Argmax and XGBoost Training over Fully Homomorphic Encryption

Ramy Masalha
IBM Research and University of Haifa
Haifa, Israel
ramymassalha@gmail.com

Adi Akavia
University of Haifa
Haifa, Israel
akavia@cs.haifa.ac.il

Allon Adir
IBM Research
Haifa, Israel
adir@il.ibm.com

Ehud Aharoni
IBM Research
Haifa, Israel
aehud@il.ibm.com

Eyal Kushnir
IBM Research
Haifa, Israel
Eyal.Kushnir@ibm.com

Abstract

Fully Homomorphic Encryption (FHE) is a promising solution to enable privacy-preserving inference and training of machine learning models over encrypted data. Among the machine learning methods used in practice, Extreme Gradient Boosting (XGBoost) is one technique that shines in many applications. While previous works have tackled the problem of training tree-based models over FHE, these works either rely on interaction with the client, which adds the extra burden of communication, or consume a typically unreasonable amount of time to train a large model. In this work, we present an efficient system for a non-interactive XGBoost training over FHE that achieves up to 360× speedup compared to the state of the art.

The argmax operation is a basic building block invoked repeatedly during the XGBoost training as well as other machine learning algorithms, but computing it over FHE is time consuming. When utilizing the Single Instruction Multiple Data (SIMD) parallelism capability offered by most FHE schemes and using a configuration with s slots, the state of the art methods compute argmax on $n \leq s$ values using either $O(\log_2 n)$ SIMD-comparisons in tournament-style comparison or $\lceil \frac{n^2}{s} \rceil$ SIMD-comparisons using all pairs comparison.

As a second contribution of this work, we propose an efficient argmax algorithm that is based on a novel technique to maximize SIMD-utilization, and computes the argmax of $n \leq s$ values using only $O(\log_2 \log_2 n)$ SIMD-comparisons. The method extends to $n > s$ with complexity $O(\frac{n}{s} + \log_2 \log_2 s)$, compared to $O(\frac{n}{s} + \log_2 s)$ for state of the art methods. We conduct empirical experiments to compare our method with other existing argmax methods, and show that when using the HEaaN FHE scheme with a configuration of $s = 2^{15}$ to compute the argmax of $n = s$ values, our implementation is about 1.6× faster than the state of the art.

Keywords

fully homomorphic encryption, privacy preserving machine learning, XGBoost, argmax

Proceedings on Privacy Enhancing Technologies 2026(1), 175–200

https://doi.org/10.56553/popets-2026-0010

1 Introduction

Fully Homomorphic Encryption (FHE) schemes allow computations to be performed on encrypted data without the need for decryption. A user who has confidential data may encrypt the data using FHE and deploy it to a cloud service for further computations. The entire computation process in the cloud is applied while the data is encrypted, which enables taking advantage of the cloud's high computation power while preserving the privacy of the sensitive data. Among the most promising applications of FHE is its integration with machine learning (ML), enabling privacy-preserving model inference and training.

Among the machine learning methods used in practice, Extreme Gradient Boosting (XGBoost) [19] is one technique that shines in many applications. XGBoost is an efficient and scalable gradient boosting algorithm that constructs multiple decision trees sequentially, optimizing each tree based on the errors of the previous ones. It achieves state-of-the-art performance on various machine learning tasks and has been dominating applied machine learning competitions as well as real-world applications (e.g. [55], [10], [56], [1], [44]).

While previous works (e.g. [6], [51]) have tackled the problem of training tree-based models over FHE, these works either rely on interaction with the client, which adds the extra burden of communication, or consume a typically unreasonable amount of time to train a large model. For instance, the current fastest work implementing a non-interactive XGBoost training over FHE [51] reported that it took them about 7 hours to train a single tree of depth 4 on the sepsis dataset [23]. By extrapolating this information we estimate that it would take more than 100 days to train a typical model of 100 trees of depth 6.

One operation that is repeatedly invoked during the XGBoost training algorithm, and which is of an independent interest in this work, is the argmax operation. Given an input vector of real values, the argmax operation locates the index of the maximal value of the input vector. While trivially computed over cleartext data, argmax becomes latency-intensive when operating over FHE. The argmax component is a basic building block in XGBoost training as well as in many other machine learning applications and hence enhancing its performance is of high importance.

There are two common approaches to compute argmax in the state of the art: the tournament method and the league method. The tournament method (e.g. [18, 24, 36, 60, 61]) finds the maximum

of n values using $\log_2 n$ tournament rounds. In each round, the remaining values are divided into pairs. The maximal value within each pair proceeds to the next round while the other value gets eliminated. The algorithm stops when only one element is left, representing the maximum of the array. The league method (e.g. [30, 32, 36, 52]), on the other hand, relies on performing all possible pairs of comparisons to obtain the maximum.

To reduce the latency of argmax (and FHE computations more generally), schemes supporting Single Instruction Multiple Data (SIMD) operations are often used. These schemes support ciphertexts that have multiple *slots*, such that each slot encrypts a separate value, and basic operations, such as multiplication, rotation and addition, are applied on all slots of a ciphertext concurrently. Maximizing the utilization of the ciphertexts' slots is crucial in the design of algorithms that operate over FHE.

Since comparisons are usually the bottle-neck of argmax algorithms (see Table 2), we measure their complexity in the terms of the number of SIMD-comparisons they apply. Assuming a SIMD FHE scheme that packs s slots in each ciphertext is used, the tournament method takes $\log_2 n$ SIMD-comparisons if $n \leq s$ and $\frac{n}{s} + \log_2(s)$ SIMD-comparisons if $n > s$ (see Section 2.1.1), while the league method takes $\frac{n^2}{s}$ SIMD-comparisons for both cases of $n > s$ and $n \leq s$ (see Section 2.1.2). Moreover, a variant of the tournament algorithm that is explained in Section 2.1.1 takes $\log_4 n$ SIMD-comparisons when $n \leq \frac{2s}{3}$.

These observations raise two research questions addressed in this work:

(1) Can the SIMD capability be leveraged to develop an argmax algorithm over FHE that requires less than $\min(\frac{n}{s} + \log_2 n, \lceil \frac{n^2}{s} \rceil)$ SIMD comparisons?
(2) To what extent may the state of the art method for XGBoost training over FHE be improved, both by speed-up of the argmax component and by other optimizations?

This work answers the first question affirmatively, and shows a substantial improvement in the topic of the second question.

1.1 Our Contribution

This paper presents two key contributions. First, we present a novel argmax algorithm that reduces the number of required SIMD-comparisons by a better utilization of the SIMD capability. Second, we introduce a system for XGBoost training over FHE. Our implementation incorporates the optimized argmax method along with additional system-level enhancements, achieving up to a 360× speedup compared to the state of the art [51]. More specifically, our contributions may be summarized as follows.

- In the case $n \leq s$, our argmax algorithm reduces the number of required SIMD-comparisons from $min(\log_4 n, \lceil \frac{n^2}{s} \rceil)$ to $\log_2 \log_2 n$.
- In the case $n > s$, our algorithm reduces the number of required SIMD-comparisons from $O(\frac{n}{s} + \log_2 s)$ to $O(\frac{n}{s} + \log_2 \log_2 s)$ (see also Table 1).
- We show empirically that when using a common HEaaN CKKS [20] configuration with $s = 2^{15}$ slots and computing the argmax of $n = 2^{15}$ values, our argmax algorithm is faster than the state of the art by a factor of 1.6×.
- We introduce a system for a non-interactive XGBoost training over FHE. Our XGBoost training method incorporates the optimized argmax method along with additional enhancements, achieving up to 360× speedup compared to the state of the art.
- Within the 360× overall speedup, the contribution of our new argmax component is as follows. In the datasets we evaluated, our argmax algorithm achieves up to 1.7 hours of runtime reduction (about 16%) from the total training time of an XGBoost model of 100 trees of depth 6, compared to an XGBoost training system that uses the prior best argmax algorithm.

Follow-up Work. While this work demonstrates the usefulness of our argmax algorithm by integrating it in a system for XGBoost training over FHE, our argmax algorithm is a standalone contribution that can be used in a variety of FHE applications. For example, in a follow-up work [7], Akavia et al. present a similarity based retrieval system over FHE that supports finding the most similar vector to an encrypted query from a dataset of half a million encrypted vectors, while reducing the communication bandwidth by 8× compared to common previous solutions. Their system strongly relies on our argmax component, which demonstrates an additional immediate application of our argmax algorithm.

Paper Organization. The rest of this paper is organized as follows. In Section 2, we compare our work to prior research on argmax and XGBoost training over FHE. In Section 3, we provide an overview of the preliminary material. In Section 4, we introduce our proposed argmax algorithm. In Section 5, we present a system for XGBoost training over FHE that uses our argmax algorithm and in Section 6 we present experimental results.

2 Related Work

This section reviews prior methods for performing argmax over FHE and previous work on training tree-based models over FHE. The former is addressed in Section 2.1, and the latter in Section 2.2. In both parts, we demonstrate that our approach yields a clear improvement over the state of the art.

2.1 Prior Max and Argmax Methods over FHE

Let $\mathbf{X} = \{x_0, x_1, \ldots, x_{n-1}\}$ be a vector of real values. In this section, we discuss state of the art FHE-friendly methods to compute the max and argmax of $\mathbf{X}$. If i is the index of the maximal value in $\mathbf{X}$, then the required output of the max operation is x_i and the required output of the argmax operation is i. However, in the argmax algorithms explained below, the output is an indicator vector of length n that contains 1 in its i-th element and 0 in all other elements. The index i of the maximal element can easily be extracted from this indicator vector.

Table 1 compares the number of SIMD-comparisons required by our algorithm vs. state of the art argmax algorithms explained in this section. Note that the numerical argmax [22] (Section 2.1.4) algorithm is not included in Table 1 because it is not based on comparisons.

2.1.1 Tournament Method. The tournament method is a well-known method to select a "winner" among a set of values. This winner

Table 1: Comparison of our argmax algorithm vs. existing argmax algorithms that rely on comparisons to compute the argmax of n values, assuming a SIMD FHE configuration with s slots is used. See a detailed description of the compared works in Section 2.1.

Algorithm	Complexity ($n > s$)	Complexity ($n \leq s$)
Tournament [18, 24, 36, 60, 61]	$\lceil \frac{n}{s} \rceil - 1 + \log_2 s$	$\log_2 n$
Four-way-tournament [51]	N/A (not applicable)[1]	$\log_4 n$
League [30, 32, 36, 52]	$\lceil \frac{n^2}{s} \rceil$	$\lceil \frac{n^2}{s} \rceil$
Approximate argmax [42][2]	$\lceil \frac{n}{s} \rceil \log_2 n$	$\log_2 n$
Hierarchical argmax (ours)	$\mathbf{\lceil \frac{n}{s} \rceil - 1 + \log_2 \log_2 s}$	$\mathbf{\log_2 \log_2 n}$

[1] [51] is not applicable in this parameter regime because it requires that $n \leq \frac{2}{3}s$.

[2] Assuming their parameter, k, equals $\log_2 n$ so the approximate argmax becomes accurate argmax.

may represent, for example, the maximal value in the set. Using this method to compute max and argmax over FHE was discussed, for example, in [24], [18], [36], [61] and [60].

The tournament method finds the maximum of n values $\{x_0, x_1, \ldots, x_{n-1}\}$ using $\log_2 n$ tournament rounds. In each round, the remaining values are divided into pairs. The maximal value within each pair proceeds to the next round while the other value gets eliminated. The algorithm stops when only one element is left. The remaining element represents the maximum of the array. The argmax indicator vector may be extracted from the comparison results as well. This may be done, for example, by creating after each tournament round a temporary indicator vector that contains 1 in the indexes of the winning values and 0 in the indexes of the losing values. The required final argmax indicator vector can be computed as the product of all temporary indicator vectors.

When n is not greater than the number of slots, s, the tournament method takes $\log_2 n$ SIMD-comparisons: The first round involves $\frac{n}{2}$ independent comparisons which can be done concurrently using one SIMD comparison. The second round involves $\frac{n}{4}$ independent comparisons which again can be done concurrently using one SIMD comparison. In the same manner, after $\log_2 n$ rounds, the tournament is finished while taking $\log_2 n$ SIMD-comparisons in total. When $n > s$, $\lceil \frac{n}{s} \rceil - 1$ SIMD-comparisons are needed to reduce the number of candidate values to s values that fit into one ciphertext, and then $\log_2 s$ other SIMD-comparisons are needed to find the final maximal value among the remaining s candidates, thus taking $O(\frac{n}{s} + \log_2 s)$ SIMD-comparisons in total.

Shin et al. [51] presented an argmax algorithm which is similar to the tournament algorithm, but compares sets of four elements instead of pairs of elements in each round of the tournament. Their algorithm operates under the assumption $n \leq \frac{2}{3}s$ and requires $\log_4 n$ SIMD-comparisons instead of $\log_2 n$ SIMD-comparisons that are needed when using the usual tournament algorithm. Their algorithm utilizes the SIMD capability to perform all 6 pairs of comparisons needed to find the maximum in a set of 4 values while using only one SIMD-comparison.

2.1.2 League Method. The league method relies on performing all possible pairs of comparisons to obtain the maximum of $\mathbf{X} = \{x_0, \ldots, x_{n-1}\}$. To this end, the league algorithm starts by creating a matrix, $A_{n\times n}$, whose i-th row contains n copies of x_i. The transpose, $A^t_{n\times n}$, of $A_{n\times n}$ is also computed. Next, element-wise comparison between A and A^t is performed. The comparison result is a matrix $I_{n\times n}$ such that $I_{i,j}$ is 1 if $A_{i,j} > A^t_{i,j}$ and 0 otherwise. The columns of $I_{n\times n}$ are then multiplied to obtain a column vector $V_{n\times 1}$ that contains 1 in the index of the maximal element of $\mathbf{X}$ and 0 in all other places. The vector $V_{n\times 1}$ is returned as the argmax result.

Assuming again that a SIMD-supporting FHE scheme with a configuration of s slots is used, the league method takes $O(\frac{n^2}{s})$ SIMD-comparisons. Hence, the drawback of this method is the huge number of comparisons needed for big values of n. Using the league method in FHE was discussed, for example, in [52], [36], [32] and [30].

2.1.3 Combined Tournament and League. Iliashenko and Zucca [36] suggested combining the tournament and the league methods as follows. They start with T iterations of the tournament method to reduce the number of remaining values from n to $\frac{n}{2^T}$. Then, they proceed with a league to compute the maximum of the remaining $\frac{n}{2^T}$ values. Given an FHE configuration with s slots, the combined tournament and league argmax takes $T + \frac{1}{s} \cdot \left(\frac{n}{2^T}\right)^2$ SIMD-comparisons. If $n = s$, this expression is minimized when $T = \frac{\log_2 n}{2}$, resulting with $\frac{\log_2 n}{2} + 1$ SIMD-comparisons. If $n > s$, then we can apply tournament rounds until reducing the candidate values into s candidates and then continue as in the case $n = s$, thus taking $\lceil \frac{n}{s} \rceil - 1 + \frac{\log_2 s}{2} = O(\frac{n}{s} + \log_2 s)$ SIMD-comparisons in total.

2.1.4 Numerical Method. Cheon et al. [22] suggested approximating the i-th bit of the argmax indicator vector corresponding to $argmax(x_0, x_1, \ldots x_{n-1})$ as $\frac{x_i^k}{\sum_{j=0}^{n-1} x_j^k}$. Since this formula converges to the argmax as $k \to \infty$, using a large enough k gives an approximation of argmax. This formula is difficult to compute directly using FHE because it involves values that are too small, assuming all inputs are in the range $(0, 1)$. In FHE schemes that encrypt real numbers and use approximate computations (such as CKKS), the inherent FHE noise may ruin the computation results if the encrypted values are too small. To tackle this problem, [22] presented an iterative process that computes $\frac{x_i^k}{\sum_{j=0}^{n-1} x_j^k}$ in a gradual manner. Even when using the gradual approach of [22], due to FHE noise,

this numerical argmax algorithm fails to find the correct argmax result if the input values are too close to each other.

More precisely, the inverse operation applied in the numerical method is difficult to compute over FHE. While inverse polynomial approximations, such as the Goldschmidt approximation [33], may be used, these approximations become less precise when inverting too small values. The accuracy of the Goldschmidt approximation may be increased to a certain level by increasing the number of iterations in the Goldschmidt algorithm, but this comes at the expense of making the inverse operation too slow and requiring additional costly bootstrap operations. Moreover, our experiments showed that when using 15 Goldschmidt iterations in the inverse polynomial approximation, the numeric argmax algorithm of [22] produces a wrong output when the input consists of 256 or more values where the minimal distance between two consecutive values in the input is 2^{-15}. In contrast, in Section 6, we show that our argmax algorithm successfully handles 2^{15} values where the minimal distance between two consecutive values is 2^{-15}.

2.1.5 Approximate Argmax. Lee et al. [42] proposed an approximate argmax algorithm. Given an integer parameter $k > 0$, the input of this algorithm is a vector $\mathbf{X} = \{x_0, x_1, \ldots, x_{n-1}\}$, and its output is an indicator vector I of length n, whose i-th value is 1 if and only if x_i agrees with the maximum of $\mathbf{X}$ on its first k bits. The algorithm uses $\lceil \frac{n}{s} \rceil \cdot k$ SIMD-comparisons. The smaller k is, the faster the k-approximate argmax algorithm becomes. If we assume the number of bits of the input values has a known limit, then using a large enough k converts this approximate argmax algorithm to an exact argmax algorithm. If we assume that the n input values are different values in the range $(0, 1)$, then at least $\log_2 n$ bits of accuracy are needed to find the exact argmax, meaning that k must be equal to $\log_2 n$ and therefore at least $\lceil \frac{n}{s} \rceil \log_2 n$ SIMD-comparisons are needed.

2.2 Prior Work on XGBoost Training over FHE

The problem of privacy preserving training and prediction over tree-based models have been widely investigated. Below is an overview of the existing prior art in this topic and an explanation of the advantages our work has compared to these prior works.

2.2.1 Partially Secure Training and Prediction. Some works offered solutions that compromise privacy by using techniques that expose some of the information on the data. These methods include, for example, order preserving encryption [49], searchable symmetric encryption [45] and applying perturbation on the data to limit the amount of information that can be deduced on specific samples [3], [48]. Our solution uses FHE to enable the data-owner to train an XGBoost model on his data using the cloud, where the data remains encrypted during the whole training process. FHE prevents the (semi-honest) server to deduce any information on the encrypted data, thus offering complete privacy.

2.2.2 Interactive Training using Multi-Party Computation. Some works (e.g. [11], [14], [17], [25], [39], [40], [53], [54], [58], [26], [27], [28], [59], [47], [62], [37], [46], [13]) have tackled the problem of training and prediction of tree-based models using interactive secure multiparty computation and secret sharing protocols. These protocols have the disadvantage of requiring multiple communication rounds and requiring the user to stay active for a longer period. In practical deployment scenarios, the server has a machine with strong computation power and the client's device has weak computation power. Requiring the client to stay active for multiple communication rounds, apply local computations and pass heavy data over the network is expected to be too much of a burden on such typical clients. For example, in one of the most recent works that perform interactive decision tree training, Lin et al. [46] report performing 61,000 communication rounds involving 51 GB to train a single tree of depth 6 on the adult dataset. Training a typical model of 100 trees of depth 6 is thus not feasible with this approach. Lu et al. [37] report training a single tree of depth 6 using 500MB of communication (i.e. 50GB for a model of 100 trees by extrapolation), but their model 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. Zhu et al. [63] show that disclosing this information can be exploited to infer training data. Our XGBoost training algorithm performs all computations of the server side in ciphertext space using FHE, thus requiring no intermediate rounds of interaction.

2.2.3 Training with FHE. While many papers dealt with the problem of performing *prediction* with tree-based models over FHE, only few papers have researched *training* algorithms of such models over FHE. Louis et al. [9] presented an algorithm for training a model called Completely Random Forest (CRF). In this model, several binary trees with a fixed depth are constructed. In the training phase, random features and thresholds are assigned to the internal nodes of the trees. Then, the training samples are passed through the random trees until they reach their corresponding leaves. Finally, the output values of the leaves are determined depending on the number of positive and negative training samples that reached them. While this model is quite simple to train over FHE, its completely random nature may make it less accurate than more advanced models, such as XGBoost. [6] discussed an FHE-friendly algorithm for training and prediction with decision trees. However, their training algorithm relies on the client to compute the argmin of impurity scores of candidate splits. This client-aided approach has the disadvantages of putting extra-work on the client, increasing communication bandwidth and a security risk imposed due to possible attacks [8] against CKKS scheme when it is used in client-aided algorithms.

Lee and Clark [43] discussed a method for non-interactive training of decision trees over FHE, but they only demonstrate the feasibility of their solution on small usecases consisting of one tree of depth 3 on a binary dataset, where our method can support XGBoost models as large as 100 trees of depth 6 on datasets including real values. More recently, [51] presented a non-interactive random forest training algorithm over FHE and demonstrated its feasibility on 64 trees of depth 4. Our algorithm for XGBoost training over FHE differs from [43] and [51] in two ways. First, we improve the efficiency of the training algorithm by a factor of up to 360×, by using more efficient building blocks such as a faster argmax algorithm, novel packing techniques and the tile tensor data structure [4]. Second, our algorithm deals with XGBoost training while [43], [51] deal with decision tree and random forest training. Since the

XGBoost model is a widely used model which proved to be among the most useful models for various machine learning tasks ([55], [10], [56], [1], [44]), we expect that providing an efficient algorithm to perform XGBoost training over FHE will prove highly beneficial.

3 Preliminaries

3.1 Fully Homomorphic Encryption

FHE is a cryptographic primitive which supports computation over encrypted data. In this paper, we focus on FHE schemes that support SIMD-operations. These schemes encrypt multiple values in one ciphertext, and support operations that are applied on all values of the ciphertexts concurrently. Our experiments use the CKKS scheme [20], which supports approximate arithmetic over real numbers using polynomial operations. Non-polynomial functions, such as comparisons, can be approximated via polynomial expressions. The algorithms presented in this paper are compatible with any FHE scheme that supports SIMD operations. A formal definition of FHE is provided in Appendix A.

3.2 XGBoost in Cleartext

XGBoost is a widely used machine learning model which was introduced in [19]. In this section, we briefly summarize some of the ideas presented in [19].

3.2.1 Model Definition. Given a dataset of n samples and m features, $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$, where $|\mathcal{D}| = n$, $\mathbf{x}_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$, an XGBoost model consists of K trees. Tree k is associated with T_k leaves, a function q that maps samples to leaves and a vector of weights $\mathbf{w} \in \mathbb{R}^{T_k}$. The prediction of the k-th tree is defined by a function $p_k : \mathbb{R}^m \to \mathbb{R}$ s.t. $p_k(\mathbf{x}) = \mathbf{w}_{q(\mathbf{x})}$. That is, given a sample $\mathbf{x}$, p_k returns the weight of the leaf which $\mathbf{x}$ reaches. The XGBoost prediction on a sample $\mathbf{x} \in \mathbb{R}^m$, which we denote by $\phi(\mathbf{x})$, is the sum of the predictions of the trees: $\phi(\mathbf{x}) = \sum_{k=0}^{K-1} p_k(\mathbf{x})$

A tree that is represented by the function $p(\mathbf{x}) = \mathbf{w}_{q(\mathbf{x})}$ consists of leaves and decision nodes. Each decision node has a split condition of the shape $f_j < t$, for an index $0 \le j < m$, a feature f_j and a threshold $t \in \mathbb{R}$. Let u be a node with a split condition $f_j < t$ and let $\mathbf{x}_i$ be a sample that reaches the node u. The sample $\mathbf{x}_i$ goes to the left child of u if $\mathbf{x}_{i,j} < t$ and it goes to the right child of u otherwise. This tree traversal is continued recursively until $\mathbf{x}_i$ reaches the leaf $q(\mathbf{x}_i)$. The output value, $\mathbf{w}_{q(\mathbf{x}_i)}$, is returned as the prediction result of the tree.

3.2.2 Training objective function. Let $l(\hat{y}_i, y_i)$ be a loss function, where $\hat{y}_i$ is the prediction of the model on the input sample $\mathbf{x}_i$ and y_i is the original label associated with $\mathbf{x}_i$. The XGBoost training algorithm builds a model that minimizes the following objective:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(p_k), \quad (1)$$
$$\text{where } \Omega(p_k) = \gamma T_k + \frac{1}{2}\lambda ||\mathbf{w}||^2,$$

for fixed parameters γ and λ. The XGBoost training algorithm operates in a greedy iterative process in which the t-th tree is built during the t-th iteration in order to correct the results of the previous trees. Denote by $\hat{y}_i^{t-1}$ the prediction of the trees built until iteration $t-1$ (inclusive) on the input sample $\mathbf{x}_i$. In the t-th iteration, we grow a tree that minimizes the following objective:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(\hat{y}_i^{(t-1)} + p_t(\mathbf{x}_i), y_i) + \Omega(p_t). \quad (2)$$

The XGBoost algorithm uses a second order approximation of $\mathcal{L}^{(t)}$:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} l(\hat{y}_i^{(t-1)}, y_i) + g_i p_t(\mathbf{x}_i) + \frac{1}{2} h_i p_t^2(\mathbf{x}_i) + \Omega(p_t), \quad (3)$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(\hat{y}^{(t-1)}, y_i)$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ are the gradient and Hessian of the loss function, respectively.

3.2.3 Growing the Trees. Growing the trees is done recursively. We start from a root node, choose a split that maximizes a score called *gain* (see Definition 3.1) for the root and then split the root to two children using the chosen split. Then, we proceed by splitting the two newly created children recursively. Growing the tree is stopped when reaching a pre-specified maximum depth.

Definition 3.1. Gain Value. Let u be a node, let $f_i < t$ be a split condition, let I_L, I_R be the samples that are sent to the left and right children of u following the split $f_i < t$ and denote $I = I_L \cup I_R$. The gain value associated with the split $f_i < t$ is defined as

$$\frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right]. \quad (4)$$

3.2.4 Splitting Nodes. There are several training algorithms for the XGBoost model. In the *basic exact greedy* algorithm, there is a set of candidate splits that is fixed throughout the entire training process. This fixed set consists of all possible splits on all features. More precisely, let $\mathcal{U}_j = (u_{j,1}, u_{j,2}, \ldots, u_{j,k_j})$ be the sorted and unique-d set of the values associated with the j-th feature in the training dataset, $\mathcal{D}$. The candidate splits corresponding to feature j that will be used during the training are the mid-points between each two consecutive values of $\mathcal{U}_j$, i.e. $f_j < \frac{u_{j,1}+u_{j,2}}{2}$, $f_j < \frac{u_{j,2}+u_{j,3}}{2}$, …, $f_j < \frac{u_{j,k_j-1}+u_{j,k}}{2}$ (see Figure 5). Upon splitting a node, we choose the split condition that achieves the highest gain among the set of all candidate splits.

When reaching a pre-specified maximum depth, we stop the splitting process and declare the reached nodes as leaves. We then compute output values to be associated with these leaves. Let I_j be the set of samples that have reached the j-th leaf. The output value of the j-th leaf is given by the following equation:

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \Lambda}, \quad (5)$$

where Λ is a fixed parameter.

3.3 Tile Tensors

Many FHE schemes (e.g. *BGV* [16], *B/FV* [15], [29] and *CKKS* [20]) work in a *SIMD* fashion. Leveraging the *SIMD* capabilities of such FHE schemes often requires the use of complex packing method. The choice of the data packing method can dramatically affect run-time and memory costs.

HeLayers [4] framework proposes a data structure called *tile tensor* that simplifies the packing decision for the user. In this section, we briefly summarize some of the basic tile tensor notation defined in [4], the reader is referred to [4] for further details.

3.3.1 Tensors. We use the term *tensor* as synonymous with multi-dimensional array. A rank-k tensor has a shape $[n_1, n_2, \ldots, n_k]$, where $0 < n_i$ is the size of the i'th dimension. For example, a matrix M with 5 rows and 6 columns is a rank-2 tensor of shape $[5, 6]$.

3.3.2 Tile Tensors. A tile tensor is a data structure that stores a tensor packed into one or more ciphertexts. Given a tensor A of shape $[n_1, n_2, \ldots, n_k]$ we first break it into equal-size blocks of some shape $[t_1, t_2, \ldots, t_k]$. These blocks are called *tiles*. Each tile is flattened in row-major order, and stored in a separate ciphertext.

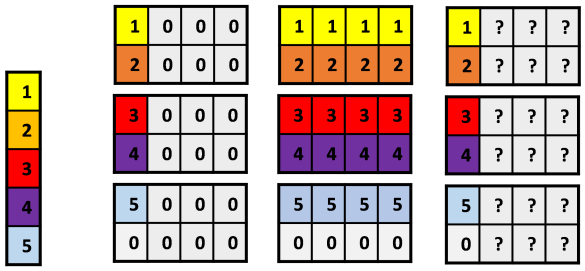
The notation $[\frac{n_1}{t_1}, \frac{n_2}{t_2}, \ldots, \frac{n_k}{t_k}]$, called the tile tensor shape, is used to denote a tile tensor structure that contains a tensor of shape $[n_1, \ldots, n_k]$ divided into tiles of shape $[t_1, \ldots, t_k]$, each stored flattened in a ciphertext with $\prod t_i$ slots. For example storing a matrix M of shape $[8, 8]$ in a tile tensor of shape $[\frac{8}{2}, \frac{8}{4}]$ means we first break M into sub-matrices of shape $[2, 4]$, and store each one flattened in a ciphertext with 8 slots. The row-major order convention means each such sub-matrix is flattened row by row, i.e., the ciphertext will contain the 4 elements of the first row contiguously, then the 4 elements of the second row.

If a tensor's size along some dimension is not divisible by the tile size along this dimension, we add zero padding in the unused slots.

Duplicates. Tile tensor shapes have further notation to indicate slots with duplicated values. An element $\frac{*}{t_i}$ in the tile tensor shape indicates values are duplicated along the i'th dimension.

Consider as an example a column vector V of shape $[5, 1]$ as shown in Figure 1a. We can pack it into a tile tensor T_V of shape $T_V[\frac{5}{2}, \frac{1}{4}]$, as in Figure 1b. This is similar to the previous example, where we break V into sub-matrices of size $[2, 4]$. Since it only has one column, we pad it with 3 zero columns. Similarly, since 5 is not divisible by 2 the last tile at the bottom has a zero padding row.

An alternative option is to pack it into a tile tensor T_V of shape $T_V[\frac{5}{2}, \frac{*}{4}]$ as in Figure 1c. The $\frac{*}{4}$ element indicates the tensor's size along this dimension is 1, but instead of padding it with zeroes, we duplicate it along the size of the tile, which in this example means duplicating each value 4 times.



(a) $V[5, 1]$ **(b)** $T_V[\frac{5}{2}, \frac{1}{4}]$ **(c)** $T'_V[\frac{5}{2}, \frac{*}{4}]$ **(d)** $T''_V[\frac{5}{2}, \frac{1?}{4}]$

Figure 1: Packing $V[5, 1]$ into tile tensors using different tile tensor shapes. Every rectangle represents a tile. Panel (c) demonstrates tiling with replication. Panel (d) demonstrates unknown values along the second dimension using question mark symbols.

Sum Operator. The operator $sum(T_A, i)$ refers to summing the values of T_A along the i-th dimension. It works by summing the different tiles along the i-th dimension to get one tile, and then summing inside this resulting tile. The latter summation is done using the rotate and sum algorithm [34], and it requires using $\log t_i$ rotations and $\log t_i$ additions over FHE, where t_i is the size of the i-th dimension in the tile shape of T_A.

For example, the operation $sum(T_V, 1)$ over the example in Figure 1b, will first add the 3 tiles together. Then it will add the two rows of the remaining single tile by adding it to itself with a rotation offset of 4. This will result with a single tile containing the sum of V in the first slot.

Binary Operators on Tile Tensors. Let T_A, T_B be two tile tensors of the shapes $s1, s2$, respectively.

Definition 3.2. $s1$ and $s2$ are said to be *compatible* shapes if they have the same number of dimensions, and for each dimension i, $s1$ and $s2$ either agree on the i-th dimension, or one is $\frac{*}{t_i}$ while the other is $\frac{n_i}{t_i}$.

If $s1$ and $s2$ are compatible, then binary operators between T_A and T_B are supported. In this case, binary operators between T_A and T_B translate to binary operators between their tiles. If $s1 = s2$, then the binary operator is applied between each pair of matching tiles of T_A and T_B. Otherwise, for each dimension i which is denoted by $\frac{*}{t_i}$ in one of the shapes and by $\frac{n_i}{t_i}$ in the other shape, the tiles in the replicated shape are further replicated n_i times along the i-th dimension. The replications result with two tile tensors of identical shapes. We then apply the operation on the two tile tensors of identical shape, tile-wise.

Matrix-vector Multiplication. Let $M[a, b]$ be a matrix and $V[b]$ be a vector. Following [4], to compute the product $M \cdot v$, we first pack the transpose, M^T, of M, and V in the two tile tensors $T_{M^T}[\frac{b}{t_1}, \frac{a}{t_2}]$ and $T_V[\frac{b}{t_1}, \frac{*}{t_2}]$. Then, we compute the matrix-vector multiplication using $sum(T_{M^T}[\frac{b}{t_1}, \frac{a}{t_2}] * T_V[\frac{b}{t_1}, \frac{*}{t_2}], 1)$, where t_1, t_2 are two integers whose product is equal to the number of slots in the used FHE configuration, and the operation $*$ stands for element-wise product between T_{M^T} and T_V.

4 Our Hierarchical Argmax Algorithm

4.1 Handling the Case of $n \leq s$

Recall that the league method computes the argmax of n values using $O(n^2)$ comparisons and hence it requires too many comparisons if n is large. To tackle this problem, we split the league method into multiple hierarchical league iterations. After each league iteration, the number of remaining candidate maximum values becomes smaller, allowing us to further utilize SIMD-computations and perform larger concurrent leagues on several subsets of the remaining values.

Our algorithm relies on the following distinction: when using d^2 slots, we can compute the max and argmax of d values using the league method (i.e. all-pairs comparison) with one SIMD-comparison. This all-pairs comparison is denoted by $League_{d \times d}$, and it requires each of the d values to be duplicated d times, thus utilizing the d^2 available slots. The more redundant slots we have, the larger the leagues we can support. We will gradually decrease

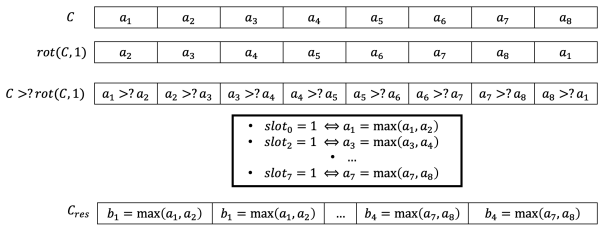


Figure 2: A tournament round that computes the maximum of each two consecutive slots using one SIMD comparison. In C_{res}, we duplicate the result among each two consecutive slots, using plaintext multiplications, rotations and additions.

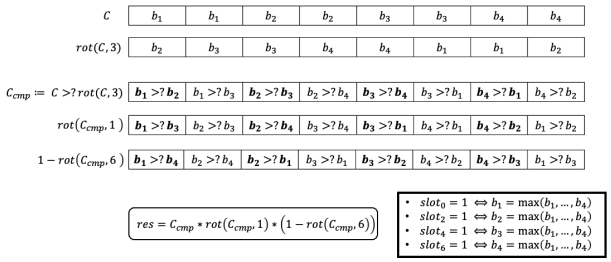


Figure 3: Implementation of $League^*_{4\times2}$ using one SIMD-comparison. The argmax result of the four values in the input ciphertext, C, end up in slots of indexes 0, 2, 4 and 6 of the result.

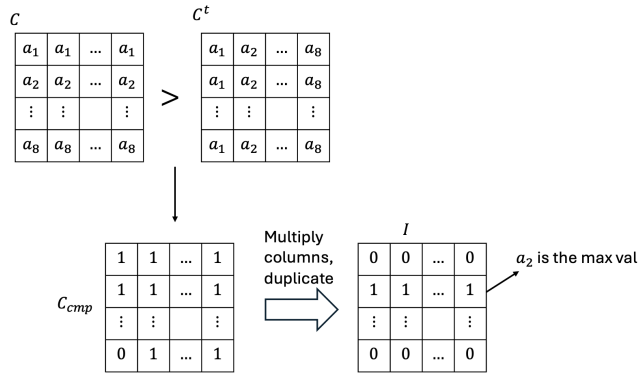


Figure 4: $League_{8\times8}$ example. Note that we add 0.5 to the values in the main diagonal of C so we will get 1 in these slots when computing $C > C^t$.

the number of candidate maximum values and increase the number of redundant slots, thus supporting larger league sizes each time.

Let s be the number of slots and assume for simplicity the number of input values, n, is equal to s. At the beginning of the algorithm, our input does not have any redundant slots and there is no place to duplicate any of the n values. Thus, our algorithm consists of two parts: the first part, which we call the *preparation part*, applies special initial comparisons to eliminate some of the maximum candidates and bring the input to a state where its values are duplicated d times each for some $d > 1$. The second part, which we call the *hierarchical part*, uses hierarchical league iterations, such that the league sizes increase each time, depending on the number of duplicates in the input of each league iteration. That is, assuming we have d duplicates for some $d > 1$, we apply $League_{d\times d}$.

In the beginning of the preparation part, since we do not have any redundant slots, we start with a tournament round that computes the maximum of every two consecutive slots (see Figure 2). After this tournament round, we are left with $\frac{n}{2}$ candidate maximum values, duplicated two times each. Theoretically we could now proceed to the second stage of the algorithm (the hierarchical part) and start the hierarchical part with a $League_{2\times2}$ operation. However, we designed an additional preparation step that will allow us to start the hierarchical part with $d = 8$ duplicates instead of $d = 2$, at the cost of merely one SIMD-comparison. To this end, we apply a special operation which we call $League^*_{4\times2}$. This operation is illustrated in Figure 3, and it computes the maximum of every 4 consecutive values that are duplicated two times each. This operation is special since it requires only 2 duplications but can compute the maximum of 4 values. The output of $League^*_{4\times2}$ contains $\frac{n}{8}$ candidate maximum values. See Figure 3 for implementation details of $League^*_{4\times2}$.

Now we are ready to proceed to the hierarchical part. Since we have $s = n$ slots and only $\frac{n}{8}$ candidate maximum values left, we can duplicate each candidate maximum value 8 times inside the input ciphertext, C (using the rotate-and-sum algorithm [34]), to get $\frac{n}{8}$ different values duplicated 8 times each. Utilizing the duplicates, we can apply simultaneous leagues (all-pairs comparisons) on each set of $8\times8 = 64$ consecutive slots of C, using a $League_{8\times8}$ operation (see Figure 4). The $League_{8\times8}$ operation divides the number of maximum candidate values by 8, which leaves us with $\frac{n}{64}$ candidate maximum values.

Now we are left with even more redundant slots, as we have $s = n$ slots with $\frac{n}{64}$ different candidate maximum values left. We duplicate each candidate value 64 times and apply leagues on sets of 64×64 blocks (i.e. $League_{64\times64}$), to be left with $\frac{n}{64^2}$ candidate maximum values. We continue applying leagues on blocks of $d \times d$ slots where d is squared each time, and we stop when we have only one candidate value left, which is equivalent to having $d \geq n$.

A more formal description of our argmax algorithm, showing the implementation details of how to compute the hierarchical leagues, is provided in Appendix B.

4.1.1 Number of SIMD Comparisons. Let us compute the number of SIMD comparisons required by our algorithm. Our algorithm starts with a tournament round and then a $League^*_{4\times2}$ operation, which together cost two SIMD comparisons. After that, we apply leagues on 8 values, then $8^2 = 64$ values and keep squaring the league size each time. We stop when having $(((8^2)^2)\cdots)^2 \geq n$. This inequality is satisfied after $\log_2 \log_2 n$ operations. Thus, a total of $O(\log \log n)$ SIMD-comparisons are required to compute the argmax.

4.2 Handling the Case of More Values than Slots

To support the case in where $n > s$, we start with tournament-like comparisons until reducing the number of candidate maximum

values to s. Then, we proceed with our hierarchical algorithm explained above to find the maximum among the remaining s values, which is also the maximum of the n input values. The number of SIMD comparisons needed for this process is $\lceil \frac{n-s}{s} \rceil + \log_2 \log_2 s$. In the first stage of the algorithm that consists of tournament-like comparisons, each SIMD-comparison compares a pair of ciphertexts containing s slots each, thus eliminating s candidates while sending the other s candidates to the next tournament-round. A total of $\lceil \frac{n-s}{s} \rceil$ SIMD comparisons will eliminate $n - s$ candidates and leave us with s maximum candidates. Then, our hierarchical argmax algorithm computes the maximum of the remaining s values using $\log_2 \log_2 s$ SIMD-comparisons. Overall, computing the argmax on n values takes $\lceil \frac{n-s}{s} \rceil + \log_2 \log_2 s$ SIMD-comparisons (c.f $\lceil \frac{n-s}{s} \rceil + \log_2 s$ SIMD-comparisons when using tournament method alone).

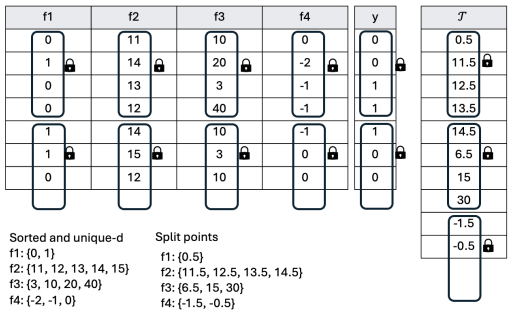


Figure 5: Encryption of training data and split points using ciphertexts with $s = 4$ slots.

5 XGBoost Training over FHE

5.1 Optimizing Matrix-vector Multiplication

Our algorithm for XGBoost training over FHE applies many matrix-vector multiplications in which the same encrypted matrix is multiplied by various encrypted vectors. In this section, we explain a method which speeds-up the matrix-vector multiplication explained in Section 3.3.2 at the cost of some extra processing on the multiplied matrix. Since we use the same matrix for many different multiplications, we get reduction in the overall runtime even though we spend some extra time processing the matrix.

As in Section 3.3.2, let $M[a, b]$ be a matrix whose transpose M^T is packed in the tile tensor $T_{M^T}[\frac{b}{t_1}, \frac{a}{t2}]$, and let $V[b]$ be a vector packed in the tile tensor $T_V[\frac{b}{t_1}, \frac{*}{t2}]$. The matrix-vector multiplication process explained in Section 3.3.2 can be divided to two steps. In the first step, we perform the element-wise multiplication $T_{mul} := T_{M^T}[\frac{b}{t_1}, \frac{a}{t_2}] * T_V[\frac{b}{t_1}, \frac{*}{t_2}]$. In the second step, we sum the result along the first dimension, using $sum(T_{mul}, 1)$. The first step requires $\lceil \frac{b}{t_1} \rceil * \lceil \frac{a}{t_2} \rceil$ multiplications, while the second step requires $\lceil \frac{b}{t_1} \rceil$ additions to sum the different tiles in the first dimension and then $\log(t_1)$ rotations and additions to sum the values inside the tiles, using the rotate and sum algorithm [34]. Since additions are cheap in the Fully Homomorphic Encryption (FHE) context, our main goal is minimizing the time of multiplications and rotations.

Assuming we use ciphertexts with s slots, the tile tensor data structure requires that $t_1 * t_2 = s$. The number of required rotations in the matrix-vector multiplication is $\log(t_1) = \log(\frac{s}{t_2})$. Therefore, using a larger t_2 and a smaller t_1 has the benefit of reducing the number of required rotations. We will show now that a larger t_2 also has a positive impact on the multiplication time. Generally, multiplication operations over *FHE* in the CKKS scheme should be followed by costly *relinearize* and *rescale* operations, but in the special case in which we have multiple products summed together, we can save relinearize and rescale operations by applying relinearize and rescale on the summed result once, instead of applying relinearize and rescale on each product. That is, given two sets of ciphertexts $\{a_i\}$ and $\{b_i\}$, we can compute the sum of their products as $rescale(relinearize(\sum a_i * b_i))$ instead of $\sum rescale(relinearize(a_i * b_i))$. Using this optimization in the above matrix-vector multiplication method, we need $\lceil \frac{a}{t_2} \rceil$ *relinearize* and *rescale* operations to compute the matrix-vector multiplication. In other words, a larger t_2 also has the benefit of reducing the number of required relinearize and rescale operations.

Our algorithm for XGBoost training over FHE applies many matrix-vector multiplications and thus it can benefit from the idea of using a large t_2 to reduce the runtime of the multiplication. However, in most other operations that happen during our algorithm, the encrypted data is required to be packed using one-dimensional tile shapes having $t_2 = 1$. Therefore, our algorithm includes a novel step of replacing the tile-tensor shape in the middle of the computation, only to reshape back to the original shape after the matrix-vector multiplication ends. Since reshaping is done once per many matrix-vector multiplications, we get a reduction in the total runtime even though we spend extra processing time on the reshape process. The implementation details of the suggested reshaped step are explained in Appendix C.

5.2 Input Packing and Encryption

As in Section 3.2, we perform the training over a dataset of n samples and m features, $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}(|\mathcal{D}| = n, \mathbf{x}_i \in \mathbb{R}^m, y_i \in \mathbb{R})$. The data-owner starts by encrypting $\mathcal{D}$ column-wise (see Figure 5), using the tile tensor data-structure explained in Section 3.3. Assuming we use ciphertexts with s slots, the features are encrypted using a tile tensor of shape $[\frac{n}{s}, m]$, and the labels, $\mathbf{y}$, are encrypted using a tile tensor of the shape $[\frac{n}{s}, 1]$. The data-owner also computes (in the clear) a fixed set of candidate split points to be used during the training, by following the process explained in Section 3.2.4. For $j < m$, Let k_j be the number of split points associated with the j-th feature. In our implementation, we assume that $k_0, k_1, \ldots, k_{m-1}$ are public parameters so they are exposed to the server. Let $f_j < t_{j,0}, f_j < t_{j,1}, \ldots, f_j < t_{j,k_j-1}$ be the split-points associated with the j-th feature, let

$\mathcal{T} = \{t_{0,0}, \ldots, t_{0,k_0-1}, t_{1,0}, \ldots, t_{1,k_1-1}, \ldots, t_{m-1,0}, \ldots, t_{m-1,k_{m-1}-1}\}$

be the set of all thresholds of all candidate split-points and denote $P = \|\mathcal{T}\|$. The data-owner encrypts $\mathcal{T}$ with a tile tensor of the shape $[\frac{P}{s}, 1]$. Figure 5 shows an example of input data packing and encryption. The data-owner sends the encrypted training data and the encryption of $\mathcal{T}$ to the server, which in turn starts the training process over FHE.

Algorithm 1: *train_XGBoost*

Input: n (num. samples), m (num. features), P (num. split pts.), an integer t_2, $Enc(\mathcal{D})$ (encrypted database), $Enc(\mathcal{T})$ (encrypted split pts.), an integer t_2 and XGBoost-specific hyperparams (e.g. *max_depth*, *num_trees*)

1 $T_C[\frac{n}{s}, P] \leftarrow compute\_comparisons()$
2 $T_C.reshape([\frac{n}{s/t_2}, \frac{P}{t_2}])$
3 **for** $i \leftarrow 0$ **to** *num_trees* **do**
4 $\quad T_G[\frac{n}{s}, 1], T_H[\frac{n}{s}, 1] \leftarrow compute\_grad\_hess()$
5 $\quad build\_next\_tree(Enc(\mathcal{D}), T_C, T_G, T_H)$// Sec. 5.3.1-5.3.2
6 **end**

Algorithm 2: *split_node*

Input: $Enc(\mathcal{D})$ (encrypted database), $Enc(\mathcal{T})$ (encrypted split pts.), T_I (input indicator), T_G (gradients), T_H (Hessians), T_C (precomputed comparisons).
Output: T_{I_l}, T_{I_r} (input indicators for left and right children).

1 $T_{Gain} = compute\_gain\_values(T_I, T_G, T_H, T_C)$// Alg. 4
2 $T_{ind} = argmax(T_{Gain})$// Section 4
3 $T_f, T_t = extract\_split(Enc(\mathcal{D}), Enc(\mathcal{T}), T_{ind})$// Get chosen split point from argmax indicator
4 Store T_f, T_t as the split for the current node.
5 $T_{I_l}, T_{I_r} := get\_child\_indicators(\mathcal{D}, T_{ind}, T_C)$// Use precomputed comparisons and chosen split to get input indicators of children.
6 **return** T_{I_l}, T_{I_r}

5.3 Training in Server Side

Algorithm 1 presents the high-level process of the XGBoost model building. In Line 1, the server starts the training process by computing the compare conditions associated with all features. Each comparison $f_j < t$ results with an encrypted vector of length n, which contains 1 in the indexes of the samples that satisfy the condition $f_j < t$ and 0 in all other places. These comparisons are stored and will be re-used upon splitting every node during the tree growing process. Since the encrypted input data is represented by tile-tensors with the tile tensor shape of $[\frac{n}{s}, P]$, the compare results will be initially held with the tile tensor shape $[\frac{n}{s}, P]$ as well. However, in Line 2, we apply a reshape operation on the compare results to make them be represented with tile-tensors of the shape $[\frac{n}{s/t_2}, \frac{P}{t_2}]$ for an integer $t_2 > 1$ (in our experiments we used $t_2 \in \{4, 8\}$). Recall from Section 5.1 that this reshape step can speed-up matrix-vector multiplication operations, which will prove useful when growing the XGBoost trees.

Once the pre-computation of the comparisons is done, the server starts an iterative process of growing the XGBoost trees (Lines 3 – 5). The next sections deep-dive into this tree-growing functionality.

5.3.1 Tree Growing over FHE. As explained above, the XGBoost training algorithm relies on starting from a root node and splitting recursively until a pre-specified max depth is reached. Algorithm 2 explains this node-splitting process. The algorithm starts by computing the gain values corresponding to all candidate splits, as will be explained in Section 5.3.2 and then applies our hierarchical argmax algorithm (Section 4) on the computed gain values. The resulting one-hot argmax indicator is used to compute two other encrypted indicators, T_{I_l} and T_{I_r}. T_{I_l} contains 1s in the indexes of the samples that will traverse to the left child while T_{I_r} contains 1s in the indexes of the samples that will traverse to the right child.

The recursive node-splitting ends when reaching a pre-specified maximum depth. At this stage, we declare the reached nodes as leaves and compute their output values following Equation 5. The division in Equation 5 is approximated using the Goldschmidt method [33].

5.3.2 Gain Computation. In this section, we explain how the gain values associated with the split candidates are efficiently computed. This gain computation is repeated once for each node of the XGBoost and thus it is executed up to thousands of times, meaning that improving the efficiency of the gain compuatation step is crucial. The explanation below shows how by utilizing the matrix-vector multiplication optimization explained in Section 5.1, we can reduce the runtime of the gain computation step.

Let u be a node and let P be the number of candidate splits. For $j < P$, denote by I_{L_j}, I_{R_j} the sets of samples that are sent to the left and right children of u following the j-th split and denote $I = I_{L_j} \cup I_{R_j}$. Note that I is the set of all samples that have reached u and thus it does not depend on a specific split (i.e. it is the same set for each j). Recall from Equation 4 that the gain value associated with the j-th split is defined as:

$$\frac{1}{2}\left[\frac{(\sum_{i\in I_{L_j}} g_i)^2}{\sum_{i\in I_{L_j}} h_i + \lambda} + \frac{(\sum_{i\in I_{R_j}} g_i)^2}{\sum_{i\in I_{R_j}} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right].$$

Since the term $\frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}$ is not dependent on the split index j, we can omit it from the gain expression without changing the order of the gain values associated with the different splits. Hence, the expression we aim to compute in this section is

$$\frac{(\sum_{i\in I_{L_j}} g_i)^2}{\sum_{i\in I_{L_j}} h_i + \lambda} + \frac{(\sum_{i\in I_{R_j}} g_i)^2}{\sum_{i\in I_{R_j}} h_i + \lambda}. \tag{6}$$

It holds that

$$\sum_{i\in I_{R_j}} g_i = \sum_{i\in I} g_i - \sum_{i\in I_{L_j}} g_i, \tag{7}$$

and

$$\sum_{i\in I_{R_j}} h_i = \sum_{i\in I} h_i - \sum_{i\in I_{L_j}} h_i. \tag{8}$$

Thus, we can compute the gain expression from the values $\sum_{i\in I} g_i$, $\sum_{i\in I_{L_j}} g_i$, $\sum_{i\in I} h_i$ and $\sum_{i\in I_{L_j}} h_i$.

Let $S_{g,l}[1, P]$ and $S_{h,l}[1, P]$ be tensors whose j-th value represents the sum of gradients and sum of Hessians of the samples that go to the left child of u following the j-th split. That is, $S_{g,l}(0, j) = \sum_{i\in I_{L_j}} g_i$ and $S_{h,l}(0, j) = \sum_{i\in I_{L_j}} h_i$. Similarly, let $S_{g,p}[1, 1]$ and $S_{h,p}[1, 1]$ be the sum of gradients and sum of Hessians of the samples that have reached u. Because $S_{g,p}$ and $S_{h,p}$

are not dependent on a specific split, they consists of merely one value. That is, $S_{g,p}(0,0) = \sum_{i\in I} g_i$ and $S_{h,p}(0,0) = \sum_{i\in I} h_i$. We next explain how to compute the gradient tensors $S_{g,p}$ and $S_{g,l}$, while the hessian tensors $S_{h,p}$ and $S_{h,l}$ can be similarly computed.

Let $I[n,1]$ be a tensor containing 1 in the indexes of the samples that have reached the node u and 0 in all other places and let $C[n,P]$ be the tensor of the pre-computed comparison results. That is, $C(i,j)$ contains 1 if the i-th sample should go left following the j-th split condition and it contains 0 otherwise. Recall that $G[n,1]$ is the tensor of the gradients of the training samples and let $G_I[n,1] = G[n,1] * I[n,1]$ be the element-wise multiplication of I and G. The following equations hold:

$$S_{g,p}[1,1] = sum(G_I[n,1],1), \tag{9}$$

$$S_{g,l}[1,P] = sum(C[n,P] * G_I[n,1],1), \tag{10}$$

where in Equation 10, the tensor $G_I[n,1]$ is broadcast by duplicating it P times along the second-dimension, and then an element-wise multiplication is performed between $C[n,p]$ and the broadcast tensor.

Algorithm 3 utilizes the above equations to compute the sum of gradients corresponding to the left child of the node u for each candidate split. The algorithm receives a tile tensor $T_I[\frac{n}{s},1]$ which contains 1s in the indexes of the samples that have reached u and 0 in all other places. The algorithm also receives the tile tensors $T_G[\frac{n}{s},1]$, $T_C[\frac{n}{s/t_2},\frac{P}{t_2}]$ holding the gradients and the comparison results, respectively, that were computed in Algorithm 1. Recall from Section 5.1 that matrix-vector multiplication becomes faster when using a two-dimensional tile shape whose second dimension, t_2, is greater than 1. Hence, in Line 3 of Algorithm 3, we reshape the gradients vector to have a tile shape of $[\frac{s}{t_2},t_2]$, while the comparison results, T_C were already reshaped in Algorithm 1. The specific reshaping method is explained in Appendix C.

Algorithm 3: *compute_sum_grad*

Input: $T_I[\frac{n}{s},1]$ (input indicator), $T_G[\frac{n}{s},1]$ (gradients), $T_C[\frac{n}{s/t_2},\frac{P}{t_2}]$ (compare results).
Output: $T_{S_{g,l}}[\frac{P}{s},1]$: The sum of gradients in the left child for each possible split point.

1 $T_{G_I}[\frac{n}{s},1] = T_G[\frac{n}{s},1] * T_I[\frac{n}{s},1]$
2 $T_{S_{g,p}}[\frac{1}{s},1] = sum(T_{G_I},1)$// Parent sum grads.
3 $T_{G_I,2D}[\frac{n}{s/t_2},\frac{*}{t_2}] = T_{G_I}.reshape([\frac{n}{s/t_2},\frac{*}{t_2}])$
4 $T_{g,l}[\frac{n}{s/t_2},\frac{P}{t_2}] = T_{G_I,2D}[\frac{n}{s/t_2},\frac{*}{t_2}] * T_C[\frac{n}{s/t_2},\frac{P}{t_2}]$ // left child grads
5 $T_{S_{g,l}}[\frac{1}{s/t_2},\frac{P}{t_2}] = sum(T_{g,l},1)$ // left child sum grads.
6 **return** $T_{S_{g,l}}.reshape([\frac{P}{s},1])$

After computing the sums of gradients and sums of Hessians, we can proceed with computing the gain values. To this end, Algorithm 4 uses Algorithm 3 to compute the gain values associated with each split, as they appear in Equation 6.

After computing the sums of gradients and sums of Hessians in Lines 1 – 2, Lines 3 – 6 of Algorithm 4 compute the numerators and denominators of the expressions that construct the gain values, $\frac{(\sum_{i\in I_{L_j}} g_i)^2}{\sum_{i\in I_{L_j}} h_i+\lambda}$ and $\frac{(\sum_{i\in I_{R_j}} g_i)^2}{\sum_{i\in I_{R_j}} h_i+\lambda}$. Because division is costly over FHE, instead of applying two divisions and summing two fractions, Line 7 merges the two fractions into one fraction with a common denominator. This fraction is then computed by one approximate division using the Goldschmidt inverse approximation [33], in Line 8. The optimization of combining two fractions into one saves one division for each node of the XGBoost model, and because the inverse operation is costly over FHE, the total saved runtime is significant.

Algorithm 4: *compute_gain_values*

Input: $T_I[n,1]$ (input mask), $T_G[\frac{n}{s},1]$ (gradients), $T_H[\frac{n}{s},1]$ (Hessians), $T_C[\frac{n}{s/t_2},\frac{P}{t_2}]$ (compare results).

1 $T_{S_{g,l}} = compute\_sum\_grad(T_I,T_G,T_C)$// Alg. 3
2 $T_{S_{h,l}} = compute\_sum\_hess(T_I,T_G,T_C)$
3 $T_{S_{g,r}} = T_{S_{g,p}} - T_{S_{g,l}}$ // right gradients
4 $T_{S_{h,r}} = T_{S_{h,p}} - T_{S_{h,l}}$ // right Hessians
5 $T_{N,l}, T_{N,r} = T^2_{S_{g,l}}, T^2_{S_{g,r}}$ // numerators of Eqn. 6
6 $T_{D,l}, T_{D,r} = T_{S_{h,l}} + \Lambda, T_{S_{h,r}} + \Lambda$ // denominators of Eqn. 6
7 $T_N = T_{N,l} * T_{D,r} + T_{N,r} * T_{D,l}$
8 $T_D = T_{D,l} * T_{D,r}$
9 **return** $T_N * approxInverse(T_D)$

5.4 Optimizations of the Basic Algorithm

Three categories of optimizations are used to further optimize the efficiency of our algorithm for XGBoost training over FHE as it was presented in Section 5.3: compressed packing, utilizing the complex part of the ciphertexts and reducing time spent on bootstrapping. We briefly describe these optimizations in this section, while more in-depth description is provided in Appendix D.

5.4.1 Compressed Packing. In the compressed packing optimization, we utilize the idea that nodes of the same level of an XGBoost tree are not dependent on each other and thus we can parallelize computations corresponding to such nodes by using the SIMD capability. Whenever we are processing tensors whose size is smaller than the number of slots, we compress tensors relating to nodes of the same level in the tree together in the same ciphertext. This optimization explains why in our reported runtime results (see Section 6.3.1), the runtime growth with respect to the depth of the trained tree is not exponential. Even though the number of nodes of the trees grows exponentially with the tree's depth, the compressed packing of multiple nodes together mitigates the exponential growth in run time.

5.4.2 Utilizing Complex Part. Ciphertexts in CKKS scheme [20] encrypt complex values. We can store two real values a,b together as $a+ib$ in one slot of a ciphertext. This allows us to perform twice the number of operations at the same cost as performing operations on only one value. While not all operations are supported by this complex-packed form, the gain computation step of Algorithm 3 can be sped up almost by a factor of 2 (see Appendix D for more details).

5.4.3 Reducing Bootstrap Time. In the CKKS scheme, ciphertexts are encrypted at a specific chain index which gets smaller following each multiplication. If the chain index reaches a certain threshold, a costly bootstrap operation must be applied to restore the ciphertext to have a higher chain index. Reducing the total time spent on bootstrapping is achieved by careful planning of the bootstrapping points and by utilizing complex-packing to bootstrap two values at once. See more details on recommended bootstrapping locations for our algorithm in Appendix D.

6 Experimental Results

6.1 Experimental Setup

All experiments were conducted on AMD EPYC 7763 64-Core Processor, 1 CPU, 900GB RAM, 1 GPU NVIDIA A100-SXM4-80GB. We used HELayers SDK [4] which is a software tool designed to efficiently execute analytical algorithms over encrypted data using FHE, leveraging different open and closed source FHE libraries. In the reported experiments, we used HELayers' wrapper of the HEaaN [20] implementation of the CKKS scheme. HEaaN library provides a GPU support, which was used in all of the experiments reported herein.

6.1.1 FHE Parameters. We used the *FGb* preset context implemented in the HEaaN library. This context is a bootstrappable context with 2^{15} slots, a multiplication depth of 12, a quantization bit size of 42 bits and an integer precision of 18 bits. The *FGb* context supports both "normal" bootstrapping and "extended" bootstrapping. The normal bootstrapping supports ciphertexts whose encrypted values fall in the range $(-1, 1)$, while the extended bootstrapping supports ciphertexts whose encrypted values fall in the range $(-2^{20}, 2^{20})$. The normal bootstrapping raises the ciphertexts from level 3 to level 12 and consumes 15 levels in total: it reduces the level from 3 to 0, then raises the level from 0 to 25 and then reduces it from 25 to 12. The extended bootstrapping, on the other hand raises the ciphertexts from level 4 to level 12 and consumes 16 levels in total. Table 2 shows the runtime of several operations when using the HEaaN FGb context and the machine described above.

6.2 Argmax Experiments

We compared the performance of several argmax algorithms on randomly generated inputs consisting of different values. The inputs were all in the range $(0, 1)$ and the distance between each two values in the input was 2^{-15} or more.

All implemented argmax algorithms relied on the comparison approximation of [21]. This version of comparison was used because it easily supports tuning the approximation accuracy to any desired level. We used a compare polynomial approximation which supports inputs whose distance from each other is 2^{-15} or more, and its output error is no more than 10^{-5}. To this end, $compare(a, b)$ is implemented as $sgn(a-b)$, where $sgn(x)$ is implemented by composing the polynomial $g(x) = -\frac{12860}{2^{10}}x^7 + \frac{25614}{2^{10}}x^5 - \frac{16577}{2^{10}}x^3 + \frac{4589}{2^{10}}x$ seven times and then the polynomial $f(x) = -\frac{5}{16}x^7 + \frac{21}{16}x^5 - \frac{35}{16}x^3 + \frac{35}{16}x$ two times - i.e. $sgn(x) = f(f(g(g(g(g(g(g(x))))))))$. Both polynomials are taken from [21]. For the sake of a fair comparison, we configured all the implemented argmax algorithms to result in similar approximation errors. To this end, we added an extra noise-cleaning step to the outputs of some of the algorithms (including ours). The reported runtimes include the runtime of this noise-cleaning step.

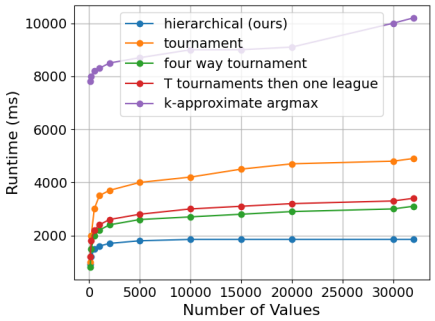
Figure 6 shows the experimental results of running several argmax algorithms on an encrypted input of n different values, $\{x_0, x_1, \ldots, x_{n-1}\}$, where n is smaller or equal than the number of slots, 2^{15}. The experimental results show that our argmax algorithm outperforms all other compared algorithms. Specifically, Figure 6 shows that when computing argmax on $n = 2^{15}$ values, our argmax algorithm is 1.6× times faster than the state of the art. It can also be seen from from the graph that the runtime of our algorithm increases at slower rate than other algorithms when n increases. This is due to the fact that other algorithms require $O(\log n)$ SIMD-comparisons, while our algorithm requires only $O(\log \log n)$ SIMD-comparisons (as long as n is not greater than the number of slots of the used FHE configuration). The graph in Figure 6b shows the maximal absolute difference between any two entries of the computed argmax one-hot indicator vector, I, and the expected argmax one-hot indicator vector.

The argmax algorithms we implemented and compared may be summarized as follows. The tournament method ([24], [18], [61], [60]) applies a tournament-like process to compute the max and argmax in $\log_2 n$ comparison rounds. The four-way tournament method [51] uses a tournament-like process in which sets of four elements are compared in each round, resulting with the final max and argmax result after $\log_4 n$ comparison rounds. The implementation of [51] requires a third of the ciphertext slots to be empty at the beginning of the algorithm, which is why the largest number of values the algorithm can be applied on is smaller than 2^{15}.

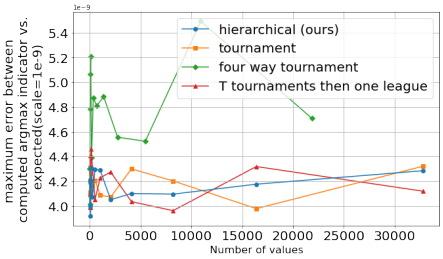
The method described in [36] relies on applying T tournament rounds and then one final league round. In our specific implementation of the argmax of [36], we chose T to be the smallest integer such that $\frac{n}{2^T} < \sqrt{2^{15}}$. This is in order to make sure that the $(\frac{n}{2^T})^2$ comparisons needed for the last league round can be carried out with one SIMD-comparison that uses 2^{15} slots.

The k-approximate argmax of [42] relies on k SIMD-comparisons to find all values that agree with the argmax on the first k most significant bits. In our implementation, we used $k = 15$ to ensure the argmax result is accurate when the input values have distance of 2^{-15} or more from each other. It should be noted that the algorithm of [42] becomes faster when it operates on input values whose distance from each other is greater, thus allowing to reduce k without harming the correctness of the argmax result. However, our test has 2^{15} unique values scaled to the range $(0, 1)$, so there must be two elements whose distance from each other is 2^{-15} or less, requiring us to use $k = 15$.

In addition to the reported methods, we implemented the numeric argmax method of [22], which approximates the i-th bit of the argmax indicator as $\frac{x_i^I}{\sum_j x_j^I}$. However, this method produced wrong results on values of n larger than 128, because it involves computations with very small values, which are difficult to handle over FHE (see Section 2.1.4 for more details). Finally, note that the k-approximate argmax method is not shown in Figure 6b of the argmax indicator error, because it resulted with errors that are an



(a) Runtime comparison



(b) Error comparison of argmax result. This error represents the maximal absolute difference between any two entries of the resulting argmax one-hot indicator vector and the expected argmax one-hot indicator vector.

Figure 6: Comparing performance of argmax algorithms. Ours vs. Tournament ([24], [18], [61], [60]) vs. 4-way tournament (([51]) vs. T-tournaments then one league ([36]) vs. k-approximate argmax ([42]).

Operation	Runtime (ms)
add	0.036
mult (level 12)	1.174
mult (level 4)	0.600
rotate	0.655
bootstrap (normal)	74.000
bootstrap (extended)	173.820
isGreater	233.860

Table 2: Runtime for basic CKKS operations

order of magnitude larger than the other methods and hence its error graph is far above the graphs of the other methods.

6.3 XGBoost Experiments

We evaluated our XGBoost training algorithm over FHE using the machine and FHE parameters described in the beginning of Section 6. Section 6.3.1 reports the runtime and accuracy of our algorithm when it is evaluated on 1 tree and Section 6.3.2 reports the runtime and accuracy of our algorithm when it is evaluated on 100 trees. Additional experiments, such as runtime breakdown for different steps of our algorithm and memory consumption, are included in Appendix E.

Table 3: Dataset descriptions including training and test set shapes.

Dataset	Training set shape	Test set shape
iris [31]	(120, 4)	(30, 4)
wine [2]	(144, 13)	(36, 13)
cancer [57]	(380, 30)	(269, 30)
sepsis [23]	(110341, 3)	(19051, 3)
adult [12]	(22793, 6)	(16281, 6)

Table 3 describes the datasets that were used for the evaluation of our algorithm. These datasets consist of three small-sized datasets (iris [31], wine [2] and cancer [57]) containing hundreds of samples and two medium-sized datasets containing tens of thousands (adult [12]) or a about a hundred thousand samples (sepsis [23]).

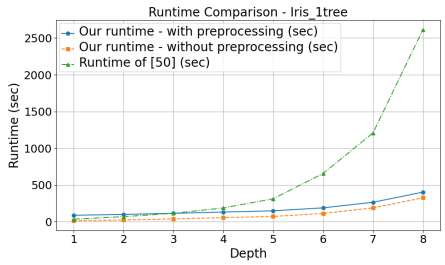
6.3.1 Evaluation of One Tree. Figure 7 compares the runtime of our XGBoost training algorithm with the runtime of the state of the art training algorithm [51]. We chose to compare with [51] because to the best of our knowledge it is the fastest prior solution that performs non-interactive XGBoost training over FHE. See Section 2.2.2 and Appendix E.3 for comparisons with works that apply interactive solutions using MPC and secret sharing.

The experiments of [51] used the same FHE parameters as we did (see Section 6.1.1). The machines used in our work and in [51] have similar performance. We deduced this by comparing Table 2 in our work and Table 4 in [51]. These two tables report the runtime of basic CKKS operations, such as multiplication, addition and bootstrapping, on the corresponding machines.

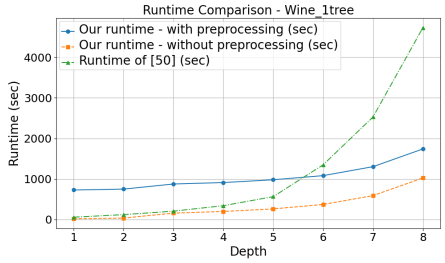
Our algorithm contains a preprocessing step which computes encrypted one-hot comparison results for each candidate split of the encrypted training data (Lines 1, 2 of Algorithm 1). The algorithm of [51], however, computes this step in the client side over cleartext data. For the sake of clear and fair comparison, we reported both the runtime of our algorithm with and without this preprocessing step. It should be noted that the time spent on this preprocessing step does not depend on the number of trees in the XGBoost model. Consequently, while it represents a significant portion of the runtime for a single-tree model, its relative cost becomes negligible in typical scenarios involving 100 trees or more.

Figure 7 show that our algorithm clearly outperforms the algorithm of [51]. When training a tree of depth 4 on the sepsis dataset, our algorithm is 360× faster than [51]. This figure also shows that while the runtime of [51] grows exponentially with the depth of the trees, our algorithm grows in a slower manner. We estimate that this happens because we have a better utilization of the SIMD slots, allowing us to parallelize computations in each level of an XGBoost tree. Due to this parallelization, the time we spend on training the l-th tree-level which contains 2^l nodes is not much larger than the time spent on the first level which contains only the root node (as long as l is small enough so the data of all 2^l nodes can fit in one ciphertext).

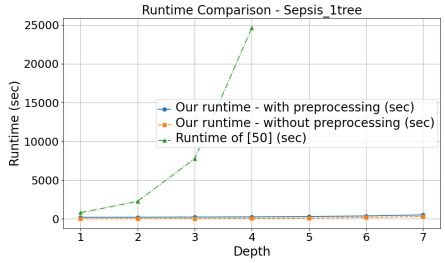
Table 4 shows that our runtime improvement does not compromise accuracy. The accuracy we achieve on all reported datasets



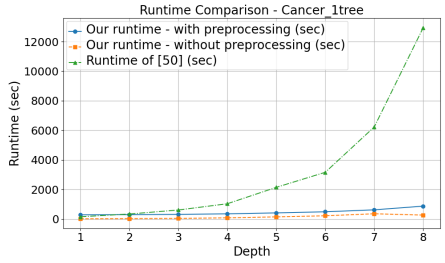
(a) Iris dataset



(b) Wine dataset



(c) Sepsis dataset. Note that [51] reported runtime only up to depth 4 for this dataset.



(d) Cancer dataset

Figure 7: Runtime comparisons across different datasets for a single tree. Each graph compares our runtime (with and without preprocessing) against the state of the art [51]

is comparable to the accuracy of the XGBoost python library [19] when it is run on cleartext data.

6.3.2 Evaluation of 100 *Trees.* To prove that our algorithm scales well when working with larger models, we report the runtime and accuracy metrics of our algorithm when it is ran on the medium-sized datasets adult and sepsis. Table 5, shows that our algorithm takes about 5 hours to train a model of 100 trees of depth 6 on these datasets. This runtime is practical in many settings, as model training can be performed offline—the client can submit a request to the cloud and receive the trained model within a single night. We also show in Table 5 that the accuracy achieved by our algorithm is very similar to the accuracy achieved by the cleartext XGBoost python library.

Table 4: Accuracy comparison between our FHE-based algorithm and the cleartext XGBoost python library [19] (1 tree, depth 6).

Dataset	our accuracy	cleartext accuracy
iris	$f_1 = 1$	$f_1 = 1$
wine	$f_1 = 1$	$f_1 = 1$
cancer	$f_1 = 0.96$	$f_1 = 0.92$
sepsis	$f_1 = 0.9$	$f_1 = 0.9$

Table 5: Runtime results for running our FHE XGBoost training algorithm with 100 trees of depth 6, as well as accuracy comparison between our FHE training and cleartext training using XGBoost python library.

Dataset	Runtime (FHE)	Accuracy (FHE / Cleartext)
sepsis	5.5	*FHE:* $f_1 = 0.89$ *Cleartext:* $f_1 = 0.89$
adult	5.1	*FHE:* $f_1 = 0.78$ *Cleartext:* $f_1 = 0.78$

6.3.3 Memory Consumption and Communication. We write a summary of our memory consumption details here and the reader is referred to Appendix E.2 fore more details. Our experiments consume up to 80GB of GPU memory and up to 900GB of CPU memory. However, this RAM consumption happens solely in the server side which is assumed to have high computation power. The communication size is about $100MB$ from the client to the server (to send the encrypted database) and less than $10MB$ from the server to the client (to send the resulting model). This does not include the setup stage in which the public FHE keys (encryption key, rotation keys and evaluation keys) should be sent from the client to the server, costing 4GB of communication. The communication of the setup stage happens offline and it is one time only, meaning that the same keys can be used for multiple FHE tasks, including training of several XGBoost models or other FHE computations.

6.3.4 Impact of our Argmax Algorithm on Overall Training Time. Among the datasets we used, the Wine dataset is the one where the argmax step required the highest percentage of the run time. We use our hierarchical argmax algorithm for this dataset, and by doing so we save about 1.7 hours (about 16%) from the total training time of a model of 100 trees of depth 6, compared to a training algorithm which would use the prior best argmax algorithm. When running on datasets with more candidate split points, the impact of our argmax algorithm will be larger. The reader is referred to Appendix E.4 for more details.

7 Acknowledgment

This work was supported in part by the Center for Cyber Law & Policy at the University of Haifa in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

References

[1] O. Adeoye and S. Oludare. 2024. A Comparative Study of LSTM and XGBoost for Stock Price Prediction. *Mountain Top University Journal of Applied Science and Technology (MUJAST)* 4, 2 (2024), 45–56. https://mujast.mtu.edu.ng/storage/issues/Year_2024_Vol_4/Number_2/1729800557_MUJAST_240801.pdf

[2] Stefan Aeberhard and M. Forina. 1992. Wine. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5PC7J.

[3] Rakesh Agrawal and Ramakrishnan Srikant. 2000. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 439–450.

[4] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. 2023. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *Privacy Enhancing Technology Symposium (PETs) 2023* (2023).

[5] Adi Akavia, Dan Feldman, and Hayim Shaul. 2018. Secure Search on Encrypted Data via Multi-Ring Sketch. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 985–1001.

[6] Adi Akavia, Max Leibovich, Yehezkel S. Resheff, Roey Ron, Moni Shahar, and Margarita Vald. 2022. Privacy-Preserving Decision Trees Training and Prediction. 25, 3, Article 24 (May 2022), 30 pages.

[7] Adi Akavia, Ramy Masalha, and Reut Meiri. 2025. Similarity-Based Retrieval over Homomorphic Encryption. (2025). Accepted for publication in the International Symposium on Cyber Security, Cryptology and Machine Learning (CSCML) 2025, to appear.

[8] Adi Akavia and Margarita Vald. 2021. On the Privacy of Protocols based on CPA-Secure Homomorphic Encryption. Cryptology ePrint Archive, Paper 2021/803. https://eprint.iacr.org/2021/803 https://eprint.iacr.org/2021/803.

[9] Louis J. M. Aslett, Pedro M. Esperança, and Chris C. Holmes. 2015. Encrypted statistical machine learning: new privacy preserving methods. arXiv:1508.06845 [stat.ML]

[10] PubMed Authors. 2024. Comparison of Artificial Neural Networks and XGBoost in Predicting Tomato Quality Parameters. *Postharvest Biology and Technology* (2024). https://doi.org/10.1016/j.postharvbio.2024.112603

[11] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Secure Evaluation of Private Linear Branching Programs with Medical Applications. In *Computer Security – ESORICS 2009*, Michael Backes and Peng Ning (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 424–439.

[12] Barry Becker and Ronny Kohavi. 1996. Adult. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20.

[13] Divyanshu Bhardwaj, Sandhya Saravanan, Nishanth Chandran, and Divya Gupta. 2024. Securely Training Decision Trees Efficiently. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4673–4687.

[14] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine Learning Classification over Encrypted Data. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society.

[15] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 868–886.

[16] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. *Electron. Colloquium Comput. Complex.* TR11 (2011).

[17] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. 2007. Privacy-preserving remote diagnostics. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 498–507.

[18] Olive Chakraborty and Martin Zuber. 2022. Efficient and Accurate Homomorphic Comparisons. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Los Angeles, CA, USA) *(WAHC'22)*. Association for Computing Machinery, New York, NY, USA, 35–46.

[19] 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* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794.

[20] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[21] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 2020. Efficient Homomorphic Comparison Methods with Optimal Complexity. In *Advances in Cryptology – ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 221–256.

[22] Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, and Keewoo Lee. 2019. Numerical Method for Comparison on Homomorphically Encrypted Numbers. In *Advances in Cryptology – ASIACRYPT 2019*, Steven D. Galbraith and Shiho Moriai (Eds.). Springer International Publishing, Cham, 415–445.

[23] Davide Chicco and Giuseppe Jurman. 2020. Sepsis Survival Minimal Clinical Records. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C53C8N.

[24] Jack L. H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. 2018. Doing Real Work with FHE: The Case of Logistic Regression. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Toronto, Canada) *(WAHC '18)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[25] Martine De Cock, Rafael Dowsley, Caleb Horst, Raj Katti, Anderson C. A. Nascimento, Wing-Sea Poon, and Stacey Truex. 2019. Efficient and Private Scoring of Decision Trees, Support Vector Machines and Logistic Regression Models Based on Pre-Computation. *IEEE Transactions on Dependable and Secure Computing* 16, 2 (2019), 217–230.

[26] Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. 2014. Practical Secure Decision Tree Learning in a Teletreatment Application. In *Financial Cryptography and Data Security*, Nicolas Christin and Reihaneh Safavi-Naini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–194.

[27] Wenliang Du and Zhijun Zhan. 2002. Building decision tree classifier on private data. In *Proceedings of the IEEE International Conference on Privacy, Security and Data Mining - Volume 14* (Maebashi City, Japan) *(CRPIT '14)*. Australian Computer Society, Inc., AUS, 1–8.

[28] F. Emekci, O.D. Sahin, D. Agrawal, and A. El Abbadi. 2007. Privacy preserving decision tree learning over multiple parties. *Data & Knowledge Engineering* 63, 2 (2007), 348–361.

[29] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144.

[30] Mazzone Federico, Everts Maarten, Hahn Florian, and Peter Andreas. 2025. Efficient Ranking, Order Statistics, and Sorting under CKKS. In *34th USENIX Security Symposium (USENIX Security '25)*. USENIX Association, Seattle, WA.

[31] R. A. Fisher. 1936. Iris. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C56C76.

[32] Evrard Garcelon, Vianney Perchet, and Matteo Pirotta. 2022. Encrypted Linear Contextual Bandit. In *AISTATS 2022 - 25th International Conference on Artificial Intelligence and Statistics*. Valence, Spain.

[33] Robert E Goldschmidt. 1964. Applications of division by convergence. https://api.semanticscholar.org/CorpusID:109445587

[34] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *Advances in Cryptology – CRYPTO 2014*, Juan A. Garay and Rosario Gennaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 554–571.

[35] W. Daniel Hillis and Guy L. Steele. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (dec 1986), 1170–1183.

[36] Ilia Iliashenko and Vincent Zucca. 2021. Faster homomorphic comparison operations for BGV and BFV. *Proceedings on Privacy Enhancing Technologies* 2021 (2021), 246 – 264.

[37] 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 *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6435–6451.

[38] Yehuda Lindell Jonathan Katz. 2020. *Introduction to Modern Cryptography*. CRC.

[39] Marc Joye and Fariborz Salehi. 2018. Private yet Efficient Decision Tree Evaluation. In *Data and Applications Security and Privacy XXXII*, Florian Kerschbaum and Stefano Paraboschi (Eds.). Springer International Publishing, Cham, 243–259.

[40] Ágnes Kiss, Masoud Naderpour, Jian Liu, N. Asokan, and T. Schneider. 2019. SoK: Modular and Efficient Private Decision Tree Evaluation. *Proceedings on Privacy Enhancing Technologies* 2019 (2019), 187 – 208.

[41] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. 2022. Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2022), 3711–3727. https://doi.org/10.1109/TDSC.2021.3105111

[42] Hyunjun Lee, Jina Choi, and Younho Lee. 2023. Approximating Max Function in Fully Homomorphic Encryption. *Electronics* 12, 7 (2023).

[43] Hunjae Lee and Corey Clark. 2023. Privacy Preserving Decision Tree Training and Prediction via Fully Homomorphic Encryption with No Decryption. *CoRR* abs/2310.13140 (2023). https://doi.org/10.48550/ARXIV.2310.13140 arXiv:2310.13140

[44] Yuxuan Li and Zihan Zhao. 2024. Sea Surface Temperature Prediction in the Great Barrier Reef Using Machine Learning. *arXiv preprint arXiv:2411.15202* (2024). https://arxiv.org/abs/2411.15202

[45] Jinwen Liang, Zheng Qin, Liang Xue, Xiaodong Lin, and Xuemin Shen. 2021. Efficient and Privacy-Preserving Decision Tree Classification for Health Monitoring Systems. *IEEE Internet of Things Journal* 8, 16 (2021), 12528–12539. https://doi.org/10.1109/JIOT.2021.3066307

[46] Guopeng Lin, Weili Han, Wenqiang Ruan, Ruisheng Zhou, Lushan Song, Bingshuai Li, and Yunfeng Shao. 2024. Ents: An Efficient Three-party Training Framework for Decision Trees by Communication Optimization. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4376–4390.

[47] 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* 15 (2020), 2914–2929.

[48] Li Liu, M. Kantarcioglu, and B. Thuraisingham. 2009. Privacy Preserving Decision Tree Mining from Perturbed Data. In *2009 42nd Hawaii International Conference on System Sciences*. 1–10. https://doi.org/10.1109/HICSS.2009.353

[49] Xianrui Meng and Joan Feigenbaum. 2020. Privacy-Preserving XGBoost Inference. arXiv:2011.04789 [cs.CR] https://arxiv.org/abs/2011.04789

[50] Marie Paindavoine and Bastien Vialla. 2016. Minimizing the Number of Bootstrappings in Fully Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2015*, Orr Dunkelman and Liam Keliher (Eds.). Springer International Publishing, Cham, 25–43.

[51] Hojune Shin, Jina Choi, Dain Lee, Kyoungok Kim, and Younho Lee. 2024. Fully Homomorphic Training and Inference on Binary Decision Tree and Random Forest. In *Computer Security – ESORICS 2024: 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16–20, 2024, Proceedings, Part III* (Bydgoszcz, Poland). Springer-Verlag, Berlin, Heidelberg, 217–237.

[52] Arnaud Sébert, Rafael Pinot, Martin Zuber, Cédric Gouy-Pailler, and Renaud Sirdey. 2021. SPEED: secure, PrivatE, and efficient deep learning. *Machine Learning* 110 (04 2021). https://doi.org/10.1007/s10994-021-05970-3

[53] Raymond K. H. Tai, Jack P. K. Ma, Yongjun Zhao, and Sherman S. M. Chow. 2017. Privacy-Preserving Decision Trees Evaluation via Linear Functions. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 494–512.

[54] Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. 2019. Private Evaluation of Decision Trees using Sublinear Cost. *Proceedings on Privacy Enhancing Technologies* 2019 (01 2019), 266–286. https://doi.org/10.2478/popets-2019-0015

[55] Güzin Türkmen and Arda Sezen. 2024. A Comparative Analysis of XGBoost and LightGBM Approaches for Human Activity Recognition: Speed and Accuracy Evaluation. *International Journal of Computational and Experimental Science and Engineering* 10 (06 2024).

[56] Lijun Wang, Xiaoxuan Ma, and Weiguo Gao. 2024. Simulation of Atmospheric Visibility Based on Meteorological Parameters Using Machine Learning Methods. *Frontiers in Environmental Science* 12 (2024), 1534113. https://doi.org/10.3389/fenvs.2024.1534113

[57] Wolberg William, Mangasarian Olvi, Street Nick, and Street W. 1993. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5DW2B.

[58] David Wu, Tony Feng, Michael Naehrig, and Kristin Lauter. 2016. Privately Evaluating Decision Trees and Random Forests. *Proceedings on Privacy Enhancing Technologies* 2016 (02 2016). https://doi.org/10.1515/popets-2016-0043

[59] Li Ye, Zoe Jiang, Xuan Wang, and Sm Yiu. 2017. Privacy-Preserving ID3 Data Mining over Encrypted Data in Outsourced Environments with Multiple Keys. 548–555.

[60] Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen-jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. 2025. Secure Transformer Inference Made Non-interactive. In *2025 Network and Distributed System Security (NDSS) Symposium*.

[61] Peng Zhang, Ao Duan, and Hengrui Lu. 2024. An Efficient Homomorphic Argmax Approximation for Privacy-Preserving Neural Networks. *Cryptography* 8, 2 (2024).

[62] Yifeng Zheng, Huayi Duan, Cong Wang, Ruochen Wang, and Surya Nepal. 2022. Securely and Efficiently Outsourcing Decision Tree Inference. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2022), 1841–1855. https://doi.org/10.1109/TDSC.2020.3040012

[63] Zutao Zhu and Wenliang Du. 2010. Understanding privacy risk of publishing decision trees. In *Proceedings of the 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy* (Rome, Italy) *(DBSec'10)*. Springer-Verlag, Berlin, Heidelberg, 33–48.

A Fully Homomorphic Encryption Definition

HE and FHE schemes may be defined as follows:

Definition A.1. A homomorphic (public-key) encryption scheme $\varepsilon = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ with message space $\mathcal{M}$ is a quadruple of Probabilistic Polynomial Time (PPT) algorithms as follows:

- Gen (key generation) takes as input the security parameter 1^λ, and outputs a pair (pk, sk) consisting of a public key pk and a secret key sk; denoted: $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$.
- Enc (encryption) takes as input a public key pk and a message $m \in \mathcal{M}$, and outputs a ciphertext c; denoted: $c \leftarrow \mathsf{Enc}_{pk}(m)$.
- Dec (decryption) takes as input a secret key sk and a ciphertext c, and outputs a decrypted message m'; denoted: $m' \leftarrow \mathsf{Dec}_{sk}(c)$.
- Eval (homomorphic evaluation) takes as input the public key pk, a circuit $C : \mathcal{M}^l \rightarrow M$ and ciphertexts $c_1, \ldots, c_l$, and outputs a ciphertext $\hat{c}$; denoted $\hat{c} \leftarrow \mathsf{Eval}_{pk}(C, c_1, \ldots, c_l)$.

Correctness. The scheme is correct if for every (pk, sk) in the range of $\mathsf{Gen}(1^\lambda)$ and every message $m \in \mathcal{M}$,

$$Pr[\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m)) = m] \geq 1 - \mathsf{neg}(\lambda),$$

where the probability is taken over the random coins of the encryption algorithm.

$\mathcal{C}$-homomorphism. A homomorphic encryption scheme is $\mathcal{C}$-homomorphic for a circuit family $\mathcal{C}$ if for all $C \in \mathcal{C}$ and for any set of inputs $x_1, \ldots, x_l$ to C, letting $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$ and $c_i \leftarrow \mathsf{Enc}(pk, x_i)$ it holds that:

$$Pr[\mathsf{Dec}_{sk}(\mathsf{Eval}_{pk}(C, c_1, \ldots, c_l)) \neq C(x_1, \ldots, x_l)] \leq \mathsf{neg}(\lambda),$$

where the probability is taken over all randomness in the experiment.

Compactness. A homomorphic encryption scheme is compact if there exists polynomial $p(\cdot)$ such that the decryption algorithm can be expressed as a circuit of size $p(\lambda)$.

Fully homomorphic. A homomorphic encryption scheme is fully homomorphic if it is both compact and $\mathcal{C}$-homomorphic for the class $\mathcal{C}$ of all efficiently computable circuits.

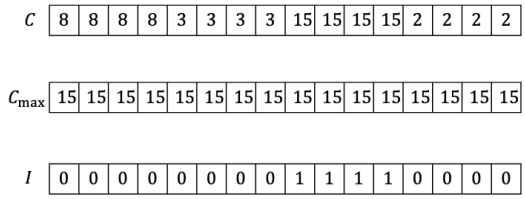
Security. A homomorphic encryption scheme is CPA-secure if no PPT adversary $\mathcal{A}$ can distinguish between the encryption of two equal length messages x_0, x_1 of his choice. See a formal definition of CPA-security in [38].

B Implementation Details of our Argmax Algorithm Over FHE

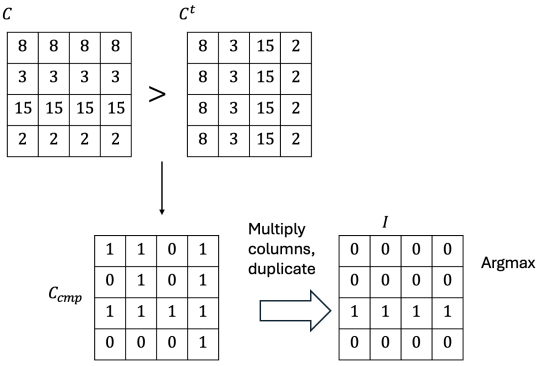
B.1 Main Algorithm

We start by introducing the $League_{m\times d}$ operation, which is defined as follows.

Definition B.1. Let s be the number of slots in the used FHE configuration and let $m \leq d$ be two integers such that $md \leq s$. $League_{m\times d}$ is an operation that receives a ciphertext C encrypting $\frac{s}{d}$ different values, that are duplicated d times each. That is, for each $0 \leq i < \frac{s}{d}$, slots $id, id+1, \ldots id+d-1$ of C have equal values. The operation applies a separate league on each block of md consecutive slots of C (these md consecutive slots contain m different values). As a result, two ciphertexts, C_{max} and I, are returned. C_{max} is a ciphertext containing the maximum of each block of md slots, duplicated along these md slots. The second returned ciphertext, I, is the argmax indicator ciphertext. I contains 1 in the slots whose corresponding value in C is equal to the chosen maximum and 0 in all other slots.



(a) An example of $League_{4\times4}$ operation. C is the input of $League_{4\times4}$. C_{max} contains the maximal value in the league, duplicated along the 4×4 league's slots. I is the argmax indicator ciphertext.



(b) Implementation of $League_{4\times4}$ using one SIMD-comparison. Note that we actually add $\varepsilon > 0$ to the elements of the main diagonal of C before comparison so the comparison result of $C + \varepsilon > C^t$ will contain 1s in the main diagonal.

Figure 8: $League_{4\times4}$ example.

See Figure 8a for an example of $League_{4\times4}$ operation on a ciphertext with 16 slots. Usually we will have $m = d$, but we also support the case $d > m$ in our definition to deal with a possible edge case which will become clear when we will present our argmax algorithm (for example, the last iteration of Example B.1 is such an edge case). Figure 8b shows how $League_{m\times d}$ may be implemented using one SIMD-comparison (in Figure 8b, $m = d = 4$). First, we consider each block of md slots of C as representing an $m \times d$ matrix (slots $0, \ldots, d-1$ represent the first row, slots $d, \ldots, 2d-1$ represent the second row, etc.). Then, we construct a second ciphertext, C^t, which is defined as follows. Each block of md consecutive slots of C^t again represents an $m \times d$ matrix. If $m = d$, then C^t is obtained from transposing every $m \times m$ matrix of C. That is, the $m \times m$ matrices composing C^t are simply the transpose of the $m \times m$ matrices that compose C. If $m < d$, then each $m \times d$ matrix of C^t is obtained from transposing the The leftmost $m \times m$ sub-matrix of the corresponding $m \times d$ matrix of C, and then putting zeros in the other $m \times (d-m)$ entries. Specific methods to construct C^t are discussed in Appendix B.2.

Once we constructed C^t, we use an approximate *isGreater* operation to apply one element-wise SIMD-comparison between C and C^t (we actually add $\varepsilon > 0$ to the values in the main diagonal of C so they will be considered as greater than their counterparts of C^t). Denote the comparison result by C_{cmp}. By multiplying the left-most m columns of the matrices represented by C_{cmp} and then duplicating the product result along the matrices rows, we obtain the desired argmax indicator, I (see Figure 8b). The max result, C_{max}, is obtained by multiplying I with C and duplicating the results along each block of consecutive md slots of C_{max}.

$League_{m\times d}(m \le d)$ relies on comparing at least $md \ge m^2$ pairs of slots to find the maximum of m values. These comparisons contain some redundancy as one could theoretically deduce the maximum from $\binom{m}{2}$ comparisons of slots of indexes i, j s.t. $i < j$, without the need for all of the md comparisons. However, properly aligning these $\binom{m}{2}$ pairs of slots in two different ciphertexts requires many rotations, which are costly to perform over FHE. Hence, and to utilize the matrix representation shown in Figure 8b, we perform $League_{m\times d}$ using md comparisons, even at the expense of having some redundant comparisons.

In the special cases $m \in \{2, 4\}$, there are easy methods to perform leagues of size m with less than m^2 comparisons, thus reducing redundancy and improving SIMD utilization. To this end, we define the operations $League^*_{2\times1}$ and $League^*_{4\times2}$.

Definition B.2. $League^*_{2\times1}$ is an operation that receives a ciphertext C with s slots encrypting s different values (i.e. C does not have any duplicated values). The operation computes the maximum of each two consecutive slots of C and returns two ciphertexts, C_{max} and I. C_{max} encrypts the maximum of each two consecutive slots, duplicated along these two slots. I is an indicator ciphertext that encrypts 1 in the indexes of the maximum slots and 0 in the indexes of the other slots.

$League^*_{2\times1}$ can be simply implemented by rotating C by 1, comparing C with its rotation, and then extracting C_{max} and I from the comparison result using few rotations and multiplications with plaintext indicator vectors. $League^*_{2\times1}$ is basically identical to a tournament round.

Definition B.3. $League^*_{4\times2}$ is an operation that receives a ciphertext C with s slots, duplicated two times each. That is, for each $i < \frac{s}{2}$, slots $2i, 2i+1$ of C encrypt the same value. $League^*_4$ computes the maximum of every block of 8 slots (that encrypts 4 different values) concurrently. The operation again returns two ciphertexts C_{max} and I that are defined as in previous operations.

Figure 9 shows how to compute I in $League^*_{4\times2}$ using one SIMD-comparison. The basic idea is that by rotating the input ciphertext, C, three slots to the left, and then comparing C with $rot(C, 3)$, we get all $\binom{4}{2} = 6$ pairs of comparisons needed to deduce the maximum of four input values of C. Denote $C_{cmp} := isGreater(C, rot(C, 3))$. Figure 9 shows that the product $C_{cmp} * rot(C_{cmp}, 1) * (1 - rot(C_{Cmp}, 6))$ results in the argmax of the four input values in the first, third, fifth and seventh slots of the product. These slots will then be duplicated to the second, fourth, sixth and eighth slots, respectively, to get the argmax indicator, I. Note that while in Figure 9 we compute the argmax on a block of eight slots that contains four different values, in practice we apply $League^*_{4\times2}$ on a ciphertext with more than 8 slots and we perform a separate league on each block of 8 slots. Thus, some extra ciphertext-plaintext multiplications and rotations may be needed to make sure that after rotations the values that fall off from the left edge of each eight slots block of the rotated ciphertext cycle back to the right edge of the same block. After computing I as described in Figure 9, computing C_{max} from I can be easily done by multiplying I with C, summing each 8 consecutive slots of the result to get the maximum in the first slot and then duplicating this first slot to the entire block of 8 consecutive slots.

We now proceed with describing our argmax algorithm. Let $s = 2^k$ for some integer $k > 0$, and let $C = Enc(x_0, ..., x_{s-1})$ be a ciphertext with $s = 2^k$ slots. We assume that all values are unique values in the range $[0, 1]$. To support other ranges, one can multiply all values by a proper scale that brings them to the range $[0, 1]$. Appendix B.3 discusses a method to deal with cases in which the input values are not unique.

Our argmax algorithm relies on a hierarchical iterative process, in which each iteration applies a $League_{m\times d}$ operation with increasingly larger values of m. Since we start with an input of s candidate values without any duplicates, in the first iteration we apply $League^*_{2\times1}$ to get an output ciphertext with $\frac{s}{2}$ different values whose values are duplicated 2 times each. Since these are exactly the requirements of the input of the $League^*_{4\times2}$ operation, we then apply the $League^*_{4\times2}$ operation on the output of $League^*_{2\times1}$, to get a new output with $\frac{s}{8}$ different values that are duplicated 8 times each. Because the output values are duplicated 8 times, we can now apply a $League_{8\times8}$ operation to be left with $\frac{s}{64}$ candidate values, duplicated 64 times each. We then continue with applying $League_{m\times m}$ operations such that m is squared each time ($League_{64\times64}$, $League_{4096\times4096}$, etc). When the number of remaining different values becomes smaller or equal to the number of duplicates, we apply a final $League_{m\times d}$ round where m is the number of remaining different values and d is the number of duplicates (possibly $m < d$). The output, C_{max}, of the last league contains the maximal value duplicated along all slots. The argmax indicator is computed by multiplying the output indicators of all league rounds.

Example B.1. Let $C = Enc(x_0, ..., x_{2^{15}-1})$ be a ciphertext encrypting $s = 2^{15}$ different values. The hierarchical argmax algorithm will proceed as follows.

(1) Apply $League^*_{2\times1}$ and get an output ciphertext encrypting $\frac{2^{15}}{2} = 2^{14}$ values, duplicated 2 times each.
(2) Apply $League^*_{4\times2}$ and get an output ciphertext encrypting $\frac{2^{14}}{4} = 2^{12}$ values, duplicated 8 times each.
(3) Apply $League_{8\times8}$ and get an output ciphertext encrypting $\frac{2^{12}}{8} = 2^9$ values, duplicated 64 times each.
(4) Apply $League_{64\times64}$ and get an output ciphertext encrypting $\frac{2^9}{64} = 2^3$ values, duplicated $64^2 = 4096$ times each.
(5) Apply $League_{8\times4096}$ and get the an output ciphertext encrypting 1 value duplicated along all slots. This value is the maximal value of the input.

The above iterative process starts from a league size of 2, then 8 and then it squares the league size on each iteration. We stop when having $(((8^2)^2)^{\cdots})^2 \ge s$. Thus, a total of $O(\log\log s)$ SIMD-comparisons are required to compute the argmax. On the other hand, most other methods that are discussed in Section 2.1 require $O(\log s)$ SIMD-comparisons or more.

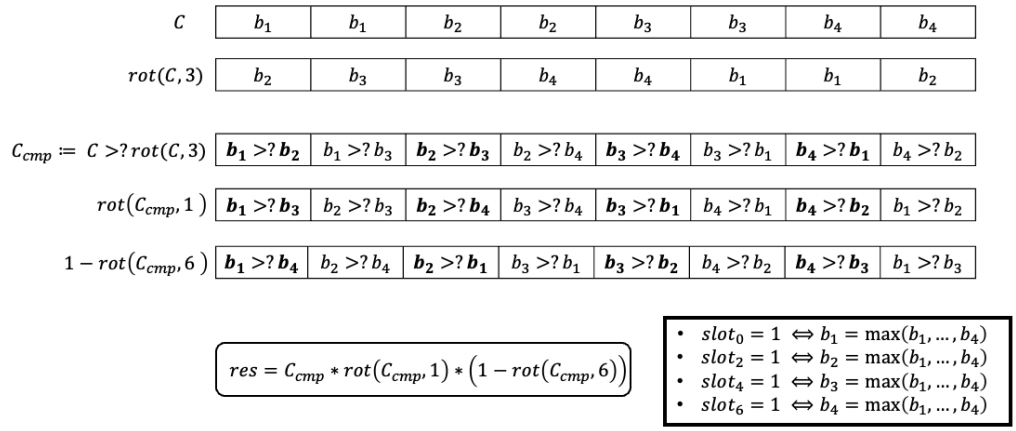


Figure 9: Implementation of $League^*_{4\times2}$ using one SIMD-comparison. The argmax result of the four values in the input ciphertext, C, end up in slots of indexes $0, 2, 4$ and 6 of the result.

Note that our hierarchical algorithm has the expense of extra rotations and ciphertext-plaintext multiplications needed to compute the transpose, C^t, of C, on each league round. However, since comparisons are usually the bottleneck of argmax algorithms (see Table 2), the trade-off of reducing the number of comparison rounds at the expense of adding rotations and ciphertext-plaintext multiplications is beneficial in terms of runtime. See Section 6 for empirical comparison of our method against other argmax methods.

The algorithm above computes the argmax of s values encrypted in a ciphertext having s slots. Using SIMD operations, the algorithm can be easily extended to compute l different argmax operations concurrently on l vectors each having k values, such that $k * l \leq s$ and all vectors are encrypted in the same ciphertext.

Algorithm 5: *hierarchical_argmax*

Input: A ciphertext $C = Enc(x_0, x_1, \ldots, x_s)$ with s slots, encrypting s unique values.
Output: Two ciphertexts, C_{max} and I, containing the maximum result (duplicated along all slots) and the argmax indicator vector, respectively.

$C_{max}, I_1 \leftarrow League^*_{2\times1}(C)$ `// Def. B.2`
$C_{max}, I_2 \leftarrow League^*_{4\times2}(C_{max})$`// Def. B.3`
$I \leftarrow I_1 \cdot I_2$`// Homomorphic mul.`
$d \leftarrow 8, n \leftarrow \frac{s}{8}$
while $n > 1$ **do**
  $m \leftarrow min(n, d)$
  $C_{max}, I_{tmp} \leftarrow League_{m\times d}(C_{max})$
  $I \leftarrow I \cdot I_{tmp}$`// Homomorphic mul.`
  $d \leftarrow d^2, n \leftarrow \frac{n}{m}$
end
return C_{max}, I

B.2 Transpose Computation

Recall that the $League_{m\times d}$ operation for $d \geq m$ (Definition B.1) relies on transposing the leftmost $m \times m$ sub-matrix of an $m \times d$ encrypted matrix whose columns are duplicated. This appendix discusses two possible implementations for this transpose operation. For simplicity, we assume $m = d$. The case $d > m$ can be dealt with by transposing the leftmost $m \times m$ sub-matrix and putting zeros in the other $m \times (d - m)$ entries.

Let C be a ciphertext encrypting an $m \times m$ matrix which we aim to transpose (slots $0, \ldots, m-1$ represent the first row, slots $m, \ldots, 2m-1$ represent the second row, etc.). The columns of the matrix are duplicated, meaning that slots $i * m, i * m + 1, \ldots i * m + (m-1)$ of C all encrypt the same value.

A trivial approach to perform the transpose operation requires $O(m)$ rotations, $O(m)$ ciphertext-plaintext multiplications and a multiplication depth of 1. This is achieved by transposing the first column of C to a row and then duplicating the resulting row along the columns of the matrix. The transpose of one column can be easily done by m ciphertext-plaintext multiplications, m rotations and a multiplication depth of 1, while duplicating one row along the matrix columns can be done using $\log_2 m$ rotations and $\log_2 m$ additions.

As rotations may be costly to perform in FHE, we discuss a second transpose method that requires $O(\log m)$ rotations, 2 ciphertext-plaintext multiplications and a multiplication depth of 2. In this method, we extract the main diagonal of the matrix by multiplying it element-wise with the plaintext unity $m \times m$ matrix. Then, we sum the rows of the resulting product matrix, using the rotate and sum algorithm [34], which requires $\log m$ rotations and $\log m$ additions. The first row of the summation result contains the transpose of the first column of the input matrix (See Figure 10b). We extract the first row of the summation result using element-wise multiplication with a plaintext matrix of zeros and ones, only to duplicate the extracted row along the columns of the matrix, using $\log m$ rotations and $\log m$ additions. To illustrate this transpose process, in Figure 10, we simultaneously transpose two 4×4 matrices that are packed in the same ciphertext. That is, slots $0, 1, \ldots, 15$ contain the first matrix, while slots $16, 17, \ldots, 31$ contain the second matrix. When summing the rows of the matrices using the rotate

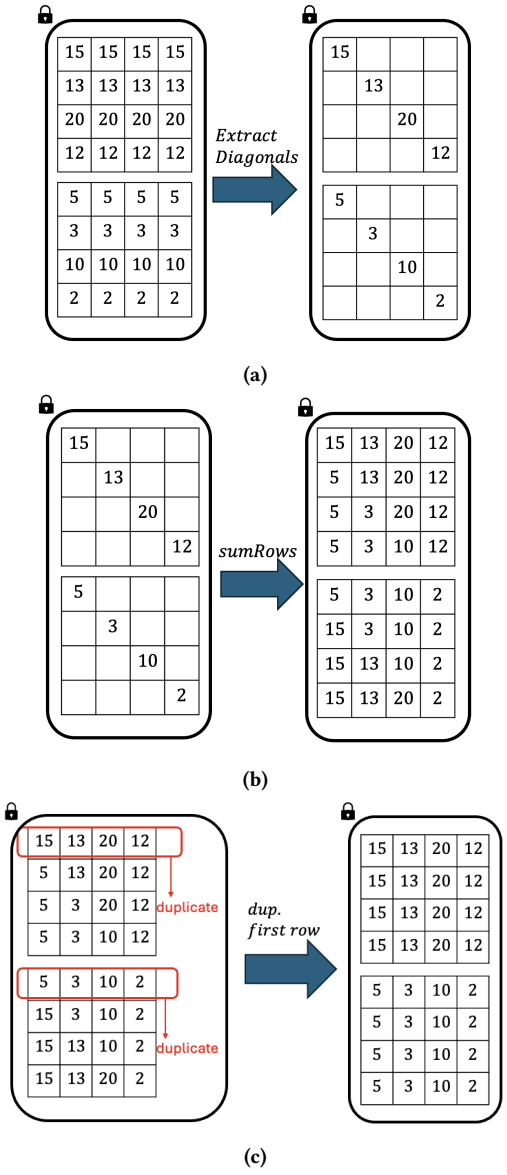


Figure 10: Transposing two matrices whose columns are duplicated. The two matrices are encrypted in the same ciphertext: slots $0, 1, \ldots, 15$ contain the same matrix and slots $16, 17, \ldots, 31$ contain the second matrix.

and sum algorithm (see Figure 10b), the summation result will be located in the first row of each matrix, but the other rows (second, third and fourth) may contain a value from the other matrix. This happens because when rotating the ciphertext upwards during the *sumRows* operation, values can move from one matrix to the other. The second, third and fourth row are then cleared using element-wise multiplication with a plaintext matrix which extracts the first row of each matrix. The extracted first rows are duplicated to get the final transpose result (Figure 10c).

The two suggested transpose approaches offer a trade-off between the number of operations and multiplication depth. The trivial approach uses $O(m)$ rotations, $O(m)$ products and a multiplicative depth of 1. The second approach, on the other hand, uses $O(\log m)$ rotations, 2 products and a multiplicative depth of 2. Empirically, we found that the second approach is faster than the trivial approach, but if the added depth ends up requiring an additional bootstrap operation, then the question of which method is faster depends on the specific value of m.

B.3 Handling Inputs with a Non-unique Maximum Value

This section is relevant to argmax implementations that rely on a polynomial approximation of the compare function as a building block. Polynomial approximations of the compare function (e.g. [21], [41]) usually support comparing two values whose distance from each other is at least ε, for some $\varepsilon > 0$. If the polynomially approximated compare function is applied on two inputs a, b, s.t. $|a - b| < \varepsilon$, the result can be anywhere in the range $(0, 1)$, and it is usually close to 0.5 if $a = b$.

Our hierarchical argmax algorithm and most of the algorithms explained in Section 2.1 compute an argmax one-hot indicator vector as a product of temporary indicator vectors resulting from the compare function. Thus, if the output of the compare function is not very close to 0 or 1, the resulting argmax one-hot indicator will contain values that are not close to 0 or 1. More precisely, if we apply our argmax algorithm on an input vector $\mathbf{X}$ whose maximum value appears t times, then the argmax one-hot indicator result will contain a value close to 0 in the places of non-maximal values, and it will contain a value close to 0.5^t in the places of the maximal values.

In many use cases, the argmax indicator output is required to be a one-hot vector containing only one value of 1 and zeros in all other places. This may be desired, for example, when using the argmax one-hot indicator result as a selector to select exactly one value from a set of candidates. Thus, some post-processing on the argmax output is required in order to ensure the output of the argmax is a one-hot indicator vector, when the maximal value in the input vector $\mathbf{X}$ is not known to be unique.

Assume the maximal value of the input vector $\mathbf{X}$ appears at most t times. In this section, we offer a method to convert the argmax output to a one-hot indicator vector. The larger t is, the harder it becomes to maintain the correct argmax result. Thus, we suggest adding a small random noise to the input vector $\mathbf{X}$ before starting the argmax algorithm. This random noise should be small enough so it does not alter the order of the values within $\mathbf{X}$. If we know that different values in $\mathbf{X}$ are far from each other by at least $\varepsilon > 0$, then adding random noise in the range $(0, \varepsilon)$ mitigates the problem of a repeated maximum value. This still does not solve the problem completely, because even after adding random noise, some of the previously equal values of $\mathbf{X}$ may become so close so their distance from each other is smaller than the sensitivity of the compare approximation being used, meaning that comparing them to each other would produce an output value close to 0.5.

After adding random-noise to the input, we apply the argmax algorithm as usual and then proceed with the following post-processing step. Let I be the argmax output that we want to post-process. I contains a positive value in the indexes of the maximal values of $\mathbf{X}$

and 0 in all other places. We apply the idea of SPiRiT [5] to keep one of the positive values in a vector while zeroing all other positive values. While the SPiRiT algorithm uses the BGV [16] FHE scheme and works in a scenario in which each element of the input vector is encrypted in a separate ciphertext, we extend the idea of SPiRiT to support the CKKS scheme and to deal with an input vector whose inputs are all encrypted in the same ciphertext.

Our post-processing step of the argmax output is summarized by Algorithm 6. First, we execute the operation $C := isGreater(I, \delta)$ for some fixed parameter $\delta > 0$ (for example, we may chose $\delta = 0.5^{t+1}$). This comparison converts all values of I that are greater than δ to 1, while zeroing all other values. Then, we compute a new ciphertext, C_{SP}, containing the sums of prefixes of the slots of C. That is, the i-th slot of C_{SP} contains the sum of the slots $0, 1, \ldots, i$ of C. C_{SP} is computed using the algorithm of [35], which is summarized by Algorithm 7 and requires $\log_2 s$ rotations, $\log_2 s$ additions and $\log_2 s$ ciphertext-plaintext multiplications. After computing C_{SP}, the post-processed one-hot argmax indicator can be computed as $C * isEqual(C_{SP}, 1)$, where the *isEqual* operation is performed slotwise and it returns 1 for each slot that is equal to 1 and 0 to all other slots. $isEqual(a, b)$ can be computed as $1 - sign(a - b)^2$, where the *sign* function is defined to output 1 for a positive, -1 for a negative input and 0 for the zero input, and it can be computed following the methods discussed in [21]. The result of the product $C * isEqual(C_{SP}, 1)$ is a one-hot vector containing 1 in the first slot of I which is greater than δ, and containing 0 in all other places. Figure 11 illustrates this post-processing algorithm.

Algorithm 6: *postprocess_argmax*

Input: A ciphertext $I = Enc(i_0, i_1, \ldots, i_{s-1})$, which is the output indicator vector of our argmax Algorithm and a parameter $\delta > 0$.

Output: The post-processed argmax one-hot indicator.

$C \leftarrow isGreater(I, \delta)$ `// element-wise isGreater.`

$C_{SP} \leftarrow sum\_prefixes(I)$ `// Algorithm 7.`

return $C * isEqual(C_{SP}, 1)$

Algorithm 7: *sum_prefixes*

Input: A ciphertext $C = Enc(c_0, c_1, \ldots, c_{s-1})$ with s slots, where s is a power of 2.

Output: A ciphertext, C_{SP} whose $i - th$ slot is equal to $c_0 + c_1 + \ldots + c_i$.

for $i \leftarrow 0$ **to** $\log_2(s) - 1$ **do**

  $C_{tmp} \leftarrow rot(C, -2^i)$ `// Rotate right by` 2^i `slots.`

  $mask(C_{tmp}, 2^i)$ `// Zero first` 2^i `slots of` C_{tmp}.

  $C+ = C_{tmp}$

end

return C

B.4 Experiments on Handling a Non-unique Maximal Value

In this section we evaluate the post-processing algorithm discussed in Appendix B.3. We applied our argmax algorithm on an input vector $\mathbf{X} = \{x_0, x_1, \ldots, x_{n-1}\}$ with varying input sizes. The maximal value in $\mathbf{X}$ was repeated $min(50, \frac{n}{10})$ times, and it was located in random indexes. Table 6 shows the runtime of the post-processing algorithm (Algorithm 6) on several sizes of the input vector $\mathbf{X}$. Due to the *sign* function being noisy when comparing equal values and because of FHE inherent noise, there were rare cases in which the algorithm didn't produce the correct result. However, even in these cases, the resulting post-processed argmax one-hot indicator vector contained 1 in an index of a value of $\mathbf{X}$ that is close to the maximum even though it is not the maximal value itself. To verify that the success rate of Algorithm 6 is high, we repeated the experiments tens of times for large inputs (e.g. $n >= 4096$) and up to thousands of times for smaller inputs (e.g. $n <= 256$), and we counted the number of successful runs of the algorithm. We considered a run to be successful if the resulting one-hot indicator vector contained a value very close to 1 in one of the indexes that contain the maximal value of $\mathbf{X}$, and if all other values of the resulting one-hot indicator vector were very close to 0. Table 6 shows that the success rate of Algorithm 6 is very high and that failures are very rare.

n (input size)	runtime (ms)	success rate
128	593	99.86%
256	583	99.79%
512	605	99.86%
1024	691	100%
2048	585	99.43%
4096	673	100%
8192	683	97.73%
16384	681	100%
32768	773	100%

Table 6: Runtime of Algorithm 6

C Reshape Operations on Tile Tensors

Let $M[a, b]$ be a matrix and $V[b, 1]$ be a vector. Let s be the number of slots in the used FHE configuration and suppose we are given the tile tensors $T_{M^T}[\frac{b}{s}, \frac{a}{1}]$ and $T_V[\frac{b}{s}, 1]$ that encrypt M^T and V respectively using one-dimensional tiles. As discussed in Section 3.3.2, the matrix-vector multiplication of M and V can be more efficiently computed if T_{M^T} and T_V were packed with a two-dimensional tile shape $[t_1, t_2]$, where $t_2 > 1$. To this end, we aim to reshape T_{M^T} and T_V into two dimensional tile-tensors with the tile shape $[t_1, t_2]$, using as least *FHE* operations as possible. Herein, we assume the indexing of the slots of the tiles follows a last-order convention. That is, the value at index (i_1, i_2) in a tile whose tile shape is $[t_1, t_2]$ is stored in the slot at index $i_1 * t_2 + i_2$ of the corresponding ciphertext.

Let $1 < t_2 < s$ be a power of 2 and denote $t_1 = \frac{s}{t_2}$. We start by discussing a method to reshape T_V to a tile tensor $T'_V[\frac{b}{t_1}, \frac{1}{t_2}]$ that encrypts a the same content as T_V. The trivial method of reshaping T_V would require $O(b)$ rotations and ciphertext-plaintext multiplications to properly rearrange the slots of T_V, while our proposed method only requires $O(\frac{b}{s} * t_2)$ rotations and ciphertext-plaintext multiplications. Our method relies on the distinction that if we

$I =$

0.25	0	0	0.25	0.25	0	0.25	0

$C =$

1	0	0	1	1	0	1	0

$C_{sum} =$

4	4	4	4	4	4	4	4

$C_{SP} =$

1	1	1	2	3	3	4	4

$C_{SP} + C =$

2	1	1	3	4	3	5	4

$isGreater(C_{SP} + C, C_{sum}) =$

0	0	0	0	0	0	1	0

Figure 11: Post-processing the argmax result, I, to get a one-hot indicator vector. C is computed as $isGreater(I, 0.125)$. C_{SP} contains the sum of prefixes of the slots of C.

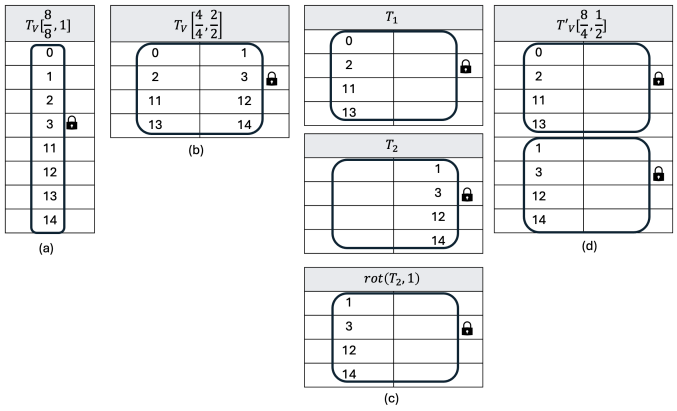


Figure 12: Reshaping of a one dimensional tile tensor, $T_V[\frac{8}{8}, 1]$ to a two-dimensional tile tensor $T'_V[\frac{8}{4}, \frac{1}{2}]$. Note that the rows of T'_V are a permutation of the rows of T_V. This permutation allows a more efficient reshape process.

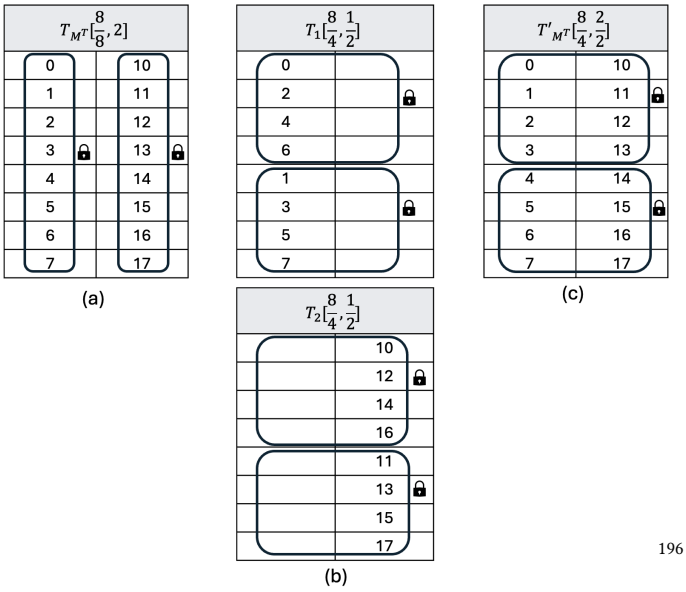


Figure 13: Reshaping of the tile tensor $T_{M^T}[\frac{8}{8}, 2]$, which encrypts a 8×2 matrix, into the tile tensor $T'_{M^T}[\frac{8}{4}, \frac{2}{2}]$. Note that the rows of T'_{M^T} are a permutation of the rows of T_{M^T}. This permutation allows a more efficient reshape process

permute the rows of V and the rows of M using the same permutation, the matrix-vector multiplication result will not be affected. Therefore, instead of trivially reshaping T_V and T_{M^T}, we reshape them to T'_V and T'_{M^T} that encrypt permuted versions of $V[b, 1]$ and $M^T[b, a]$, respectively.

Let us first consider the case $b = s$. That is, $T_V[\frac{b}{s}, 1]$ is encrypted using exactly one ciphertext. In this case, we start by changing the metadata of T_V so its shape becomes $[\frac{t_1}{t_1}, \frac{t_2}{t_2}]$. This step only alters the metadata of the tile tensor T_V, while the encrypted data stays the same. Following this metadata update, T_V can be thought of as one $t_1 \times t_2$ tile, as shown in Figure 12(b). Next, we extract each column of the t_2 columns of T_V to a separate ciphertext, using t_2 ciphertext-plaintext multiplications with plaintext vectors of 0s and 1s. Each extracted column is rotated so it becomes the leftmost column in a $t_1 \times t_2$ tile, as in Figure 12(c). The t_2 resulting tiles are concatenated together to form one tile tensor, $T'_V[\frac{s}{t_1}, \frac{1}{t_2}]$, which is the final reshaped result, as in Figure 12(d). Since we applied one rotation and one ciphertext-plaintext multiplication per each column of T_V, we used $O(t_2)$ rotations and ciphertext-plaintext multiplications in total.

For the more general case in which b is not necessarily equal to s, we iterate over all $\lceil \frac{b}{s} \rceil$ one-dimensional tiles of T_V, reshape each tile individually and concatenate all of the reshaped results along the first dimension to form the final reshaped tile tensor. This requires $O(t_2)$ operations per tile, or $O(\frac{b}{s} * t_2)$ operations for reshaping the entire tile tensor.

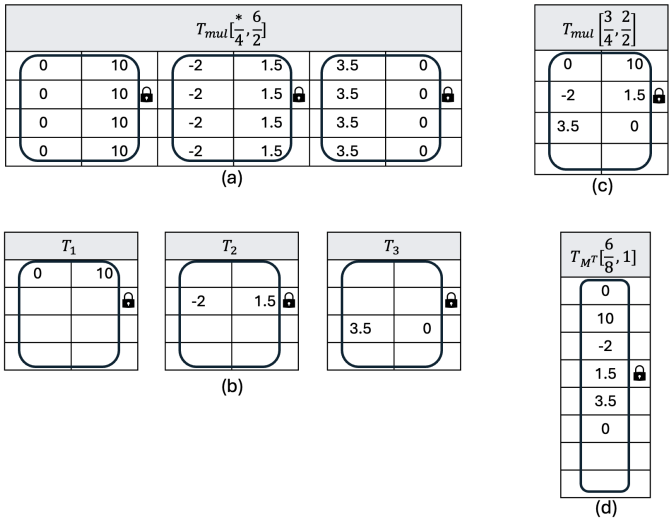


Figure 14: Reshaping the matrix multiplication result $T_{mul}[\frac{*}{4}, \frac{6}{2}]$ into $T_{mul}[\frac{6}{8}, 1]$.

A similar method is conducted to reshape the matrix tile tensor, T_{M^T}. Since $T_{M^T}[\frac{b}{s}, \frac{a}{1}]$ can be thought of as a different column tile tensors of the shape $[\frac{b}{s}, 1]$ (see Figure 13(a)), we can reshape each such a column to the shape $[\frac{b}{t_1}, \frac{1}{t_2}]$ separately as we did with T_V. However, this time instead of storing all values in the left-most column of $t_1 \times t_2$ tiles, when reshaping the i-th column of T_{M^T}, we store it in the column of index $i \mod t_2$ in the reshaped result (see Figure 13(b)). Each set of t_2 reshaped results are then added together to form a single tile tensor of the shape $[\frac{b}{t_1}, \frac{a}{t_2}]$ (see Figure 13(c)).

After obtaining the reshaped tile tensors $T'_V[\frac{b}{t_1}, \frac{1}{t_2}]$ and $T'_{M^T}[\frac{b}{t_1}, \frac{a}{t_2}]$, we duplicate T'_V along the second dimension, so it will have a tile tensor shape of $[\frac{b}{t_1}, \frac{*}{t_2}]$. Then, the matrix-vector multiplication MV may be computed as $sum(T'_{M^T} * T'_V, 1)$, where $T'_{M^T} * T'_V$ stands for the element-wise multiplication between T'_{M^T} and a broadcast copy of T'_V, and the *sum* operation is a summation over the first dimension. As explained in [4], due to a feature of the tile tensor data-structure, when summing values over the first dimension, the obtained result becomes duplicated along the first dimension. Thus, the obtained matrix-vector multiplication result is represented by a tile tensor $T_{mul}[\frac{*}{t_1}, \frac{a}{t_2}]$.

C.1 Reshaping Back to a Column Tile Tensor

The above matrix-vector multiplication result, $T_{mul}[\frac{*}{t_1}, \frac{a}{t_2}]$, is represented by a tile tensor having two dimensional tiles with a tile shape of $[t_1, t_2]$. However, to allow further processing, the XGBoost training algorithm requires the matrix-vector multiplication result to be packed using a one-dimensional tile tensor of the shape $[\frac{a}{s}, 1]$. To this end, we need to reshape the matrix-vector multiplication result to the desired shape of $[\frac{a}{s}, 1]$. The fact that $T_{mul}[\frac{*}{t_1}, \frac{a}{t_2}]$ is duplicated along the first dimension enables us to reshape it without any rotations. To do so, We iterate over the $\lceil \frac{a}{t_2} \rceil$ tiles of T_{mul}, and we extract the row of index i of the i-th tile, using one ciphertext-plaintext multiplication (see Figure 14(b)). The extracted tiles are then added together, resulting with multiple tiles of the tile shape $[t_1, t_2]$. These tiles are concatenated along the first dimension, forming a two dimensional tile tensor (see Figure 14(c)). Finally, we manually update the metadata of the resulting tile tensor to the one dimensional tile tensor shape of $[\frac{a}{s}, 1]$, without altering the encrypted data, to get the required reshape result (see Figure 14(d)).

D Optimizations of Our XGBoost Training Algorithm

In this section, we discuss methods to optimize the efficiency of our algorithm for XGBoost training over FHE as it was presented in Section 5. Three categories of optimizations are presented: compressed packing methods, utilizing the complex part of the ciphertexts and reducing time spent on bootstrapping. We will now describe each of these three categories in more details and explain how they enhance the efficiency of the XGBoost training over FHE.

Our first optimization relies on the fact that nodes of the same level of the XGBoost trees are independent and thus can be trained in the same time using the SIMD capability of FHE schemes. Whenever we are processing tensors whose size is smaller than the number of slots, we can compress tensors relating to nodes of the same level in the tree together in the same ciphertext. Thus, in the i-th level of the tree we can pack data relating to up to 2^i nodes. There are two specific points in our XGBoost training algorithm where such compression can be useful. First, when computing gain values corresponding to all candidate splits (Algorithm 4), if the number of candidate splits is smaller than the number of slots, we can compress tensors of multiple nodes together. This parallelizes the division operations that are needed in the gain expression from Equation 6, as well as the argmax operations that are applied on the resulting gains. Second, recall from Equation 5 that computing the output values of the nodes requires a costly division operation. Each node has two scalars that should be divided to get its output value: $\sum_{i \in I_j} g_i$ and $\sum_{i \in I_j} h_i + \Lambda$. Instead of doing the division between these two scalars once per node, we can compress the numerators of all nodes from the same level in one ciphertext, compress the denominators in another ciphertext and perform one division for all nodes of the same level at once.

The second optimization relates to utilizing the complex part of the FHE ciphertexts. Ciphertexts in CKKS scheme [20] encrypt complex values. We can store two real values a, b together as $a + ib$ in one slot of a ciphertext. The drawback of this complex packing is that it limits the operations that we are able to perform on the packed data. For example, multiplying two complex-packed values, $(a + ib) * (c + id)$, does not give us the two multiplications $a * b$ and $c * d$ and thus we can not multiply two complex-packed ciphertexts. Summations and multiplying by real values, on the other hand, are supported on complex packed ciphertexts. For example, when summing $a + ib$ with $c + id$, we get $a + b + i(c + d)$, which contains $a + b$ in the real part and $c + d$ in the imaginary part.

Even though not all operations can be performed on complex packed data, complex packing can be used to speed-up the gain computation step of Algorithm 3 almost by a factor of 2. Recall that Algorithm 3 receives an encrypted input indicator vector, $T_I[\frac{n}{s}, 1]$, that contains 1 in the indexes of the samples that have reached

the current node and 0 in all other places. Using the complex-packing optimization, we can complex pack different input masks corresponding to pairs of nodes of the same level of the XGBoost-tree together. So, two tile tensors, T_{I_1} and T_{I_2}, are complex packed as $T_{I_c} = T_{I_1} + i * T_{I_2}$, while the rest of Algorithm 3 stays the same. Because Algorithm 3 merely consists of multiplying with real values and summations, it can be performed on complex packed data, virtually computing the required sums of gradients for two nodes at once. At the end of the algorithm, we unpack the two complex packed results using the equations $real(z) = 0.5 * (z + conj(z))$ and $imag(z) = 0.5 * (z - conj(z))$.

The third optimization we discuss reduces the time spent on bootstrapping operations. In the CKKS scheme, ciphertexts are encrypted at a specific chain index which gets smaller following each multiplication. If the chain index reaches a certain threshold, a costly bootstrap operation must be applied to restore the ciphertext to have a higher chain index. While bootstrapping is necessary when a ciphertext reaches a certain chain index, in some cases, choosing to apply bootstrapping beforehand saves the total number of required bootstrap operations. For example, if we want to evaluate a polynomial on an encrypted input x which has only one chain index left, it is more efficient to bootstrap x before evaluating the polynomial and starting the evaluation on a fresh ciphertext with high chain index, because otherwise we would have to bootstrap each monomial of the polynomial separately. Choosing the optimal locations of where bootstrapping should be applied to minimize the overall number of bootstrap operations is an NP-hard problem [50]. However, by manual trial and error one can find specific points where bootstrapping is beneficial. Thus, our XGBoost training algorithm includes manual optimization of the bootstrap points to reduce the overall number of required bootstrap operations. Some examples of recommended points for manual bootstrapping include:

- Bootstrapping the tile tensor encrypting the gain values before we apply an argmax operation on it (Algorithm 2, Line 2) can help reducing the number of bootstrap operations during the argmax operations.
- Bootstrapping the output one-hot indicator of the argmax operation mentioned above can reduce the number of bootstrap operations during later stages that rely on this one-hot indicator.
- Bootstrapping the input masks of the children obtained at the end of the node splitting algorithm (Algorithm 2, Line 5) to start the next recursive splits with fresh input masks can be beneficial.

Aside from manual planning of bootstrap points, we can reduce the time spent on bootstrapping in other ways. One way to do this utilizes complex-packing to bootstrap two ciphertexts on the same time. The tile tensor data structure offers an automatic optimization in which if it is bootstrapped while having more than one tile, pairs of tiles are complex packed together so the number of tiles is halved. The complex packed tiles are bootstrapped, and the bootstrapped result is then unpacked. Because complex packing and unpacking are much less costly than bootstrapping, this optimization is beneficial.

Another way to reduce the bootstrap time relies on specific details of the HEaaN library implementation [20] of the CKKS scheme, which we used in our training. There are two kinds of bootstrapping operations implemented in the HEaaN library. The first is the regular bootstrapping, which supports input ciphertexts whose encrypted values fall in the range $(-1, 1)$. The second is the *exteneded bootstrapping*, which supports values in the range $(-2^{20}, 2^{20})$, but it is twice slower than the regular bootstrapping. Generally, because bootstrapping is applied on encrypted values, we can not guarantee that the bootstrapped values fall in the range $(-1, 1)$, which requires us to use the slower extended bootstrapping. However, there are specific points in the XGBoost training algorithm in where we have an additional knowledge on the values. For example, when comparing two values, the compare algorithm naturally produces an output of 0s and 1s, and thus its output can be bootstrapped using the regular bootstrapping. Moreover, the input indicator vectors used during the algorithm also consist of vectors of 0s and 1s, meaning that they can be bootstrapped using the regular bootstrapping instead of the extended bootstrapping as well. Thus, combining the two kinds of bootstrapping and using the regular bootstrapping wherever possible is yet another optimization of our algorithm.

E Further Discussion on XGBoost Training Experiments

This Appendix includes further experimental results and additional discussions to the experiments presented in Section 6.3. We discuss the runtime breakdown of our algorithm, memory consumption details, runtime comparison with works that use multi-party communication techniques and the impact of our hierarchical argmax algorithm on the overall XGBoost training runtime.

E.1 Runtime Breakdown of our Algorithm

Figure 15 shows the runtime breakdown of our algorithm. Computing the gain values (Algorithm 2, Line 1) is the slowest step in the training algorithm. This is due to the fact that we have to compute a separate gain value for each sample of the dataset, and this process is repeated one time for each node in the XGBoost model. The argmax step is fast for datasets that include few candidate splits (for example, the Sepsis dataset contains only 105 candidate splits) and it is slow for datasets that include many candidate splits (for example, the Wine dataset contains 1095 candidate splits). The "other" category includes, for example, bootstrap operations that happen outside of other reported operations, moving ciphertexts between GPU and CPU and changing the shape of the tile tensors as explained in Section 5.1.

E.2 Memory Consumption

Our experiments consume up to 80GB of GPU memory and up to 900GB of CPU memory. Some of the temporary ciphertexts computed during the training algorithm add a memory overload (e.g. Algorithm 1, Line 1). However, this RAM consumption happens solely in the server side which is assumed to have high computation power. Before returning the final trained model to the client side, the model can be compressed in few CKKS ciphertexts. Indeed, to store an XGBoost model we need two values per internal node (i.e. the feature index and the comparison threshold used in the split condition of the corresponding node) and one value per leaf node (i.e. the output value). A typical XGBoost model of 100 trees of depth 6 contains 6300 internal nodes and 6400 leaf nodes and hence it can

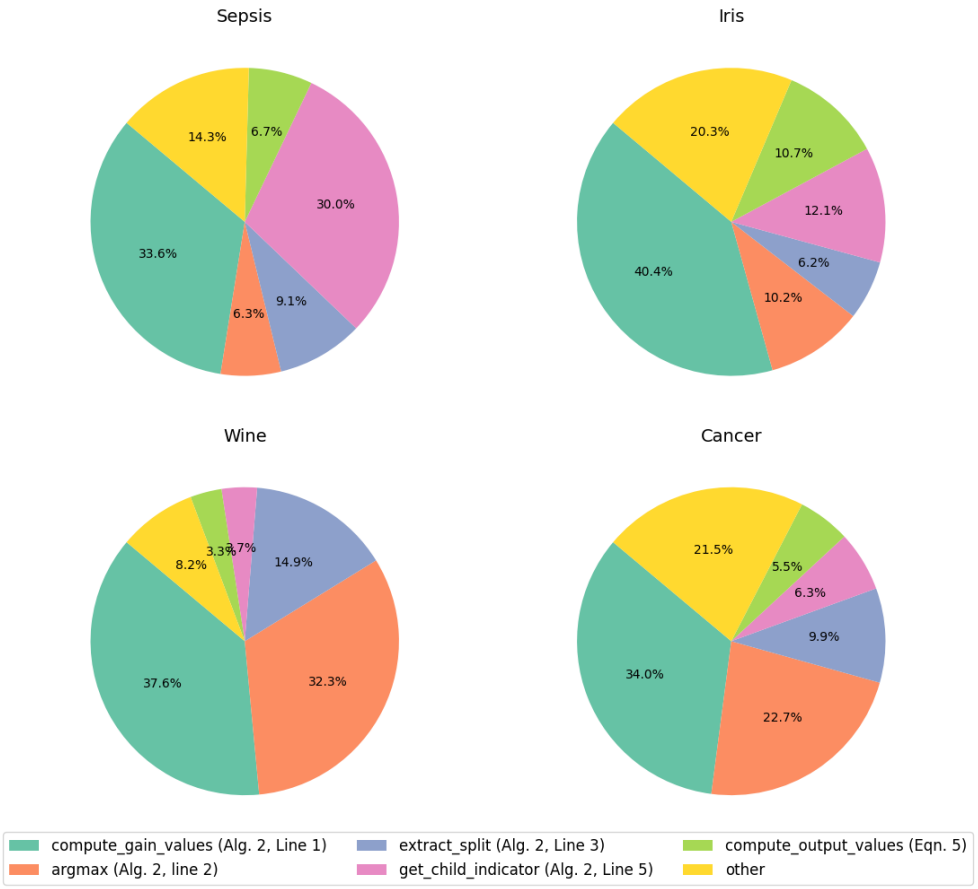


Figure 15: Timing breakdown for the experiments of Figure 7. The "other" category includes, for example, bootstrap operations that happen outside of other reported operations, moving ciphertexts between GPU and CPU and changing the shape of the tile tensors as explained in Section 5.1.

be stored using one CKKS ciphertext of $32K$ slots. The size of one CKKS ciphertext with $32K$ slots at level 1 is about 2MB, meaning that our XGBoost model can be compressed and returned to the client using only few megabytes of communication. In contrast, works that use multi-party communication and rely on interaction between the server and the client, such as [37], require at least 50GB of interaction to train a model of 100 trees of depth 6 on a dataset of 10^5 samples.

To setup the system, the public FHE keys (encryption key, rotation keys and evaluation keys) should be sent from the client to the server, costing 4GB of communication. However, this communication happens offline and it is one time only, meaning that the same keys can be used for multiple FHE tasks, including training of several XGBoost models or other FHE computations. Deploying the encrypted input dataset also requires (one round of) communication between the client and the server. The encrypted datasets in our experiments have a size of no more than 100MB. Further examinations and optimizations of the memory consumption of our XGBoost training algorithm are left as an are for future work.

E.3 Runtime Comparison with Multi-party Computation Works

As discussed in Section 2.2.2, Multi-Party Computation (MPC) approaches for the secure training of tree-based models have the disadvantage of requiring the client to stay active during the computation and inducing a high communication cost, which is less suitable for clients with weak computation power as common in typical deployment scenarios. However, we report the runtime of two of the recently published MPC solutions for completeness and clarity. Recall from Table 5 that our FHE training algorithm takes up to 5.5 hours to train a model of 100 trees of depth 6 on a dataset of up to hundred thousand samples on a GPU machine.

Lu et al. [37] report training a single tree of depth 6 on a dataset of 10^5 samples in 18.4 seconds in LAN network settings and 39.3 seconds in WAN network settings. By extrapolation we conclude that to train a model of 100 trees of depth 6 it would take them 0.5 hour in LAN settings and 1.1 hour in WAN settings. However, their work 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,

and this disclosure of information was shown by Zhu et al. [63] to potentially allow an attacker to infer training data.

More recently, Bhardwaj et al. reported using a multi-party communication approach to train a tree of depth 4 on a dataset of 2^{13} samples in 5 seconds in LAN settings and 152 seconds in WAN settings, meaning that training a model of 100 trees of depth 4 would take them 500 seconds in LAN settings 4.2 hours in WAN settings.

E.4 Impact of our Argmax Algorithm on Overall Training Time

It can be seen from Figure 15 that the argmax step costs between 6% (for the Sepsis dataset) and 32% (for the Wine dataset) from the entire training process. In the Sepsis dataset, the number of candidates split points is only 105, and we used the league method to compute the argmax. In the Wine dataset, there are 1095 candidate split points, and we used our hierarchical argmax algorithm to compute the argmax. That is, the argmax step is slower and heavier to compute for the Wine dataset, because it contains more candidate split values.

For the Wine dataset, the argmax step costed 120 seconds for each tree of depth 6. When applying argmax on 1095 as in the Wine dataset, our argmax algorithm is faster than the prior argmax algorithms by 1.5× (1834 milliseconds for our algorithm vs. 2748 milliseconds for the four-way tournament algorithm [51]). This means that by using our argmax algorithm we saved 60 seconds per tree, or about 1.7 hours in total for training a typical model of 100 trees of depth 6. A 1.7 hours reduction is about 16% runtime reduction from the total training time for 100 trees of depth 6 on the Wine dataset. In general, when using datasets that contain more candidate split points, the portion of the argmax step from the entire training process becomes larger and the runtime saved by using our argmax algorithm becomes larger.