Faster Private Decision Tree Evaluation for Batched Input from Homomorphic Encryption

Kelong Cong[1*], Jiayi Kang[2], Georgio Nicolas[2], and Jeongeun Park[3*]

[1] Zama, Paris, France
kelong.cong@zama.ai
[2] COSIC, KU Leuven, Leuven, Belgium
firstname.lastname@esat.kuleuven.be
[3] Norwegian University of Science and Technology (NTNU), Trondheim, Norway
jeongeun.park@ntnu.no

Abstract. Privacy-preserving decision tree evaluation (PDTE) allows a client that holds feature vectors to perform inferences against a decision tree model on the server side without revealing feature vectors to the server. Our work focuses on the non-interactive batched setting where the client sends a batch of encrypted feature vectors and then obtains classifications, without any additional interaction. This is useful in privacy-preserving credit scoring, biometric authentication, and many more applications.
In this paper, we propose two novel non-interactive batched PDTE protocols, BPDTE_RCC and BPDTE_CW, based on two batched ciphertext-plaintext comparison algorithms, our batched range cover comparison (RCC) comparator and the constant-weight (CW) piece-wise comparator, respectively. When comparing 16-bit batched encrypted values to a single plaintext value, our comparison algorithms show a speedup of up to 72× compared to the state-of-the-art Level Up (CCS'23). Moreover, we introduced a new tree traversal method called adapted SumPath, to achieve $\mathcal{O}(1)$ complexity of the server's response, whereas Level Up has $\mathcal{O}(2^d)$ complexity for a depth-d tree and the client needs to look up classification values in a table. Overall, our PDTE protocols attain the optimal server-to-client communication complexity and are up to 17× faster than Level Up in batch size 16384.

Keywords: Machine learning · Private Decision Tree Evaluation · Homomorphic encryption.

1 Introduction

In the era of big data, machine learning (ML) has emerged as a powerful tool to connect data and extract valuable information. Many well-known companies such as Amazon, Microsoft and IBM are present in this market by providing machine learning as a service (MLaaS). Namely, the cloud server holds a pre-trained

[*] Work partially done while the author was at COSIC, KU Leuven.

machine learning model and provides useful service by performing inference with clients' data.

However, clients' data may be confidential and sharing it in the clear with the server can threaten their privacy. This leads to rising interests in privacy-preserving machine learning protocols [34,12,11,23]. This work focuses on Private Decision Tree Evaluation (PDTE) [29,11,23,1,18,28], where the server holds a decision tree classification model and the client obtains the inference result without revealing the input data.

In particular, the focus of this work is on *non-interactive batched* PDTE. Non-interactive implies the client sends a query and receives the output without additional interactions with the server. This allows the client to stay offline during the evaluation process and achieve full outsourcing. Two recent works, SortingHat [11] and Level Up [23] use homomorphic encryption (HE) for non-interactive PDTE. In particular, SortingHat uses schemes such as TFHE [10], FINAL [3] and outperforms for single-query scenarios, while Level Up employs the levelled BFV [5,14] scheme, which supports homomorphic evaluations in a SIMD (Single-Instruction Multiple-Data) manner.

Batched PDTE allows evaluations of the same decision tree for multiple samples in parallel. Precisely, for a fixed decision tree held by the server and a client with multiple feature vectors as inputs, batched PDTE allows the client to send and receive once, instead of sending these feature vectors over and over to get the inference result of each. This could be useful in PDTE applications, e.g., when a bank outsources a credit-scoring decision tree and needs evaluations for various applicants without revealing their profiles [32,9,19].

Our work focuses on the batched PDTE using BFV, and our newly-proposed protocols, BPDTE_RCC and BPDTE_CW, outperform Level Up for large batch sizes (e.g. > 2100). Since PDTE consists of ciphertext-plaintext comparisons in decision nodes and a tree traversal procedure for aggregation, these building blocks are improved in Section 3 and Section 4, respectively. In Section 3, we propose two batched ciphertext-plaintext comparisons, our batched RCC comparator and the constant-weight piece-wise comparator, which are based on the prior RCC comparator [23] and folklore bit-wise comparator [15,22,23]. By fully exploiting the fact that batched encrypted values are compared to a single plaintext value, we achieve up to over $72\times$ speedup for 16-bit numbers while maintaining a low multiplicative depth.

Moreover, Level Up uses SumPath for tree traversal, where the amortized response of the server is $\mathcal{O}(2^d)$ for a decision tree of depth d and the client needs to look up classification values in a table. This further restricts the extension of decision tree evaluations to tree ensembles. Therefore, we introduce an adapted SumPath in Section 4, where the amortized response of the server is $\mathcal{O}(1)$ at the cost of $\mathcal{O}(\log_2 d)$ multiplicative depth. By combining the adapted SumPath with batched ciphertext-plaintext comparisons, our two batched non-interactive PDTE protocols, BPDTE_RCC and BPDTE_CW, avoid the client looking up classification values and are also up to $17\times$ faster than Level Up in batch size 16384.

1.1 Related Work

In interactive PDTE, the client and server communicate multiple rounds and perform a secure two-party computation. Previous protocols in [8,4,30,2] fall into this category, and an enlightening survey of PDTE was presented in [18]. With sufficient bandwidth, decision tree training is also feasible, as in [33,20]. Interactive protocols, however, do not support computation outsourcing since the client needs to be online during the evaluation.

For non-interactive PDTE, SortingHat and Level Up are the respective state-of-art using non-batched FHE such as TFHE and batched data via BGV/BFV. Other prior works include [31] and [28] using additive homomorphic encryption, [21] that improves non-interactive comparisons, [1] that uses private information retrieval (PIR) in tree traversal, and Tueno et al. [29] that firstly made non-interactive PDTE practical. A concurrent work [26] evaluates binary decision trees in a ciphertext-ciphertext operation setting based on CKKS and proposes a decision tree training method. Their protocol uses the SIMD packing method to run a protocol per an input efficiently by mapping one tree model into one ciphertext, therefore, the purpose of using SIMD packing is different to ours.

2 Preliminaries

2.1 Notation

Bold symbols such as $\mathbf{a}$ denote arrays of elements. The notation $\mathbf{a}[i]$ denotes the i-th element in $\mathbf{a}$, and $\mathbf{a}[i,j]$ denotes the sub-array from the i-th element to the j-th element (both inclusive) in $\mathbf{a}$. The first element in the array has index 1. The notation $\mathbb{1}_f$ denotes the binary output of evaluating the condition f, which equals 1 if f holds and 0 otherwise.

2.2 Decision Trees

A decision tree represents a function $\mathcal{T} : X \longrightarrow \{0,\dots,k-1\}$ which maps an n-dimensional feature vector into a classification value. The function $\mathcal{T}$ contains m *decision nodes* organized hierarchically in depth d, together with $m+1$ *leaves*, each associated with a value in $\{0,\dots,k-1\}$. Table 1 presents a complete list of symbols used in a decision tree.

The decision tree evaluation amounts to traversing a path from the root node to a resulting leaf, whose associated classification value is returned as the output. Precisely, each decision node compares an input feature x_i to a pre-trained threshold value y_j, yielding $b \leftarrow \mathbb{1}_{x_i \geq y_j}$. If $b = 1$, the evaluation proceeds to the right child node; otherwise, it moves to the left child node. As such, the evaluation path contains at most d decision nodes and ends up in an *output leaf*, whose corresponding classification value is returned.

2.3 Levelled Homomorphic Encryption

Levelled homomorphic encryption (LHE) such as BGV [6] and BFV [5,14] allows evaluations of bounded-depth circuits without knowing the secret key. In practice, applications with higher multiplicative depth necessitate larger LHE parameters, consequently resulting in higher communication, storage and computation costs. Hence, algorithms with reduced multiplicative depth are preferred for LHE.

For $\mathsf{BGV}/\mathsf{BFV}$, the ring $R = \mathbb{Z}[X]/\left(X^N + 1\right)$ where N is a power of 2 is widely used. With a plaintext modulus t and a ciphertext modulus $q \gg t$, the plaintext space is $R_t = R/tR$ and the ciphertext space is $R_q \times R_q$ where $R_q = R/qR$. For a prime t that satisfies $t \bmod 2N = 1$, the polynomial $\left(X^N + 1\right)$ splits into N linear factors modulo t. Therefore, according to the Chinese Reminder Theorem, there exists an isomorphism $R_t \cong \mathbb{F}_t^N$ between the plaintext space R_t and N copies of $\mathbb{F}_t$, with each termed a *slot* [27]. This enables encoding and encrypting messages in N slots into a single ciphertext and performing homomorphic operations over encoded values in a SIMD manner.

2.4 PDTE and Tree Traversal

Suppose the server holds a pre-trained decision tree model $\mathcal{T}$, and a client wants to evaluate $\mathcal{T}$ on his feature vectors without disclosing them to the server or interactions during the evaluation. This necessitates a non-interactive PDTE, which could be achieved using homomorphic encryption.

In the homomorphic evaluation $\mathcal{T}$, a homomorphic comparison in a decision node gives an encrypted bit $\mathsf{Enc}(b) \leftarrow \mathsf{Enc}(\mathbb{1}_{x_i \geq y_j})$. Since the server cannot infer the value of b from $\mathsf{Enc}(b)$, determining which child node (left or right) to evaluate is infeasible unless a costly PIR procedure is incorporated [1]. Otherwise, *both* child nodes of every decision node must be evaluated, resulting in evaluations of all the m decision nodes in $\mathcal{T}$.

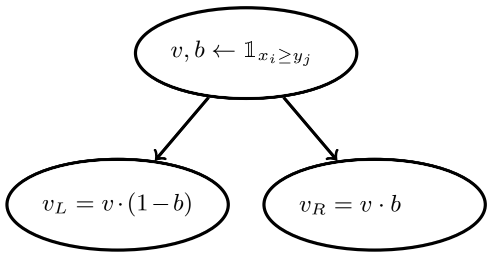
Table 1: List of symbols for a decision tree

Symbol	Meaning
$\mathcal{T}$	Decision tree
d	Depth of decision tree
m	Number of decision nodes
$\mathbf{y} = \{y_1, \ldots, y_m\}$	Thresholds for decision nodes
X	Collection of feature vectors
n	Dimension of a feature vector
s	Bitlength of a feature
$\mathbf{x} = \{x_1, \ldots, x_n\}$	Input feature vector
k	Number of classification values
$\mathbf{v} = \{v_1, \ldots, v_{m+1}\}$	Classification values associated with leaf nodes

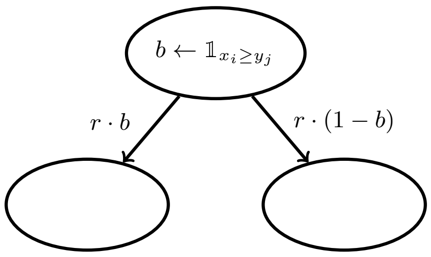
Tree traversal is a data-oblivious procedure to aggregate evaluation results of these m decision nodes. In previous works, SortingHat employs Path Conjugation for tree traversal, which is also used in [30]. On the other hand, Level Up [23] employs another SumPath method, which is also used in [18,28,29].

In Path Conjugation, every decision node is associated with two values: a node value v and a control bit b comparing some feature value to a threshold value. The node value is determined by the node value and the control bit of the previous decision node, as illustrated in Figure 1a. As such, the leaf node is also associated with a node value, which equals one for the desired output leaf and zero otherwise.

In SumPath, every edge is assigned an *edge cost* determined by the control bit of the previous decision node, as illustrated in Figure 1b. Since each leaf node is connected to the root in a unique path, summing up the edge costs along this path yields the *path cost* of a leaf node. As such, only the path cost of a desired output leaf equals to zero, and for all other leaves path costs are non-zero values.



(a) Path Conjugation, where v denotes the value stored in a node



(b) SumPath, where r is a random number or 1

Fig. 1: Two oblivious tree traversal methods

2.5 Oblivious Binary Codes Comparison

Binary encoding for an integer $x \in [0, L-1]$ is generally classified into two categories: binary representation $BR(x)$ of length $\log_2 L$, or a constant-weight encoding $CW_{h,\ell}(x)$ of weight h and bit length ℓ. In the latter category, the bit length ℓ is determined by the relation $\binom{\ell}{h} \geq L$, which approximates to $\ell \in O(\sqrt[h]{h!L} + h)$ [22]. Notably, $CW_{1,L}(x)$ yields the one-hot encoding of x.

Constant-Weight Equality Operator Typically, the bitlength ℓ in constant-weight codes is higher than $\log_2 L$ in the binary representation. However, constant-weight codes support oblivious equality checks of a low multiplicative depth [22]. Precisely, the equality check for $\mathbf{a} = CW_{h,\ell}(a)$ and $\mathbf{b} = CW_{h,\ell}(b)$ can be achieved by evaluating

$$h' \coloneqq \sum_{i=1}^{\ell} \mathbf{a}[i] \cdot \mathbf{b}[i]$$
$$\mathsf{EQ}(a,b) = \frac{1}{h!} \prod_{i=0}^{h-1} (h' - i), \tag{1}$$

where the multiplicative depth is $1 + \lceil \log_2 h \rceil$ and the number of multiplications is $\ell + h - 1$.

Range Cover Comparison (RCC) Operator This constant-weight equality operator can furthermore be combined with a range cover representation [25,17] to obtain a low-depth comparator, as proposed by Mahdavi et al. in Level Up [23]. Precisely, given $a, b \in [0, 2^s - 1]$, computing

$$\mathsf{GT}(a,b) = \begin{cases} 1, & \text{if } a > b \\ 0, & \text{otherwise} \end{cases}$$

is equivalent to checking whether the point a lies in the range $[b+1, 2^s - 1]$, i.e.

$$\mathsf{GT}(a,b) = \mathbb{1}_{a \in [b+1, 2^s-1]}.$$

This leads to the following definition of an *interval tree* where points and ranges can be efficiently represented, as visualized in Figure 2.

Definition 1 (Adapted from [25,23]). *Let T be a binary interval tree whose leaf nodes contain elements in $[0, 2^s - 1]$. A* range cover *$RC(b+1, 2^s-1)$ contains the set of nodes in T such that (1) it contains at most one node in each level (2) its set of children at the leaf level is exactly $[b+1, 2^s-1]$. A* point encoding *$PE(a)$ contains the set of nodes from leaf a to the root (except the root itself).*

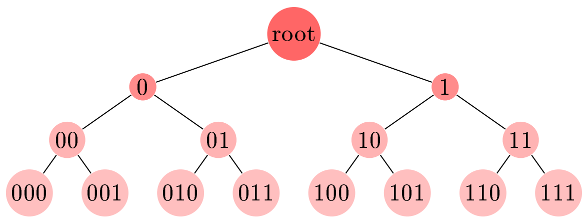


Fig. 2: A binary interval tree containing $[0, 7]$. For example, the point encoding of the number 5 is $PE(5) = \{1, 10, 101\}$ and the range cover of $[1, 7]$ is $RC(1, 7) = \{1, 01, 001\}$.

As observed in [25], if $a \notin [b+1, 2^s-1]$, then $RC(b+1, 2^s-1) \cap PE(a) = \emptyset$; otherwise, they will intersect at one and only one node. As $RC(b+1, 2^s-1)$

contains at most s elements (one node at each level), this comparison contains at most s equality checks of i bits for $i = 1, 2, \ldots, s$, i.e.

$$\mathsf{GT}(a, b) = \sum_{i=1}^{s} \mathsf{EQ}\left(RC(b+1, 2^s - 1)[i], PE(a)[i]\right), \tag{2}$$

assuming $RC(b+1, 2^s-1)[i]$ has i digits. In Level Up [23], the s numbers in range cover are encoded using $CW_{h,\ell}(\cdot)$ where the weight h is small (such as 2 or 4), and the ℓ is the lowest number satisfying $\binom{\ell}{h} \geq 2^s$. Then their equality checks are performed using Equation (1). As such, this comparator contains $s \cdot (\ell + h - 1)$ multiplications in multiplicative depth $1 + \lceil \log_2 h \rceil$.

Folklore Bit-Wise Comparator The folklore comparator compares the binary representations of two numbers *bit-by-bit* [15,22,23]. Precisely, bit-wise comparisons can be achieved with degree-2 polynomials, i.e. for $a, b \in \{0, 1\}$,

$$\theta_{EQ}(a, b) = 1 - (a - b)^2$$
$$\theta_{GT}(a, b) = (1 - a) \cdot b.$$

Then using recursion, Algorithm 1 compares two numbers of bit length s using $2s - 1$ multiplications, and the lowest multiplicative depth to realize this algorithm is $(1 + \log s)$.

Algorithm 1 Folklore bit-wise comparator

```
Input: $\mathbf{a} = BR(a)$, $\mathbf{b} = BR(b) \in \{0,1\}^s$
Output: $\mathsf{GT}(a, b)$
1: function BitwiseComp($\mathbf{a}, \mathbf{b}$)
2:     if $s = 1$ then
3:         return $\theta_{GT}(\mathbf{a}[1], \mathbf{b}[1])$
4:     else
5:         return $\theta_{GT}(\mathbf{a}[1], \mathbf{b}[1]) + \theta_{EQ}(\mathbf{a}[1], \mathbf{b}[1]) \cdot$ BitwiseComp($\mathbf{a}[2, s], \mathbf{b}[2, s]$)
6:     end if
7: end function
```

3 Batched Ciphertext-Plaintext Comparisons

In the batched PDTE, a client encrypts N feature vectors $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}\}$ and queries about the inference results for each of them using the decision tree $\mathcal{T}$ with thresholds $\mathbf{y}$. In the SIMD evaluation of a decision node, the server homomorphically compares features $\{x_i^{(1)} \in \mathbf{x}^{(1)}, x_i^{(2)} \in \mathbf{x}^{(2)}, \ldots, x_i^{(N)} \in \mathbf{x}^{(N)}\}$ to a threshold value $y_i \in \mathbf{y}$. Since threshold values are stored in the server in plaintexts, this amounts to performing a batched ciphertext-plaintext comparison.

This section proposes two methods for batched ciphertext-plaintext comparisons with improved performance.

3.1 Batched ciphertext-plaintext RCC comparator

The RCC comparator for two numbers of s bits, as described in (2), contains at most s equality checks whose operands are of i bits for $i = 1, 2 \ldots, s$. In Level Up, these equality checks are performed using the constant-weight operator in Equation (1). This allows comparing batched encrypted values to various plaintext values.

In our batched ciphertext-plaintext comparison using RCC, batched encrypted values are compared to the same plaintext value. For this scenario, we follow the procedure above and optimize a subcomponent, the constant-weight equality operator in Equation (1). This further leads to a distinct ciphertext packing method from Level Up, which improves the amortized communication and storage.

Ciphertext-Plaintext Constant-Weight Equality Operator Given $\mathbf{a} = CW_{h,\ell}(a)$ and $\mathbf{b} = CW_{h,\ell}(b)$, the equality operator in Equation (1) is data-oblivious to both a and b, demonstrating its suitability for ciphertext-ciphertext comparisons.

In the ciphertext-plaintext scenario, the equality check only needs to be data-oblivious to a. Therefore, Equation (1) can be further simplified into

$$\mathsf{EQ}(a, b) = \prod_{\mathbf{b}[i]=1} \mathbf{a}[i], \tag{3}$$

and its homomorphic evaluation requires $(h - 1)$ ciphertext-ciphertext multiplications in depth $\lceil \log_2 h \rceil$ and zero ciphertext-plaintext multiplications.

Our Ciphertext Packing Although the ciphertext packing method in Level Up naturally supports our batched ciphertext-plaintext RCC comparator, its storage and communication cost could be further improved, as pointed out in the Future Work section of [23]. In line with this, we introduce another ciphertext packing method, as depicted in Figure 3.

Precisely, let N denote the number of SIMD slots for given BFV parameters, our method allows to pack N values for one feature $\{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}$ into BFV ciphertexts. Subsequently, these s-bit values are compared to a plaintext threshold value y.

As explained in Section 2.5, comparing two values is equivalent to checking the intersection between the point encoding of one element and the range cover of the other. In our method, point encodings of features are encrypted and packed, and the range cover of the threshold y is in plaintext.

For each feature $x^{(i)}$, its point encoding $PE\left(x^{(i)}\right)$ is a length-s vector and the component $x^{(i)}_{(j)} = PE\left(x^{(i)}\right)[j]$ contains j bits where $j = 1, \ldots, s$. Each $x^{(i)}_{(j)}$ is further encoded using constant weight h_j into $CW_{h_j,\ell_j}\left(x^{(i)}_{(j)}\right)$ of length ℓ_j. Since the bit-length of $x^{(i)}_{(j)}$ is independent of i and decreases as j decreases, the Hamming weight for encoding is also independent of i and $h_s = \max(h_j)$. The bit length ℓ_j is determined by the relation $\binom{\ell_j}{h_j} \geq 2^j$, which approximates to $\ell_j \in O(\sqrt[h_j]{h_j!2^j} + h_j)$, as explained in Section 2.5. For simplicity, the ℓ_j bits in $CW_{h_j,\ell_j}\left(x^{(i)}_{(j)}\right)$ are denoted as $x^{(i)}_{(j,k)} = CW_{h_j,\ell_j}\left(x^{(i)}_{(j)}\right)[k]$ where $k = 1, \ldots, \ell_j$.

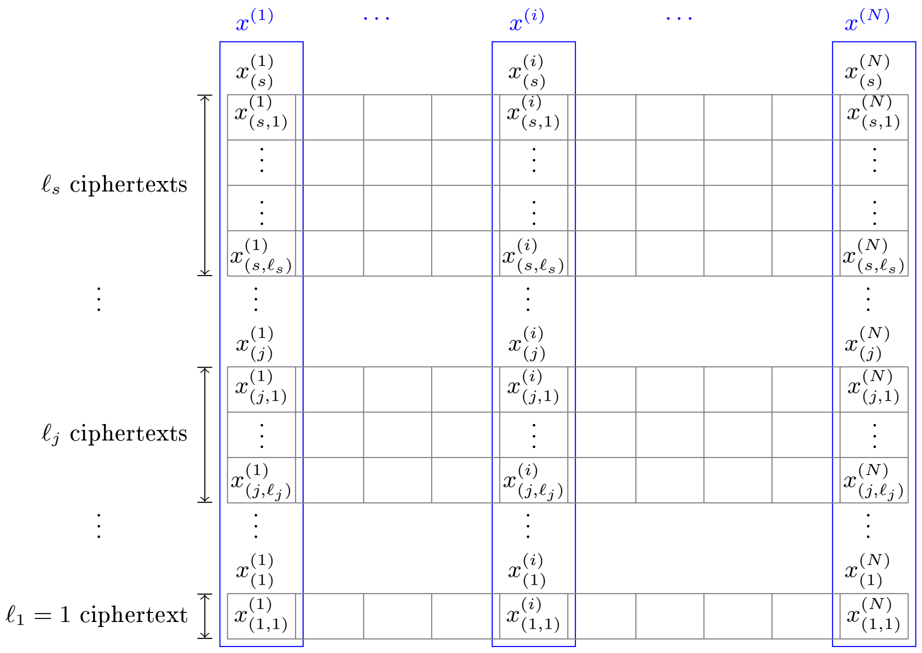


Fig. 3: Our method of packing N values for one feature $\{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}$ of s bits into BFV ciphertexts, which will be compared to one plaintext threshold value y using our batched RCC comparator.

In practice, Hamming weight h_s are small numbers. For example, two common choices of h_s in the Level Up implementation are 2 and 4. Therefore, we choose $h_s = h_{s-1} = \cdots = h_{j'}$ for some small j', and the Hamming weight h_j steadily decreases with decreasing j until $h_1 = 1$. Therefore, the length $\ell_j \in O(\sqrt[h_j]{h_j!2^j} + h_j)$ decreases exponentially with j. As such, the amortized storage for our ciphertext packing is

$$\frac{\ell_s + \ell_{s-1} + \ldots + \ell_1}{N} \ll \frac{s \cdot \ell_s}{N},$$

and the right-hand side (RHS) corresponds to the amortized storage for Level Up ciphertext packing.

Homomorphic Evaluation of our Batched RCC Comparator On the other hand, the range cover $RC(y\_range)$ determined by y contains maximum s

numbers, each with bit precision ranging from 1 to s. Section 2.5 details y_range for the GT comparator, and for GE, LT and LE comparators, the y_range can be constructed similarly. Denote $RC(y\_range)[j]$ of j bits as y_j, which are encoded using constant weight h_j into $CW_{h_j,\ell_j}\left(y_{(j)}\right)$ with binary components $y_{(j,k)} = CW_{h_j,\ell_j}\left(y_{(j)}\right)[k]$ where $k = 1, \ldots, \ell_s$.

As such, our ciphertext-plaintext constant-weight equality operation gives

$$\mathsf{EQ}(x^{(i)}_{(j)}, y_{(j)}) = \prod_{y_{(j,k)}=1} x^{(i)}_{(j,k)}, \tag{4}$$

which contains $h_j - 1$ ciphertext-ciphertext multiplications in depth $\log_2 h_j$. Similar to Equation (2), the comparison result can be obtained from

$$\mathsf{COMP}(x^{(i)}, y) = \sum_{j=1}^{s} EQ(x^{(i)}_{(j)}, y_{(j)}) \tag{5}$$

where COMP is predetermined choice of GT,GE, LT or LE.

Overall, our batched ciphertext-plaintext RCC comparator requires

$$\frac{\sum_{j=1}^{s}(h_j - 1)}{N} < \frac{s \cdot (h_s - 1)}{N}$$

ciphertext-ciphertext multiplications at depth $\log_2 h_s$ and zero ciphertext-plaintext multiplications. The RHS corresponds to the number of ciphertext-ciphertext multiplications of the RCC comparator in $\mathsf{Level\ Up}$, which also requires $\frac{s \cdot \ell_s}{N}$ ciphertext-plaintext multiplications.

3.2 Batched Ciphertext-Plaintext Constant-Weight Piece-Wise Comparator

Inspired by this bit-by-bit comparison in Algorithm 1, we propose a *piece-by-piece* comparator for constant-weight codes, which is only oblivious to one operand and is therefore suitable for ciphertext-plaintext comparisons.

Let $\mathbf{a} = CW_{h,\ell}(a)$ and $\mathbf{b} = CW_{h,\ell}(b)$, and suppose encryptions $\{\mathsf{Enc}(\mathbf{a}[i]), 1 < i \leq \ell\}$ and the plaintext $\mathbf{b}$ are given. The first piece in $\mathbf{a}$ is from its most significant bit (inclusive) to the position of the first one in $\mathbf{b}$ (exclusive).

If there is any number one in this first piece, then $\mathsf{GT}(a, b) = 1$. This condition is checked by summing all elements in this piece to obtain a number $x \in \{0, 1, \ldots, h\}$. Then evaluating the function $\theta_{GTZero}(x, h) = 1 - \frac{1}{h!}\prod_{i=1}^{h}(i - x)$ returns one if $x \in \{1, \ldots, h\}$ and zero if $x = 0$.

Otherwise, if the first one in $\mathbf{a}$ has the same position as $\mathbf{b}$, we compare the code in lower digits piece-by-piece recursively. The complete algorithm is presented in Algorithm 2, and the minimum multiplicative depth to realize it is $\lceil \log_2 ((h + 1) + (h - 1 + 1) + \ldots + (2 + 1) + 1) \rceil = \left\lceil \log_2 \left(\frac{(h+4)(h-1)}{2} + 1 \right) \right\rceil$.

The ciphertext packing strategy for the constant-weight piece-wise comparator is presented in Figure 4. Compared to the ciphertext packing for the RCC comparator in Figure 3, no point encoding is needed, hence the amortized storage $\frac{\ell}{N}$ is also lower for comparable choices of Hamming weight h_s and h.

Algorithm 2 Constant-weight piece-wise comparator

```
Input: $\mathbf{a} = CW_{h,\ell}(a), \mathbf{b} = CW_{h,\ell}(b) \in \{0,1\}^\ell$
Output: $\mathsf{GT}(a,b)$
1: function PiecewiseComp($\mathbf{a}, \mathbf{b}, h$)
2:     $\mathbf{c} \leftarrow [i \mid \mathbf{b}[i] = 1]$        ▷ $\mathbf{c}$ is an ordered array of size $h$
3:     if $h = 1$ then
4:         return $\sum_{i=1}^{\mathbf{c}[1]-1} \mathbf{a}[i]$
5:     else
6:         $\alpha \leftarrow \theta_{GTZero}(\sum_{i=1}^{\mathbf{c}[1]-1} \mathbf{a}[i], h)$
7:         return $\alpha + (1-\alpha) \cdot \mathbf{a}\big[\mathbf{c}[1]\big] \cdot \textsc{PiecewiseComp}(\mathbf{a}\big[\mathbf{c}[1]+1, \ell\big], \mathbf{b}\big[\mathbf{c}[1]+1, \ell\big], h-1)$
8:     end if
9: end function
```

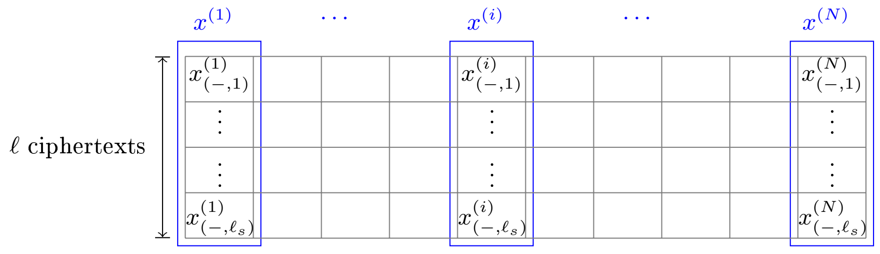


Fig. 4: Our method of packing N values for one feature $\{x^{(1)}, x^{(2)}, \dots, x^{(N)}\}$ of s bits into BFV ciphertexts, which will be compared to one plaintext threshold value y using the constant-weight piece-wise comparator. Each feature $x^{(i)}$ is encoded using constant weight h into $CW_{h,\ell}\left(x^{(i)}\right)$ of length ℓ, and its binary components $CW_{h,\ell}\left(x^{(i)}\right)[k]$ are denoted as $x^{(i)}_{(-,k)}$, with the first subscript indicating no point encoding is applied.

3.3 Benchmarking Batched Ciphertext-Plaintext Comparisons

For the experiment, we assume a client sends ciphertexts corresponding to N values of s bits each, which will be compared to a plaintext value in the server. After the homomorphic evaluation, the client receives a ciphertext whose SIMD slots encode the N comparison results.

We consider four methods for such batched ciphertext-plaintext comparisons: 1) the RCC operator in [23] with the same plaintext values in all slots, 2) our batched RCC in Section 3.1, 3) the folklore bit-wise comparator with the same plaintext values in all slots and 4) our constant-weight piece-wise comparator in Section 3.2.

Table 2 presents their performance for input bitlength 8 and 16. Specifically, the performance of 1) and 3) are obtained from running the Level Up implementation[4], and 2) and 4) are implemented using Microsoft SEAL [24].

In summary, our methods 2) and 4) provide computation time ranges from 4.8× to over 72× faster than prior methods 1) and 3) while maintaining comparable communication costs and multiplicative depth.

Table 2: Performance of different batched ciphertext-plaintext comparators in BFV with $N = 2^{14}$ and $t = 65537$. The multiplication depth refers to the depth of ciphertext-ciphertext multiplications. Non-applicable parameters are denoted as $\perp$.

		Amortized Computational Time		Amortized Client-to-server Communication Cost		Multiplicative Depth	
		$s = 8$	$s = 16$	$s = 8$	$s = 16$	$s = 8$	$s = 16$
RCC [23]	$h = 2$	245 μs	8340 μs	45 kb	1342 kb	1	1
	$h = 4$	188 μs	1526 μs	20 kb	136 kb	2	2
	$h = 8$	$\perp$	1308 μs	$\perp$	70 kb	3	3
Our batched RCC	$h_s = 2$	19 μs	41 μs	11 kb	180 kb	1	1
	$h_s = 4$	39 μs	82 μs	8 kb	38 kb	2	2
Folklore bit-wise [23]	$\perp$	457 μs	1982 μs	1 kb	3 kb	3	4
Constant-weight piece-wise	$h = 2$	10 μs	18 μs	3 kb	52 kb	2	2
	$h = 4$	37 μs	39 μs	1 kb	5 kb	4	4

4 Tree Traversal Methods

From homomorphic evaluations of decision nodes and tree traversal, the server obtains an encrypted value $\mathsf{Enc}(r_j)$ for each leaf j, where $1 \leq j \leq m + 1$. This

[4] https://github.com/RasoulAM/private-decision-tree-evaluation

value r_j indicates whether the leaf j is the output leaf, and we denote the array of r_j as $\mathbf{r}$.

In Path Conjugation, the result vector $\mathbf{r}_c$ is a unit vector whose inner product with $\mathbf{v}$ yields the predicted classification. The encrypted classification value is sent to the client. However, this unit vector in Path Conjugation comes with a price: it requires an expensive RLWEtoRGSW conversion [11] procedure in TFHE. Let w denote the multiplicative depth of a ciphertext-plaintext comparison algorithm in BFV, instantiating Path Conjugation in BFV leads to a high multiplicative depth $\mathcal{O}(d \cdot w)$.

On the other hand, SumPath returns the encryption of $\mathbf{r}_s$ to the client, whose value is zero for the output leaf and non-zero otherwise. The client decrypts, obtains the index of the output leaf, and looks up its corresponding classification value. Its instantiation in both TFHE and BFV is fast and straightforward, and its low multiplicative depth $\mathcal{O}(w)$ enables PDTE using practical BFV parameters.

However, compared to Path Conjugation, the server-to-client communication in SumPath is $\mathcal{O}(m)$ larger. Besides, integrating decision trees into a tree ensemble [7,16] is a widely used technique to improve prediction accuracy. Since SumPath requires the client to look up the classification value for every decision tree, its applicability for private evaluations of tree ensembles is strongly limited.

Then a natural question is whether there is a tree traversal method that not only achieves low multiplicative depth but also yields a unit result vector with reasonable computation costs. This leads to our adapted SumPath method.

4.1 Our Adapted SumPath Method

The edge cost computation in SumPath is visualized in Figure 1b. Our adaption of SumPath follows from this observation: when the parameter r in Figure 1b is set to be 1 for all decision nodes, the path cost of every leaf counts the number of unsatisfied conditions in the path from the root to that leaf. As such, the path cost of the desired leaf equals zero, and the path costs of all the other leaves are in $\{1, \dots, d-1\}$.

Since the function

$$\theta_{EQZero}(x, d) = \frac{1}{(d-1)!} \prod_{i=1}^{d-1} (i - x)$$

maps zero to one and any elements in $\{1, \dots, d-1\}$ to zero[5], evaluating $\theta_{EQZero}(\cdot, d)$ on the path cost of each leaf maps the result vector $\mathbf{r}_s$ in SumPath into the desired unit vector denoted as $\mathbf{r}_{as}$.

As such, using our adapted SumPath for tree traversal leads to multiplicative depth $\mathcal{O}(w + \log_2 d)$ for PDTE, where w is the multiplicative depth for one homomorphic comparison in BFV.

[5] This function is also used in the concurrent work [26] to integrate decision trees into random forests.

Optimization: Tree Truncation Since the server knows the classification values in leaves $\mathbf{v}$ in plaintext, the procedure above can be optimized. Precisely, in the inner product $\mathbf{r}_{as} \cdot \mathbf{v}$, the components in $\mathbf{r}_{as}$ that correspond to zero labels do not contribute. Therefore, these leaves can be *truncated* from the decision tree, obviating the need to compute their path costs and evaluations of $\theta_{EQZero}(\cdot, d)$. The visualization of the tree truncation technique is included in Appendix A.

By renaming the most abundant label to zero, at least $\frac{1}{k}$ leaves have zero classification values and can be truncated. Moreover, badly trained models may contain decision nodes whose children leaves both have zero classification values. These nodes can also be truncated without impacting the final output.

5 Batched Private Decision Tree Evaluation

5.1 Security Model

Our work considers the client/server scenario, where a cloud server holds a pre-trained decision tree model $\mathcal{T}$ and a client holds multiple input feature vectors $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}\}$ and wants to know the inference result with $\mathcal{T}$ for each of them. The goal is to protect input privacy such that the server would not be able to learn the clients' input values. Guaranteeing model privacy is out of the scope of this paper: a client with enough resources may be able to reverse-engineer the server-side cloud model after a given number of queries. Moreover, the protocol should be non-interactive to allow full outsourcing computations to the server.

Our threat model is similar to prior works, where the server is an honest-but-curious adversary. This implies that the server always follows the protocol strictly but may attempt to deduce information from the client's inputs.

5.2 Protocol

For setup, the server performs a tree truncation to $\mathcal{T}$ to get $Trun(\mathcal{T})$ and receives the necessary keys (e.g. relinearization keys) from the client. Under the standard circular security assumption, these keys do not leak information about the client's secret key. Our batched PDTE protocol is as follows.

1. The client sends encryptions of N input feature vectors to the server.
2. For j-th decision node where $1 \leq j \leq m$, the server homomorphically compares encryptions of N feature values and the plaintext threshold y_j in an SIMD manner. Section 3 provides two methods for such comparisons. The output $\mathsf{Enc}(\mathbf{b}_j)$ is a ciphertext encoding N binary numbers in its SIMD slots.
3. The server performs adapted $\mathsf{SumPath}$ to $\{\mathsf{Enc}(\mathbf{b}_j)\}_{j=1,\ldots,m}$ in $\mathcal{T}$, whose homomorphic inner product with $\mathbf{v}$ gives a ciphertext. The SIMD slots of this ciphertext are N classification values
4. The client decrypts this ciphertext to obtain these N classification values, one for each feature vector.

Security of Batched PDTE Clients' feature vectors, comparison results of decision nodes, and classification labels are all encrypted using BGV schemes with 128-bit security parameters. Its semantic security (IND-CPA) ensures the server (honest but curious) cannot infer corresponding plaintexts, preserving the client's privacy.

5.3 Implementation and Performance

We implement two versions of our batched PDTE protocol using different comparators: BPDTE_RCC using our batched RCC comparators in Section 3.1, and BPDTE_CW using constant-weight piece-wise comparators in Section 3.2. These protocols are evaluated on UCI datasets [13] and compared with the state-of-art prior works [11,23].

Experimental Details We use the same UCI datasets as in prior works: Breast, Heart, Spam and Steel. Furthermore, we apply a tree truncation procedure to reduce server computation without influencing the output. Table 3 presents the key properties of these datasets. Our implementation uses the Microsoft SEAL

Table 3: Characteristics of UCI datasets used in our evaluation, where # **Decision Nodes/Leaves (before|after)** gives the number of decision nodes/leaves in each model before and after tree truncation if that number changes.

	# Features n	**Depth d**	**# Decision Nodes m**	**# Leaves**
Breast	30	7	15	16|8
Heart	13	3	4	5|3
Spam	57	11	108|107	109|52
Steel	33	5	5	6|1

library (v4.1.1) [24], which supports BFV in the SIMD manner and it is also used by Level Up. For SortingHat and Level Up, we use the implementation provided by the authors. Experiments are conducted on a desktop with an Intel Core i7-13700 CPU and 32GB of RAM using a single thread.

Results and Discussion We compare our batched PDTE protocol with prior works in terms of amortized server computation time including comparisons and tree traversals, and amortized query size, *i.e.*, the client-to-server communications. The amortized server-to-client communication is lower than $1kb$ for all protocols and therefore not listed.

Table 4 and Table 5 compare the amortized performance of different PDTE protocols with batch size 16384 for input feature bit-length $s = 11$ and $s = 16$, respectively. This corresponds to the scenario where the client sends encryptions of 16384 feature vectors $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(16384)}\}$ and wants to know the classification output for each of them. The large batch size is useful in practice, for example, when a bank outsources a credit-scoring decision tree and needs to evaluate numerous applicants securely. For completeness, we also compare PDTE protocols with different batch sizes in Appendix B. Since the maximum bit-length supported by SortingHat is 11, SortingHat is not listed in Table 5. Moreover, for $s = 11$, BPDTE_CW outperforms BPDTE_RCC in both communication and computation, hence BPDTE_RCC is not listed in Table 4.

As for BFV parameters, BPDTE_CW and BPDTE_RCC use larger parameters than Level Up to provide higher depth. Precisely, Level Up uses SumPath, where the amortized response of the server is $\mathcal{O}(m)$ and the client needs to look up classification values in a table. On the other hand, BPDTE_CW and BPDTE_RCC use the Adapted SumPath method, where the amortized response of the server is $\mathcal{O}(1)$ at the cost of $\mathcal{O}(\log_2 d)$ multiplicative depth.

As a remark, it is possible to combine our batched ciphertext-plaintext comparators with SumPath for PDTE, which requires the same BFV parameters as in Level Up and therefore attains better communication and computational performance. However, with this $\mathcal{O}(m)$ response, the client needs to perform a table lookup to obtain classification values and the extension to tree ensembles is restricted.

In summary, with batch size 16384, SortingHat is about 10^3 slower than those supporting SIMD operations. Compared to Level Up, BPDTE_RCC and BPDTE_CW are 1.5× to 17× faster overall and have comparable query sizes. For large precision (e.g. $s = 16$), BPDTE_RCC provides slightly lower query sizes than BPDTE_CW (e.g. 0.73×) at the expense of slightly higher computation costs (e.g. $1.4 - 2\times$).

Table 4: Amortized performance of different PDTE protocols with batch size 16384 and input feature bit-length $s = 11$, where SortingHat uses TFHE with $N = 2^{11}$, Level Up uses BFV with $N = 2^{13}$ and BPDTE_CW with $h = 2$ uses BFV with $N = 2^{14}$

	SortingHat ($s = 11$)			**Level Up** ($s = 11, h = 4$)			**BPDTE_CW** ($s = 11, h = 2$)		
	Comparison	Traversal	Query Size	Comparison	Traversal	Query Size	Comparison	Traversal	Query Size
Breast	7 ms	178 ms	960 kb	139 μs	117 μs	310 kb	9 μs	139 μs	90 kb
	Total: 185 ms			Total: 256 μs			Total: 148 μs		
Heart	3 ms	47 ms	416 kb	156 μs	25 μs	135 kb	3 μs	18 μs	117 kb
	Total: 50 ms			Total: 181 μs			Total: 21 μs		
Spam	69 ms	1283 ms	1824 kb	378 μs	1089 μs	589 kb	78 μs	1326 μs	513 kb
	Total: 1352 ms			Total: 1467 μs			Total: 1404 μs		
Steel	3 ms	59 ms	1056 kb	125 μs	34 μs	341 kb	4 μs	12 μs	297 kb
	Total: 62 ms			Total: 159 μs			Total: 16 μs		

Table 5: Amortized performance of different PDTE protocols with batch size 16384 and input feature bit-length $s = 16$, where Level Up uses BFV with $N = 2^{13}$, BPDTE_RCC with $h_s = 4$ and and BPDTE_CW with $h = 2$ both use BFV with $N = 2^{14}$

	Level Up ($s = 16, h = 4$)			**BPDTE_RCC** ($s = 16, h_s = 4$)			**BPDTE_CW** ($s = 16, h = 2$)		
	Comparison	**Traversal**	**Query Size**	**Comparison**	**Traversal**	**Query Size**	**Comparison**	**Traversal**	**Query Size**
Breast	583 μs	159 μs	968 kb	75 μs	139 μs	1140 kb	17 μs	138 μs	1560 kb
	Total: 742 μs			Total: 214 μs			Total: 155 μs		
Heart	309 μs	34 μs	420 kb	20 μs	18 μs	494 kb	4 μs	18 μs	676 kb
	Total: 343 μs			Total: 38 μs			Total: 22 μs		
Spam	1857 μs	1595 μs	1839 kb	536 μs	1501 μs	2166 kb	118 μs	1489 μs	2964 kb
	Total: 3452 μs			Total: 2037 μs			Total: 1607 μs		
Steel	262 μs	46 μs	1065 kb	25 μs	12 μs	1254 kb	6 μs	12 μs	1716 kb
	Total: 308 μs			Total: 37 μs			Total: 18 μs		

6 Conclusion

In this work, we proposed two batched ciphertext-plaintext comparisons, our batched RCC comparator and the constant-weight piece-wise comparator. Compared to directly applying previous methods to this scenario, our evaluation of these comparison operators shows a speedup of up to 72× for 16-bit numbers while maintaining comparable communication costs and multiplicative depth.

These batched ciphertext-plaintext comparisons, together with our adapted SumPath tree traversal method, lead to two non-interactive PDTE protocols, BPDTE_RCC and BPDTE_CW. Compared to the prior state-of-art [23], these protocols not only avoid the client looking up classification values in a table but also demonstrate an enhanced performance of up to 17× in batch size 16384.

Acknowledgments. This work is partially supported by the Research Council KU Leuven under the grant C24/18/049, CyberSecurity Research Flanders with reference number VR20192203. The authors would also like to thank Dr. Svetla Nikova for the valuable discussions and Prof. Frederik Vercauteren for helpful feedback on this paper.

References

1. Azogagh, S., Delfour, V., Gambs, S., Killijian, M.: PROBONITE: private one-branch-only non-interactive decision tree evaluation. In: Brenner, M., Costache, A., Rohloff, K. (eds.) Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Los Angeles, CA, USA, 7 November 2022. pp. 23–33. ACM (2022). https://doi.org/10.1145/3560827.3563377
2. Bai, J., Song, X., Cui, S., Chang, E.C., Russello, G.: Scalable private decision tree evaluation with sublinear communication. In: Suga, Y., Sakurai, K., Ding, X., Sako, K. (eds.) ASIACCS 22. pp. 843–857. ACM Press (May / Jun 2022). https://doi.org/10.1145/3488932.3517413
3. Bonte, C., Iliashenko, I., Park, J., Pereira, H.V.L., Smart, N.P.: FINAL: Faster FHE instantiated with NTRU and LWE. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part II. LNCS, vol. 13792, pp. 188–215. Springer, Cham (Dec 2022). https://doi.org/10.1007/978-3-031-22966-4_7
4. Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine learning classification over encrypted data. In: NDSS 2015. The Internet Society (Feb 2015). https://doi.org/10.14722/ndss.2015.23241
5. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Berlin, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5_50
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012. pp. 309–325. ACM (Jan 2012). https://doi.org/10.1145/2090236.2090262
7. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324
8. Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: Ning, P., De Capitani di Vimercati, S., Syverson, P.F. (eds.) ACM CCS 2007. pp. 498–507. ACM Press (Oct 2007). https://doi.org/10.1145/1315245.1315307
9. Chern, C., Lei, W., Huang, K., Chen, S.: A decision tree classifier for credit assessment problems in big data environments. Inf. Syst. E Bus. Manag. **19**(1), 363–386 (2021). https://doi.org/10.1007/S10257-021-00511-W
10. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 3–33. Springer, Berlin, Heidelberg (Dec 2016). https://doi.org/10.1007/978-3-662-53887-6_1
11. Cong, K., Das, D., Park, J., Pereira, H.V.L.: SortingHat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 563–577. ACM Press (Nov 2022). https://doi.org/10.1145/3548606.3560702
12. Cong, K., Geelen, R., Kang, J., Park, J.: Revisiting oblivious top-k selection with applications to secure k-nn classification. Cryptology ePrint Archive, Report 2023/852 (2023), https://eprint.iacr.org/2023/852
13. Dua, D., Graff, C.: UCI Machine Learning Repository (2017), http://archive.ics.uci.edu/ml
14. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), https://eprint.iacr.org/2012/144

15. Hao, Y., Qin, B., Sun, Y.: Privacy-preserving decision-tree evaluation with low complexity for communication. Sensors **23**(5), 2624 (2023). https://doi.org/10.3390/S23052624
16. Hastie, T., Tibshirani, R., Friedman, J.H.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition. Springer Series in Statistics, Springer (2009). https://doi.org/10.1007/978-0-387-84858-7
17. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 669–684. ACM Press (Nov 2013). https://doi.org/10.1145/2508859.2516668
18. Kiss, Á., Naderpour, M., Liu, J., Asokan, N., Schneider, T.: SoK: Modular and efficient private decision tree evaluation. PoPETs **2019**(2), 187–208 (Apr 2019). https://doi.org/10.2478/popets-2019-0026
19. Liu, W., Fan, H., Xia, M.: Credit scoring based on tree-enhanced gradient boosting decision trees. Expert Syst. Appl. **189**, 116034 (2022). https://doi.org/10.1016/J.ESWA.2021.116034
20. Lu, W., Huang, Z., Zhang, Q., Wang, Y., Hong, C.: Squirrel: A scalable secure two-party computation framework for training gradient boosting decision tree. In: Calandrino, J.A., Troncoso, C. (eds.) 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023. pp. 6435–6451. USENIX Association (2023), https://www.usenix.org/conference/usenixsecurity23/presentation/lu
21. Lu, W., Zhou, J.J., Sakuma, J.: Non-interactive and output expressive private comparison from homomorphic encryption. In: Kim, J., Ahn, G.J., Kim, S., Kim, Y., López, J., Kim, T. (eds.) ASIACCS 18. pp. 67–74. ACM Press (Apr 2018). https://doi.org/10.1145/3196494.3196503
22. Mahdavi, R.A., Kerschbaum, F.: Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. In: Butler, K.R.B., Thomas, K. (eds.) USENIX Security 2022. pp. 1723–1740. USENIX Association (Aug 2022)
23. Mahdavi, R.A., Ni, H., Linkov, D., Kerschbaum, F.: Level up: Private non-interactive decision tree evaluation using levelled homomorphic encryption. In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) ACM CCS 2023. pp. 2945–2958. ACM Press (Nov 2023). https://doi.org/10.1145/3576915.3623095
24. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL (Jan 2023), microsoft Research, Redmond, WA.
25. Shi, E., Bethencourt, J., Chan, H.T.H., Song, D.X., Perrig, A.: Multi-dimensional range query over encrypted data. In: 2007 IEEE Symposium on Security and Privacy. pp. 350–364. IEEE Computer Society Press (May 2007). https://doi.org/10.1109/SP.2007.29
26. Shin, H., Choi, J., Lee, D., Kim, K., Lee, Y.: Fully homomorphic training and inference on binary decision tree and random forest. Cryptology ePrint Archive, Report 2024/529 (2024), https://eprint.iacr.org/2024/529
27. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. DCC **71**(1), 57–81 (2014). https://doi.org/10.1007/s10623-012-9720-4
28. Tai, R.K.H., Ma, J.P.K., Zhao, Y., Chow, S.S.M.: Privacy-preserving decision trees evaluation via linear functions. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017, Part II. LNCS, vol. 10493, pp. 494–512. Springer, Cham (Sep 2017). https://doi.org/10.1007/978-3-319-66399-9_27
29. Tueno, A., Boev, Y., Kerschbaum, F.: Non-interactive private decision tree evaluation. In: Singhal, A., Vaidya, J. (eds.) Data and Applications Security and Privacy

XXXIV - 34th Annual IFIP WG 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25-26, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12122, pp. 174–194. Springer (2020). https://doi.org/10.1007/978-3-030-49669-2_10

30. Tueno, A., Kerschbaum, F., Katzenbeisser, S.: Private evaluation of decision trees using sublinear cost. PoPETs **2019**(1), 266–286 (Jan 2019). https://doi.org/10.2478/popets-2019-0015
31. Wu, D.J., Feng, T., Naehrig, M., Lauter, K.E.: Privately evaluating decision trees and random forests. PoPETs **2016**(4), 335–355 (Oct 2016). https://doi.org/10.1515/popets-2016-0043
32. Zhang, D., Zhou, X., Leung, S.C.H., Zheng, J.: Vertical bagging decision trees model for credit scoring. Expert Syst. Appl. **37**(12), 7838–7843 (2010). https://doi.org/10.1016/J.ESWA.2010.04.054
33. Zheng, W., Deng, R., Chen, W., Popa, R.A., Panda, A., Stoica, I.: Cerebro: A platform for multi-party cryptographic collaborative learning. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021. pp. 2723–2740. USENIX Association (Aug 2021)
34. Zuber, M., Sirdey, R.: Efficient homomorphic evaluation of k-NN classifiers. PoPETs **2021**(2), 111–129 (Apr 2021). https://doi.org/10.2478/popets-2021-0020

A Tree Truncation

For the decision tree in Figure 5, applying the tree truncation gives Figure 6.

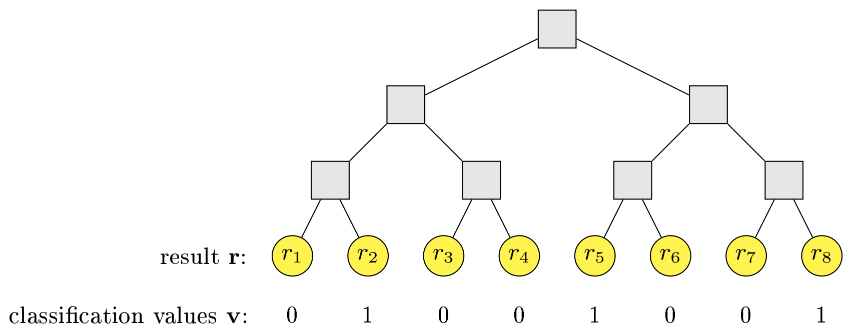

result **r**:
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8
classification values **v**:
0 1 0 0 1 0 0 1

Fig. 5: An example decision tree in depth $d = 3$ with $m = 7$ decision nodes, $m+1 = 8$ leaves and $k = 2$ classification values. In its PDTE, the server obtains an encrypted value $\mathsf{Enc}(r_j)$ for each leaf j, where $1 \le j \le 8$

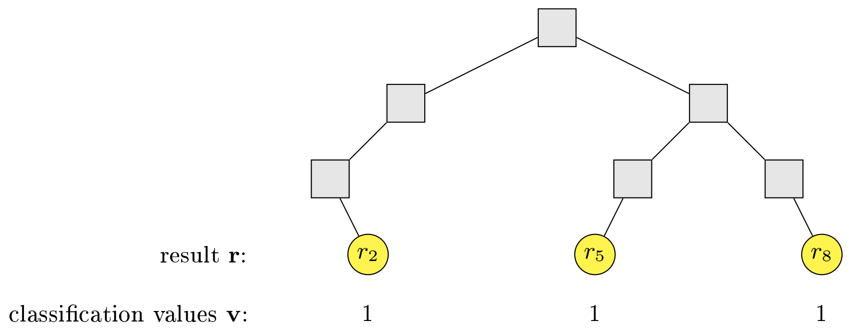


Fig. 6: The truncated decision tree in Figure 5, where the tree contains 6 decision nodes instead of 7 and the result vector contains 3 elements instead of 8.

B Performance Comparison between Different Batch Sizes

In batched PDTE with batch size a, the client sends encryptions of a feature vectors $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(a)}\}$ and wants to know the classification output for each of them. This appendix discusses PDTE running times for a fixed decision tree $\mathcal{T}$ but different a.

For SortingHat with $N = 2^{11}$, Level Up with $N = 2^{13}$ and $h = 4$, BPDTE_CW with $N = 2^{14}$ and $h = 2$ (*i.e.* PDTEs in Table 4), Figure 7 compares their running times for the **Heart** model with 11-bit feature precision. Since SortingHat does not support SIMD packing, the total running time scales linearly with a, assuming their FHE parameters are fixed. In Level Up, components of 712 features are packed in one ciphertext in their implementation, hence the total running time is a step function with step 712. In BPDTE_CW, components of 2^{14} features are packed in one ciphertext, hence the total running time is a step function with step 2^{14}.

As shown in Figure 7, for PDTEs of the **Heart** model, SortingHat is the fastest for batch sizes from 1 to ~ 10, Level Up is the fastest for batch sizes from ~ 10 to ~ 2100, and BPDTE_CW is the fastest for batch sizes larger than ~ 2100. PDTEs of other models attain similar behaviour, but intersection points for the optimal PDTE will differ.

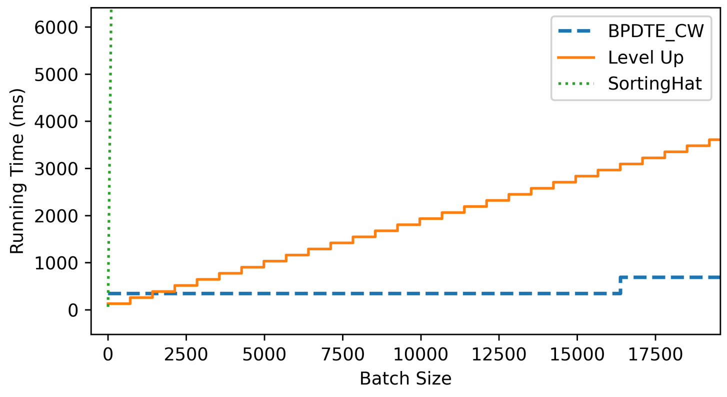


Fig. 7: Computation time (comparison+tree traversal) for the **Heart** model of different PDTE protocols with input feature length $s = 11$ and different batch sizes in x-axis.