_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^1_{2^\ell}$ and b_1, from $\mathcal{A}$ on behalf of P_0, and receives $[\![d_1]\!]^2_{2^\ell}$, $[\![d_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^2_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$ and b_1, from $\mathcal{A}$ on behalf of P_2. Then $\mathcal{S}$ selects random values to simulate $[\![d_1]\!]^0_{2^\ell}$, $[\![d_1]\!]^1_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^1_{2^\ell}$ and b_1, and sends them to $\mathcal{A}$ on behalf of P_0. (3) If $\mathcal{A}$ corrupts P_2, $\mathcal{S}$ selects random values to simulate $[\![d_1]\!]^0_{2^\ell}$, $[\![d_1]\!]^1_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^1_{2^\ell}$ and b_1, and sends them to $\mathcal{A}$ on behalf of P_0. Besides, $\mathcal{S}$ selects random values to simulate $[\![d_1]\!]^0_{2^\ell}$, $[\![d_1]\!]^1_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^1_{2^\ell}$ and b_1, and sends them to $\mathcal{A}$ on behalf of P_1.

Security for *ConvertShareTwoParty* (Protocol 11): For the offline phase, we use the generation process of the daBit [13] in a black-box manner. Therefore, the simulation follows the same in the study [13]. For line 3 and 7 in Protocol 6, which are the only two steps that require interaction between parties, we analyze it case by case: (1) If $\mathcal{A}$ corrupts P_0, $\mathcal{S}$ receives $[\![d_0]\!]^1_{2^\ell}$, $[\![truncd_0]\!]^1_{2^\ell}$ and b_0, from $\mathcal{A}$ on behalf of P_1. Then $\mathcal{S}$ selects random values to simulate $[\![d_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$ and b_1, and sends them to $\mathcal{A}$ on behalf of P_1. (2) If $\mathcal{A}$ corrupts P_1, $\mathcal{S}$ receives $[\![d_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$ and b_1, from $\mathcal{A}$ on behalf of P_0. Then $\mathcal{S}$ selects random values to simulate $[\![d_1]\!]^0_{2^\ell}$, $[\![truncd_1]\!]^0_{2^\ell}$ and b_1, and sends them to $\mathcal{A}$ on behalf of P_0.

Since all the messages sent and received in the protocol are uniformly random values in both the real protocol and the simulation, the $\mathcal{A}$'views in the real and ideal worlds are both identically distributed and indistinguishable. This concludes the proof. □

F DETAILS OF COMMUNICATION COMPLEXITY ANALYSIS

We first present the communication complexity of basic primitives introduced in Section 2.3 and Appendix **??**.

- *Basic Operations.* Assuming the basic operations are performed on a ring $\mathbb{Z}_{2^k}$, their communication complexity is as follows: (1) Secure addition with constant, secure multiplication with constant, and secure addition usually can be performed without communication. (2) Secure multiplication and secure probabilistic truncation both usually require a communication size of $O(k)$ in $O(1)$ communication rounds. (3) Secure comparison and equality test usually require a communication size of $O(k \log k)$ in $O(k)$ communication rounds. (4) Secure division usually requires a communication size of $O(k \log f)$ in $O(\log f)$ communication rounds, where f is the bit length of the decimal part and set to be $2\lceil \log n \rceil$ in our evaluation.
- *Vector Max Protocol Based on RSS.* Assuming the protocol *VectMax* [18] is performed on a ring $\mathbb{Z}_{2^k}$ and its input secret-shared vectors are of size n, it requires a communication size of $O(nk)$ bits in $O(\log n)$ communication rounds.
- *Secure Radix Sort Protocols Based on RSS.* Assuming the secure radix sort protocols [10] Based on RSS are all performed on a ring $\mathbb{Z}_{2^k}$ and their input secret-shared vectors are of size n, the communication complexity of the protocols is as follows: the protocols *GenPerm* and *BitVecDec* both require a communication size of $O(nk^2)$ bits in $O(k)$ communication rounds. The protocols *GenPermByBit*, *ApplyPerm*, *ComposePerms* and *UnApplyPerm* all require a communication size of $O(nk)$ bits in $O(1)$ communication rounds.
- *Group-Wise Protocols Based on RSS.* Assuming group-wise protocols [18] Based on RSS are all performed on a ring $\mathbb{Z}_{2^k}$ and their input secret-shared vectors are of size n, the communication complexity of the protocols is as follows: the protocols *GroupSum* and *GroupPrefixSum* both require a communication size of $O(nk)$ bits in $O(1)$ communication rounds. The protocol *GroupMax* and its extension version require a communication size of $O(n \log nk \log k)$ bits in $O(\log n \log k)$ communication rounds [5].

Next, we analyze the online communication complexity for each training protocol proposed by us as follows:

- *TrainDecisionTree* (Protocol 2): In this protocol, the parties first call the protocol *GenPerm* (protocol 1) m times in parallel, which

[5]We show the communication complexity of optimized version of the group-wise protocols implemented in MP-SPDZ [25], which is used to implement Ents. Therefore, the communication complexity is inconsistent with the complexity in the literature [18].

requires a communication size of $O(mn\ell^2)$ bits in $O(\ell)$ communication rounds. Next, the parties call the protocol *TrainInternalLayer* (protocol 4) h times sequentially, which requires a communication size of $O(hmn \log n\ell \log \ell)$ bits in $O(h \log n \log \ell)$ communication rounds. Finally, the parties call the protocol *TrainLeafLayer* (protocol 9), which requires a communication size of $O(vn\ell)$ bits in $O(\log v)$ communication rounds. Therefore, the protocol *TrainDecisionTree* requires a communication size of $O(hmn \log n\ell \log \ell + mn\ell^2)$ bits in $O(h \log n \log \ell + \ell)$ communication rounds

- *UpdatePerms* (Protocol 3): In this protocol, the parties mainly call the protocols *ApplyPerm* (introduced in Section 2.3.4), *GenPermByBit* (introduced in Section 2.3.4), and *ComposePerms* (introduced in Section 2.3.4) m times in parallel. Therefore, the protocol *UpdatePerms* requires a communication size of $O(mn\ell)$ bits in $O(1)$ communication rounds.
- *TrainInternalLayer* (Protocol 4): In this protocol, the parties mainly call the protocol *AttributeWiseSplitSelection* (protocol 7) m times in parallel, which requires a communication size of $O(mn \log n \ell \log \ell)$ bits in $O(\log n \log \ell)$ communication rounds, and call the protocol *VectMax* (introduced in section 2.3.3) n times in parallel, which requires a communication size of $O(nm\ell)$ bits in $O(\log m)$ communication rounds. Since m is typically smaller than n, the protocol *TrainInternalLayer* requires a communication size of $O(mn \log n\ell \log \ell)$ bits in $O(\log n \log \ell)$ communication rounds.
- *ComputeModifiedGini* (Protocol 5): In this protocol, the parties mainly perform multiplication v times in parallel, which requires a communication size of $O(nv\ell)$ bits in $O(1)$ communication rounds, and perform secure division operation one time, which requires a communication size of $O(n \log n\ell)$ bits in $O(\log n)$ communication rounds. Therefore, the protocol *ComputeModifiedGini* requires a communication size of $O(nv\ell + n\ell \log \ell)$ bits in $O(\log n)$ communication rounds.
- *ConvertShare* (Protocol 6): In this protocol, P_0 and P_1 collectively share four numbers on $\mathbb{Z}_{2^\ell}$ and transmit four bits. Note that, the sharing process and transmitting process can be performed in the same communication round. Besides, using the sharing method proposed in the study [35], sharing a single number on $\mathbb{Z}_{2^\ell}$ only requires transmitting ℓ bits. Therefore, *ConvertShare* requires a communication size of $4\ell + 4$ bits in one online communication round.
- *AttributeWiseSplitSelection* (Protocol 7): In this protocol, the parties mainly call the protocol *ComputeModifiedGini* (Protocol 5), which requires a communication size of $O(nv\ell + n\ell \log \ell)$ bits in $O(\log n)$ communication rounds, and call the protocol *GroupMax* (introduced in section 2.3.5), which requires a communication size of $O(n \log n\ell \log \ell)$ bits in $O(\log n \log \ell)$ communication rounds. Therefore, the protocol *AttributeWiseSplitSelection* requires a communication size of $O(n \log n\ell \log \ell)$ bits in $O(\log n \log \ell)$ communication rounds.
- *TestSamples* (Protocol 8): In this protocol, the parties operate on m secret-shared vectors in parallel. Therefore, the protocol *TestSamples* requires a communication size of $O(mn\ell)$ bits in $O(1)$ communication rounds.
- *TrainLeafLayer* (Protocol 9): In this protocol, the parties mainly call the protocol *ApplyPerm* (introduced in Section 2.3.4) two times, which requires a communication size of $O(n\ell)$ bits in $O(1)$ communication rounds, call the protocol *GroupSum* (introduced in Section 2.3.5) v times in parallel, which requires a communication size of $O(vn\ell)$ bits in $O(1)$ communication rounds, call the protocol *VectMax* (introduced in Section 2.3.3) once, which requires a communication size of $O(vn\ell)$ bits in $O(\log v)$ communication rounds, and call the protocol *FormatLayer* (Protocol 10) one time, which requires a communication size of $O(n\ell)$ bits in $O(1)$ communication rounds. Therefore, the protocol *TrainLeafLayer* requires a communication size of $O(vn\ell)$ bits in $O(\log v)$ communication rounds.
- *FormatLayer* (Protocol 10): In this protocol, the parties mainly call the protocol *GenPermFromBit* (introduced in Section 2.3.4) once, which requires a communication size of $O(n\ell)$ bits in $O(1)$ communication rounds, and call the protocol *ApplyPerm* (introduced in Section 2.3.4) c times, which requires a communication size of $O(cn\ell)$ bits in $O(1)$ communication rounds. Since c is a small constant representing the number of vectors to be formatted, we disregard the factor c. Therefore, the protocol *FormatLayer* requires a communication size of $O(n\ell)$ bits in $O(1)$ communication rounds.

G PROOF OF THEOREM 1

We prove Theorem 1 as follows.

Proof. We define

$$d = d'' * 2^c + d', d'' \in [0, 2^{\ell+1-c}), d' \in [0, 2^c)$$

$$d_0 = d_0'' * 2^c + d_0', d_0'' \in [0, 2^{\ell-c}), d_0' \in [0, 2^c)$$

Since $d_0 + d_1 = d$, then $d_1 = d - d_0$. Since $d_0 \in [0, 2^\ell]$ and $c < \ell < \ell - 1$, then $d \gg c = d''$.

We prove the theorem case by case.

(1) If $d_0' \geq d'$: $d_0 - d = (d_0'' - d'') * 2^c + (d_0' - d')$. Then $(d_0 - d) \gg c = ((d_0'' - d'') * 2^c + (d_0' - d')) \gg c = (((d_0'' - d'') * 2^c) \gg c) + ((d_0' - d') \gg c) = d_0'' - d''$. Thus, $(d_0 \gg c) + (-((-d_1) \gg c)) = d_0'' + (-((d_0 - d) \gg c)) = d_0'' + (-(d_0'' - d'')) = d''$

(2) If $d_0' < d'$: $d_0 - d = (d_0'' - d'' - 1) * 2^c + (d_0' + 2^c - d')$. Then $(d_0 - d) \gg c = ((d_0'' - d'' - 1) * 2^c + (d_0' + 2^c - d')) \gg c = (((d_0'' - d'' - 1) * 2^c) \gg c) + ((d_0' + 2^c - d') \gg c) = d_0'' - d'' - 1$. Thus, $(d_0 \gg c) + (-((-d_1) \gg c)) = d_0'' + (-((d_0 - d) \gg c)) = d_0'' + (-((d_0'' - d'' - 1) \gg c)) = d_0'' + (-(d_0'' - d'' - 1)) = d'' + 1$

In summary, $(d_0 \gg c) + (-((-d_1) \gg c)) = \lfloor d/2^c \rfloor + bit$, where $bit = 0$ *or* 1

□