algorithms

MDPI

Article

Non-Interactive Decision Trees and Applications with Multi-Bit TFHE

Jestine Paul [1,2,*], Benjamin Hong Meng Tan [1], Bharadwaj Veeravalli [2] and Khin Mi Mi Aung [1,*]

[1] Institute for Infocomm Research, A*STAR, Connexis North, Singapore 138632, Singapore; benjamin_tan@i2r.a-star.edu.sg
[2] Department of Electrical and Computer Engineering, National University of Singapore, Singapore 117583, Singapore; elebv@nus.edu.sg
* Correspondence: jestine_paul@i2r.a-star.edu.sg (J.P.); mi_mi_aung@i2r.a-star.edu.sg (K.M.M.A.)

Abstract: Machine learning classification algorithms, such as decision trees and random forests, are commonly used in many applications. Clients who want to classify their data send them to a server that performs their inference using a trained model. The client must trust the server and provide the data in plaintext. Moreover, if the classification is done at a third-party cloud service, the model owner also needs to trust the cloud service. In this paper, we propose a protocol for privately evaluating decision trees. The protocol uses a novel private comparison function based on fully homomorphic encryption over the torus (TFHE) scheme and a programmable bootstrapping technique. Our comparison function for 32-bit and 64-bit integers is 26% faster than the naive TFHE implementation. The protocol is designed to be non-interactive and is less complex than the existing interactive protocols. Our experiment results show that our technique scales linearly with the depth of the decision tree and efficiently evaluates large decision trees on real datasets. Compared with the state of the art, ours is the only non-interactive protocol to evaluate a decision tree with high precision on encrypted parameters. The final download bandwidth is also 50% lower than the state of the art.

Keywords: (fully) homomorphic encryption; machine learning classification; decision trees

Citation: Paul, J.; Tan, B.H.M.; Veeravalli, B.; Aung, K.M.M. Non-Interactive Decision Trees and Applications with Multi-Bit TFHE. *Algorithms* **2022**, *15*, 333. https://doi.org/10.3390/a15090333

Academic Editor: Frank Werner

Received: 4 August 2022
Accepted: 9 September 2022
Published: 18 September 2022

1. Introduction

Machine learning seeks to develop techniques for building models that can accurately and efficiently predict unseen data based on a given training dataset. It is widely used in the areas of computer vision, natural language processing, and many other fields. There is a growing demand for cloud-based machine learning services, where users can upload query data and get back their predicted result. However, these classifiers sometimes require access to sensitive data, raising concerns about the security and privacy of the query data. For example, a client may have to reveal their entire medical record to the diagnosis server. Any leak of such data can have severe repercussions, and it is crucial to ask whether this prediction service can run without revealing any information. Similarly, the model is the result of dedicated research, with considerable effort and resources expended in its development cycle. It is a valuable asset for an organization; exposing it is not an option. Furthermore, recent work [1] shows that models are susceptible to inversion attacks, which recover information about the training data when given access to the model. Therefore, machine learning as a service protocol is only secure if, after execution, the client only learns the result, and the server learns nothing.

A decision tree is a classifier that is more efficient when the data have a hierarchical structure. It is a binary tree storing a threshold in each branching node and a classification result in the leaf nodes. The classifier compares the thresholds with the feature vector elements and decides which node to traverse. This level-by-level traversal continues until it arrives at some leaf node representing the classification result. A decision tree requires less parameter tuning and training cost than other classifiers, such as neural networks.

In this work, we focus on privacy-preserving classification using a decision tree. The secure evaluation of the decision tree involves the data owner, the model owner and a third-party evaluator, such as a cloud server. The model owner encrypts the model, and the data owner encrypts the feature vector. A third-party evaluator, such as a public cloud service, evaluates the query using the encrypted model parameters and returns the result, which the data owner decrypts, as shown in Figure 1. The main challenge in secure evaluation is to have an efficient way to compare the thresholds with the feature vector securely. We propose a secure comparison algorithm using fully homomorphic encryption (FHE) [2] and a decision tree evaluation protocol that preserves all the functionality without compromising the privacy of the data and model owners.

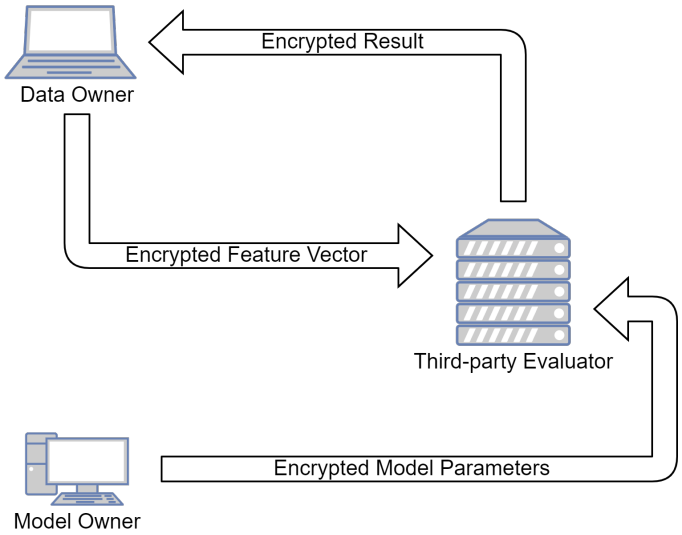


Figure 1. Private machine learning as a service.

This work aims to design and implement a private decision tree evaluation protocol that is both efficient and non-interactive. To this end, we propose a new non-interactive private comparison method that allows its output to be further used in the decision tree evaluation. This method can compare two encrypted integers, a and b, and return a ciphertext that decrypts to 1 if $a > b$ and 0 otherwise. We encrypt each nibble of integer a and b using the fully homomorphic encryption over the torus (TFHE) [3] scheme, and then we compare the encrypted nibbles using programmable bootstrapping [4]. Since the TFHE scheme supports bootstrapping, we can use the encrypted output in further homomorphic operations. This method's non-interactive property is used in the non-interactive private decision tree evaluation protocol. In this protocol, we use our private comparison method to compare the thresholds of the decision tree with the feature vector elements. Then, we use the output of the private comparison method to traverse the decision tree to evaluate the classification result. We implemented and conducted rigorous experiments to study the effect of different bit lengths to assess scalability. We also conducted experiments with decision trees trained with real datasets from the UCI machine learning repository [5] and compared them with the existing protocols.

Existing approaches to the comparison of encrypted integers are not suitable for our purpose. The performance of bit-wise comparison operation using naive TFHE encryption is very slow and grows linearly with bit length. The state-of-the-art comparison operation using XCMP [6] only performs well for integers with a small bit length and grows exponentially with bit length. Even though the comparison operation proposed by Iliashenko et al. [7] can make the comparison for arbitrary length, it is hard to accelerate the comparison operation, as it is not using a 2^n-cyclotomic polynomial. It also does not support the bootstrapping needed for our decision tree evaluation protocol. Therefore, we propose the first multi-bit comparison method using the TFHE scheme to compare integers.

In summary, we make the following improvements to the existing protocols:

- Our private comparison method operates on two encrypted inputs with 128-bit security, facilitating the use of third-party evaluators, such as the public cloud. The performance of our method scales linearly with the input bit length and the depth of the decision tree.
- Our decision tree evaluation protocol is non-interactive, where the client sends encrypted input and receives the encrypted output. There is no other interaction between the client and the server.
- Experimentally evaluated the effect of bit lengths to assess the scalability of our method compared to other methods.
- In our implementation, we conducted experiments with decision trees trained with widely used real datasets from the UCI machine learning repository [5] and compared them with the existing protocols.
- The final output of the protocol consists of a ciphertext that decrypts the classification result. This can be extended to the decision tree returning categorical values, using one-hot encoding. The results from multiple decision trees can be combined to evaluate random forest classification.

The rest of the paper is organized as follows. The next section discusses related work on secure decision tree evaluation and integer comparison. Section 3 introduces the main ideas in fully homomorphic encryption and decision tree evaluation. Using an enhanced comparison protocol, Sections 4 and 5 propose a new method for evaluating decision trees in a privacy-preserving manner. The computational complexity and performance are discussed in Section 6. Finally, Section 7 concludes the paper.

2. Related Work

Earlier works [8,9] focus on securely constructing the decision tree using data mining techniques on joint data without sharing the participant's data. With the advent of cloud computing, privacy-preserving decision tree evaluation has become increasingly important. Generic secure multiparty computation protocols, such as Yao's garbled circuits [10], were proposed for this problem. However, this approach requires one party to create the circuit and the other to evaluate it, resulting in the client participating in the computation. It also incurs high communication costs.

Specialized protocols for the private evaluation of decision trees were subsequently developed that only used generic methods as and when needed. Brickell et al. [11] and Barni et al. [12] proposed a secure protocol for linear branching programs that utilizes both garbled circuits and homomorphic encryption. After that, new interactive protocols were built for decision tree evaluation using a combination of fully homomorphic encryption (FHE), oblivious transfer (OT), and somewhat homomorphic encryption (SHE), requiring multiple rounds of communication. All suffer from high communication and computation overhead. These protocols also do not allow the model to be shared with an evaluator without leaking the model.

Order-preserving encryption (OPE) was proposed by Agrawal et al. [13] and is a type of encryption scheme that preserves the order relationship between plaintexts. Boneh et al. [14] proposed the order-revealing encryption (ORE) scheme, which uses multilinear maps to generalize OPE. Even though these schemes look appealing for integer comparison operation, their order-revealing nature results in the leakage of plaintext from ciphertext using frequency analysis and is vulnerable to inference attacks [15].

Efficient homomorphic encryption-based integer comparison methods, such as the DGK protocol [16–18], compare an encrypted integer with a plaintext integer. Multiple rounds of interaction during protocol execution are required for methods such as that of Bost et al. [19]. The comparison operation from Illashenko et al. [7] does not support bootstrapping and is unsuitable for our purpose. The current state-of-the-art non-interactive integer comparison protocol XCMP [6] has a limitation on the bit length of the inputs.

3. Preliminaries

This section presents some fundamental concepts used in the rest of the paper. In particular, we describe and revisit the TFHE encryption scheme and decision tree structure before applying it in privacy-preserving decision tree evaluation. Further, we discuss a combinational digital circuit that comprises comparators and multiplexers.

3.1. Fully Homomorphic Encryption

Homomorphic encryption, unlike traditional encryption, allows the user to operate on ciphertexts securely. A fresh homomorphic encryption ciphertext is a summation of the plaintext $\in \mathbb{Z}_p/\langle x^n+1\rangle$, a mask $\in \mathbb{Z}_q/\langle x^n+1\rangle, q > p$ which is removable using the secret key, and an initial noise which can be removed using the modulus p operations. The initial noise is usually sampled from discrete Gaussian distribution and is small in terms of the coefficient's size. Whenever we perform a homomorphic operation, we add additional noise to the ciphertexts. At some point, if the noise becomes too big, we will not be able to decrypt the ciphertext successfully. In this paper, we use a fully homomorphic encryption scheme, where the ciphertext is refreshed using the bootstrapping technique to reduce the noise level whenever it exceeds a certain threshold. We choose the TFHE scheme [3] to implement our algorithms, as its bootstrapping technique is programmable to evaluate complex functions.

Gentry [2] defines *Bootstrapping* as a technique to reduce the noise in the ciphertext when it grows beyond a specific limit. It is done using a bootstrapping key, which is nothing but the encryption of a secret key using another secret key, i.e., $bk = Enc(sk)$. Given a ciphertext ct corresponding to encryption of plaintext m, then $Dec_{sk}(ct) = m$. Let $F(x) = Dec_x(ct)$ be a function which decrypts a ciphertext given a secret key x. Let ct' be the decryption using the bootstrapping key bk.

$$ct' = F(bk) = F(Enc(sk)) = Enc(F(sk)) = Enc(Dec_{sk}(ct)) = Enc(m) \tag{1}$$

Hence, ct' is another encryption of the same message m, but with a lower noise level.

Let us denote $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ as the real torus. $\mathbb{T}_q := q^{-1}\mathbb{Z}/\mathbb{Z}$ is the real torus modulo q. $\mathbb{B}$ is the integer subset $\{0,1\}$. We use the notation $a \xleftarrow{\$} S$ to denote that a is sampled uniformly at random from a set S. Similarly, $b \leftarrow \mathcal{D}$ denotes sampling from a probability distribution $\mathcal{D}$. For a real number x, $\lfloor x \rceil$ denotes the nearest integer to x. The standard algorithms [3] for key generation, encryption, decryption, homomorphic operations, and bootstrapping are below.

KeyGen (1^λ) On input security parameter λ, the key generator generates a secret key sk and public parameters $\{n, \sigma, p, q\}$. n is a positive integer. p and q are chosen such that $p \mid q$. A normal distribution $\chi = \mathcal{N}(0, \sigma^2)$ over $\mathbb{R}$ is used to create a discretized error distribution $\hat{\chi}$ over $q^{-1}\mathbb{Z}$. The private key sk is sampled uniformly at random from $\mathbb{B}^n$, i.e., $sk = (s_1, \ldots, s_n) \xleftarrow{\$} \mathbb{B}^n$. The plaintext space is $\mathcal{P} = \mathbb{T}_p \subset \mathbb{T}_q$.

Encrypt $_{\mathbf{sk}}(\mu)$ The encryption of $\mu \in \mathcal{P}$ is given by

$$c \leftarrow \mathrm{TLWE}_s(\mu) = (a_1, \ldots, a_n, b) \in \mathbb{T}_q^{n+1} \tag{2}$$

with

$$\begin{cases} \mu^* = \mu + e \\ b = \sum_{j=1}^{n} s_j \cdot a_j + \mu^* \end{cases} \tag{3}$$

for a random vector $(a_1, \ldots, a_n) \xleftarrow{\$} \mathbb{T}_q^n$ and a small noise $e \leftarrow \hat{\chi}$

Decrypt $_{\mathbf{sk}}(c)$ To decrypt $c = (a_1, \ldots, a_n, b)$, use private key $s = (s_1, \ldots, s_n)$, compute

$$\mu^* = b - \sum_{j=1}^{n} s_j \cdot a_j \tag{4}$$

and return

$$\mu = \frac{\lfloor p\mu^* \rceil \bmod p}{p} \tag{5}$$

Addition of ciphertexts Let $c_1 \leftarrow \text{TLWE}_s(\mu_1)$ and $c_2 \leftarrow \text{TLWE}_s(\mu_2)$ be respective TLWE encryptions of μ_1 and μ_2, i.e., $c_1 = (a_1, \ldots, a_n, b)$, $c_2 = \left(a'_1, \ldots, a'_n, b'\right)$, $b' = \sum_{j=1}^{n} s_j \cdot a'_j + \mu_2 + e_2$, and e_1, e_2 small. Then

$$c_3 := c_1 + c_2 \tag{6}$$

is a valid encryption of $\mu_3 := \mu_1 + \mu_2$ provided that the additive noise $e_3 := e_1 + e_2$ remains small.

Multiplication by a known constant Let $K > 0$ be a small integer, and ciphertext c be the $\text{TLWE}_\text{s}(\mu)$ where $\mu \in \mathcal{P}$. By multiplying K with every vector component of c, we obtain $\text{TLWE}_\text{s}(K \cdot \mu)$. If $c = (a_1, \ldots, a_n, b) \in \mathbb{T}_q^{n+1}$, then

$$K \cdot c = (K \cdot a_1, \ldots, K \cdot a_n, K \cdot b). \tag{7}$$

provided that the resulting noise is kept small.

Bootstrapping In Gentry's seminal thesis [2], bootstrapping is performed by homomorphically decrypting a ciphertext using its key's encryption to reduce the noise in the ciphertext.

The bootstrapping of a TLWE ciphertext $c \leftarrow \text{TLWE}_s(\mu) \in \mathbb{T}_q^{n+1}$ encrypted with secret key $s = (s_1, \ldots, s_n) \in \mathbb{B}^n$ is done in three steps.

- $c' \leftarrow \text{BlindRotate}_{bsk}(c)$
  This operation returns a ring (torus polynomials) LWE ciphertext in $\mathbb{T}_{N,q}[X]^{k+1}$. This operation rotates a test polynomial homomorphically (the reason for naming it blind rotation) using a bootstrapping key, which consists of a list of encryptions of the elements of the secret key s.
- $c' \leftarrow \text{SampleExtract}(c')$
  This operation extracts the constant coefficient of $\boldsymbol{c'}$, resulting in an LWE ciphertext in $\mathbb{T}_q^{N+1}$
- $c'' \leftarrow \text{KeySwitch}_{ksk}(c')$
  This final step uses key-switching keys to return to the original LWE ciphertext in $\mathbb{T}_q^{n+1}$, encrypted with the original secret key s.

More details can be found in reference [3].

Programmable Bootstrapping This technique converts a high-noise ciphertext into a new one with less noise while simultaneously evaluating a function on the encrypted data. It is achieved by encoding a look-up table in the test polynomial used in the blind rotate operation during bootstrapping. Consider a function $f : \mathbb{T}_p \to \mathbb{T}_p$ and a noisy ciphertext $c \leftarrow \text{TLWE}_s(\mu)$; programmable bootstrapping outputs a new ciphertext $c' \leftarrow \text{TLWE}_s(f(\mu))$, having a small amount of noise. Regular bootstrapping is a case of programmable bootstrapping where f is the identity function. The details regarding look-up table construction can be found in reference [4].

3.2. Combinational Digital Circuits

In this section, we give a brief overview of the combinational digital circuits that are used in our integer comparator circuits.

3.2.1. Comparator

A comparator decides whether two numbers are greater or less than the other or equal. It receives two n-bit binary numbers, X and Y, and outputs the comparison outcome as a bit. There are two common types of comparators. An equality comparator indicates whether X equals Y, and a magnitude comparator indicates whether the value of X is greater or lesser than the Y value.

The equality comparator is the more straightforward circuit compared to the magnitude comparator. It uses the XNOR gate to check that bits in X and Y corresponding to the same column are equal. If it is valid for all the columns, the numbers are equal.

The magnitude comparator also compares each bit of two numbers, such as $X = X_3X_2X_1$ and $Y = Y_3Y_2Y_1$. It starts from the most significant bits, checking if the pair of bits is equal and assigning the result to E_3.

$$E_3 = \bar{X}_3\bar{Y}_3 + X_3Y_3 \tag{8}$$

It then continues checking for the next most significant pair till it reaches the least significant pair of bits. Hence, $E_2 = \bar{X}_2\bar{Y}_2 + X_2Y_2$ and $E_1 = \bar{X}_1\bar{Y}_1 + X_1Y_1$.

If $X > Y$, then the following equation evaluates to 1.

$$X > Y = X_3\bar{Y}_3 + E_3X_2\bar{Y}_2 + E_3E_2X_1\bar{Y}_1 \tag{9}$$

Suppose the most significant bits are $X_3 > Y_3$, where $X > Y$, irrespective of the bit values in other places of the number. This logic corresponds to the first term $X_3\bar{Y}_3$ in the equation. On the other hand, if the most significant bits are equal, $X > Y$ if the next significant bit X_2 is greater than Y_2 as evaluated by the second term $E_3X_2\bar{Y}_2$ in the equation. Finally, the third term $E_3E_2X_1\bar{Y}_1$ evaluates to one if the least significant bit X_1 is greater than Y_1 and all other bits are the same.

The IC 7485 is a TTL family 16-pin 4-bit magnitude comparator with outputs $X = Y, X > Y$, and $X < Y$. Words of larger length are compared by cascading these comparator chips one after another, as shown in Figure 2.

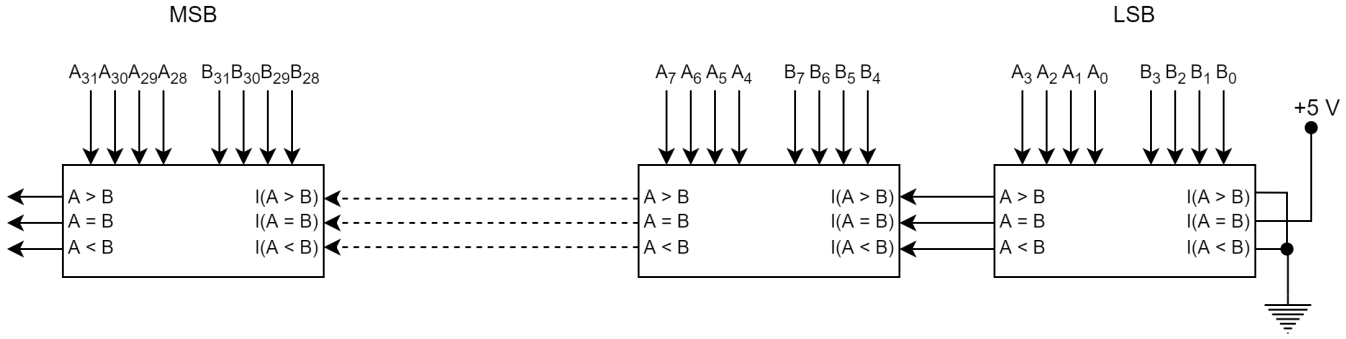


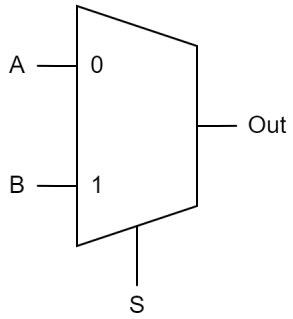
Figure 2. A 32-bit comparator circuit by cascading 4-bit IC 7485 comparator.

The rightmost cascading inputs are connected to fixed logic values, and the leftmost cascading outputs provide important final outputs. Instead of the 4-bit comparator, a single-bit comparator can be used at each step. It tracks whether the numbers corresponding to the compared bit pairs are greater or lesser.

3.2.2. Multiplexer

Multiplexing means transmitting many information units over a smaller number of channels or lines. The function of a multiplexer is to choose an output from a selection of inputs such as a rotary switch. A digital multiplexer (MUX) is a combinational circuit that selects one of 2^n data input lines based on an n-bit address, directing it to one output line. A set of selection lines controls the selection of a particular input line.

A typical multiplexer has 2^n information inputs $(I_{(2^n-1)}, \ldots, I_0)$, n select inputs $(S_{n-1}, \ldots, S_0)$ and one information output Y. The simplest multiplexer when $n = 1$ is a 2^1-to-1 multiplexer, shown in Figure 3. The single selection variable S has two values, 0 and 1. When $S = 0$, the multiplexer selects and outputs the first input, A. Similarly, when $S = 1$, the multiplexer selects the second input B. Optimizing the truth table with Karnaugh maps gives the equation $Out = \bar{S}A + SB$.



S	A	B	Out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 3. The 2-to-1 MUX and its truth table.

3.3. Decision Tree

A decision tree is a popular classification model capable of fitting complex datasets. They are the fundamental building blocks for powerful machine learning algorithms, such as random forest. Decision tree training algorithms, such as CART, produce a binary tree where the interior nodes always contain two children. It is built by recursively splitting the data into smaller parts until the data are split into a single leaf node. The decision tree is a white-box model, where their decisions are more accessible to interpret than a black-box model, such as neural networks.

Given an attribute vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{Z}^n$, the decision tree evaluates the input vector x and returns the output class. The decision tree with m internal nodes implements a function $\mathcal{T} : \mathbb{Z}^n \to \mathbb{Z}$ that maps an attribute vector $\mathbf{x} \in \mathbb{Z}^n$ to a classification result in $\{v_1, \ldots, v_{m+1}\}$. A decision tree with m decision nodes is a binary tree with $m + 1$ leaf nodes. The leaf nodes contain the output class v_i, and the interior nodes contain the threshold value τ_i. Figure 4 shows a decision tree with $m = 3$ decision nodes. The decision at each node is a comparison test on one of the feature attributes $x_{\sigma(i)}$ to a threshold value τ_i for node i. The $\sigma(i)$ is the feature attribute index used to make the decision. Decision tree evaluation starts from the root. It then moves on to the left or right child node based on the test on the current node. This step continues until reaching a leaf node storing the output class. The depth of a decision tree, denoted by d, is the length of the longest path between the root and any leaf.

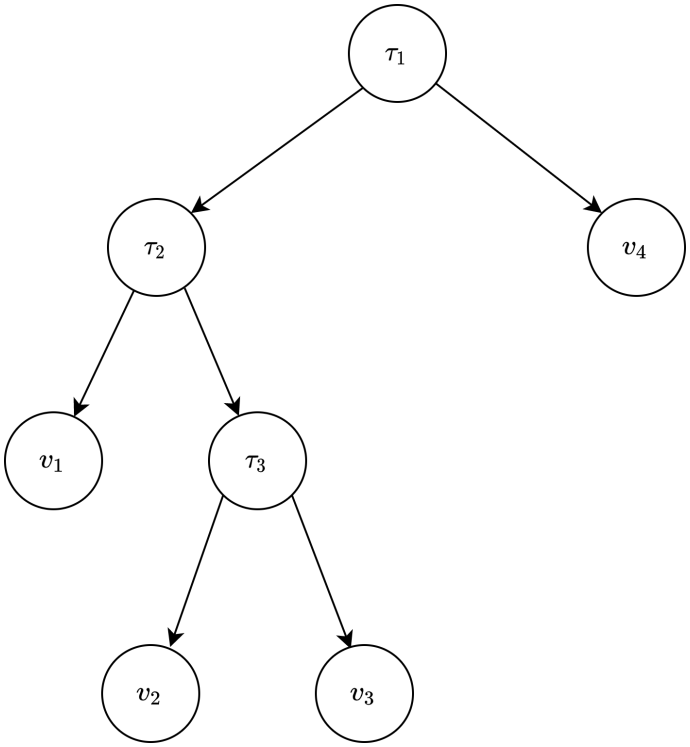


Figure 4. Decision tree with three internal nodes containing threshold value τ_i and four leaf nodes containing classification result v_i.

4. Private Comparison of Integers

This section presents our novel comparison algorithm for encrypted integers, which is the critical component of our privacy-preserving decision tree evaluation.

We encrypt an unsigned integer by considering it a sequence of 4-bit integers or nibbles. The first nibble is the most significant, and the last nibble is the least significant. Each of these nibbles is encrypted separately using TFHE [3] encryption. The encrypted nibbles are then concatenated to form the encrypted integer.

The 4-bit integers are encoded to the plaintext space $\mathcal{P} = \mathbb{T}_p$, where $p \mid q = 2^{64}$. The value of p should be greater than 4-bits. As the leading bits, these extra bits accommodate the result of homomorphic operations such as addition. These leading bits are called padding bits [4] and are initialized to zero. The number of padding bits is denoted by $\bar{\omega}$.

4.1. Nibble Comparator

We define two operations on encrypted nibble: HE-NIBBLE-EQ and HE-NIBBLE-GT. The first operation is a homomorphic equality operation, and the second is a homomorphic greater-than operation. Both of the operations start by subtracting the first encrypted nibble from the second encrypted nibble, as shown in Figure 5. The subtraction result is a ciphertext with $\bar{\omega} - 1$ bits of padding.

The resulting ciphertext is then bootstrapped, using programmable bootstrapping. The univariate function, passed to the bootstrapping function for equality, checks if the input is zero. If it is true, it returns one. Else, it returns zero. Similarly, the univariate function checks if the input is greater than zero for greater-than checking. We also specify the programmable bootstrapping function to output the final ciphertext with the initial $\bar{\omega}$ bits of padding.

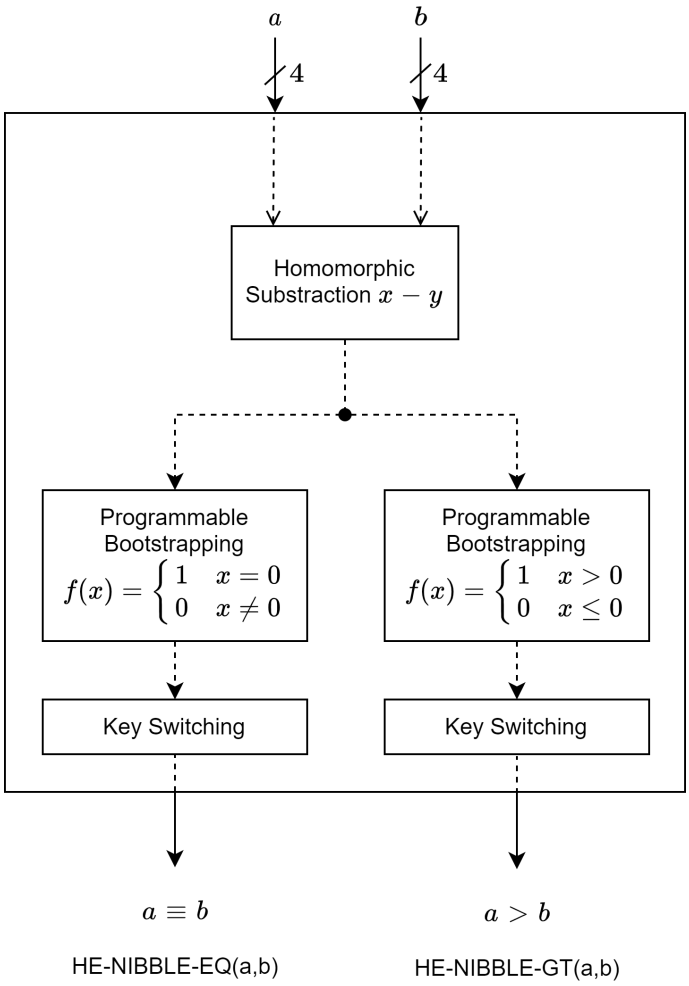


Figure 5. Encrypted nibble comparator: HE-NIBBLE-EQ and HE-NIBBLE-GT.

4.2. Integer Comparator

We compare the encrypted integers by cascading encrypted nibble comparators, similar to cascading comparators in Figure 2. Each comparator takes in two encrypted nibbles and the output from the previous comparator. If both the encrypted nibbles are equal, then the output of the comparator is the same as the output of the previous comparator. If the encrypted nibbles are not equal, then the output of the comparator is as follows. If the first encrypted nibble is greater than the second encrypted nibble, it is one. Otherwise, it is zero. A 2-to-1 multiplexer can realize this.

The equality and greater-than evaluation of the encrypted nibbles are done using the nibble comparator described in Figure 5. We design a homomorphic multiplexer HE-MUX algorithm which takes encrypted bits in the input and selection lines as described in Algorithm 1. Let A and B be the inputs and S be the selection line of the mux. We encode the A, B and S as bits of a binary number represented as SAB_2, with S being the most significant bit. The equivalent decimal number is deduced by multiplying S by 2^2, A by 2^1, and B by one and then adding all the terms. The output of the multiplexer is the encrypted bits of the decimal number.

We achieve the binary to decimal conversion homomorphically by using the parallel prefix sum [20] technique shown in Algorithm 2. If the encrypted bits have a padding $\bar{\omega}$ of 4, it is arranged in an array of length $n = 8$. S is duplicated four times, and A is duplicated twice in the array. We append encryptions of zero to fill the array. We then use the PARALLEL-PREFIX-SUM algorithm to sum the encrypted bits. The output is the encrypted sum.

Algorithm 1 HE-MUX

Input: Condition S, Input A, Input B, Zero $\emptyset$
Output: A if Condition $S \equiv 1$, B if $S \equiv 0$
$input \leftarrow [S, S, S, S, A, A, B, \emptyset]$
$sum \leftarrow$ PARALLEL-PREFIX-SUM$(input)$
$result \leftarrow$ PBS$(sum, f_{MUX}(x))$
$result \leftarrow$ KS$(result)$
return *result*

Algorithm 2 PARALLEL-PREFIX-SUM

Input: An encrypted array $\mathbb{X}$ of length n, encryption of zero $[\![0]\!]$
Output: Sum of all the elements of the array $\mathbb{X}$
$m \leftarrow 2^{\lceil \log_2 n \rceil}$
Append $m - n$ zeros to the array $\mathbb{X}$
for $i \leftarrow 0$ to $\lceil \log_2 n \rceil - 1$ **do**
  for $j \leftarrow 0$ to $m - 1$ **do**
    $X[j + 2^{i+1} - 1] \leftarrow$ HE-NIBBLE-ADD $(X[j + 2^i - 1], X[j + 2^{i+1} - 1])$
    $j \leftarrow j + 2^{i+1}$
  end for
end for
return $X[m - 1]$

Adding two ciphertexts decreases the padding and increases the noise in the output ciphertext. This decrease happens at every algorithm iteration, as shown in Figure 6. The PARALLEL-PREFIX-SUM algorithm ensures that the two ciphertexts in the add operation have the same padding.

The resulting sum is then bootstrapped using programmable bootstrapping with the function $f_{MUX}(x)$ that evaluates the following truth table of the multiplexer shown in Figure 3.

$$f_{MUX}(x) = \begin{cases} 0 & x = 000_2 \\ 0 & x = 001_2 \\ 1 & x = 010_2 \\ 1 & x = 011_2 \\ 0 & x = 100_2 \\ 1 & x = 101_2 \\ 0 & x = 110_2 \\ 1 & x = 111_2 \end{cases} \tag{10}$$

The final value is key switched to obtain the multiplexer output. All the operations in the multiplexer are summarized in Figure 7.

Our comparison algorithm PVTCMP of two encrypted integers of length n nibbles is summarized in Algorithm 3. If the encrypted integers contain just one nibble, then the result of the nibble comparator function HE-NIBBLE-GT is used. Otherwise, the encrypted integers are traversed from the least significant nibble to the most significant nibble. The encrypted nibbles are compared using the nibble comparator functions HE-NIBBLE-GT and HE-NIBBLE-EQ, and multiplexed using the HE-MUX multiplexer. The result is rippled from right to left, as shown in Figure 8, and the final result from the most significant nibble is returned.

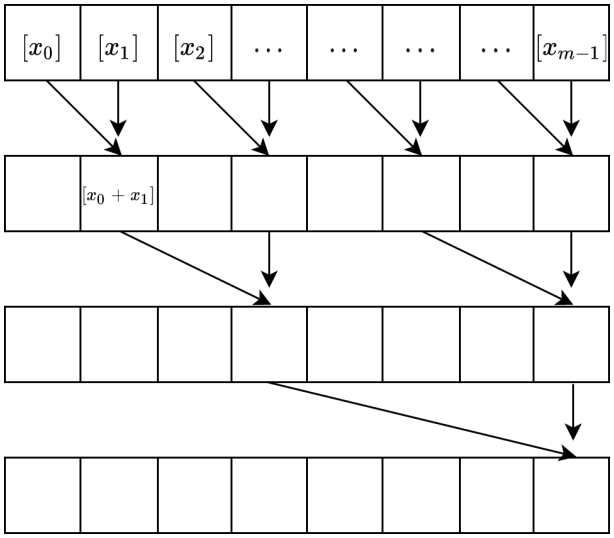


Figure 6. PARALLEL-PREFIX-SUM algorithm data flow.

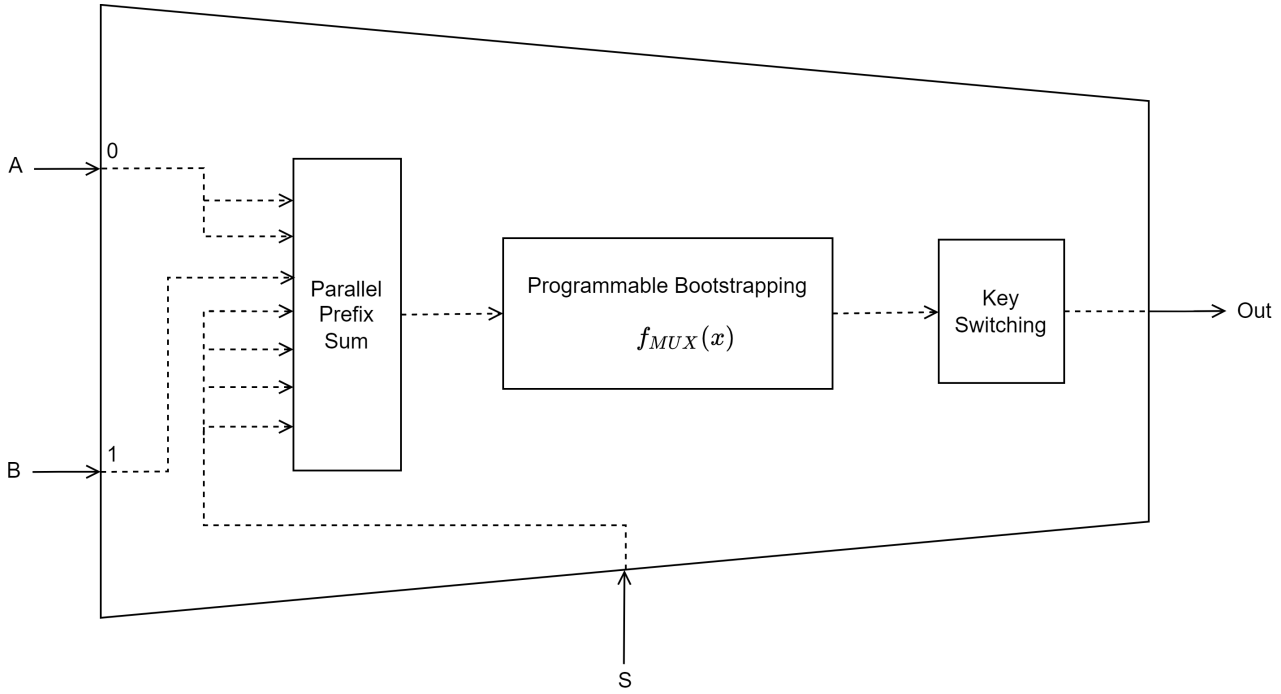


Figure 7. Multiplexer with encrypted inputs and output.

Algorithm 3 PVTCMP: ENCRYPTED GREATER THAN COMPARISON FUNCTION

Input: An encrypted array $\mathbb{X}$, $\mathbb{Y}$ of length n nibbles, encryption of zero $[\![0]\!]$
Output: Encrypted 1 if $\mathbb{X} > \mathbb{Y}$, encrypted 0 otherwise
if $n \equiv 1$ **then**
  return HE-NIBBLE-GT$(\mathbb{X}[0], \mathbb{Y}[0])$
end if
$result \leftarrow 0$
for $i \leftarrow 0$ to $n - 1$ **do**
  $eq \leftarrow$ HE-NIBBLE-EQ $(\mathbb{X}[i], \mathbb{Y}[i])$
  $gt \leftarrow$ HE-NIBBLE-GT $(\mathbb{X}[i], \mathbb{Y}[i])$
  $result \leftarrow$ HE-MUX $(eq, result, gt, 0)$
end for
return *result*

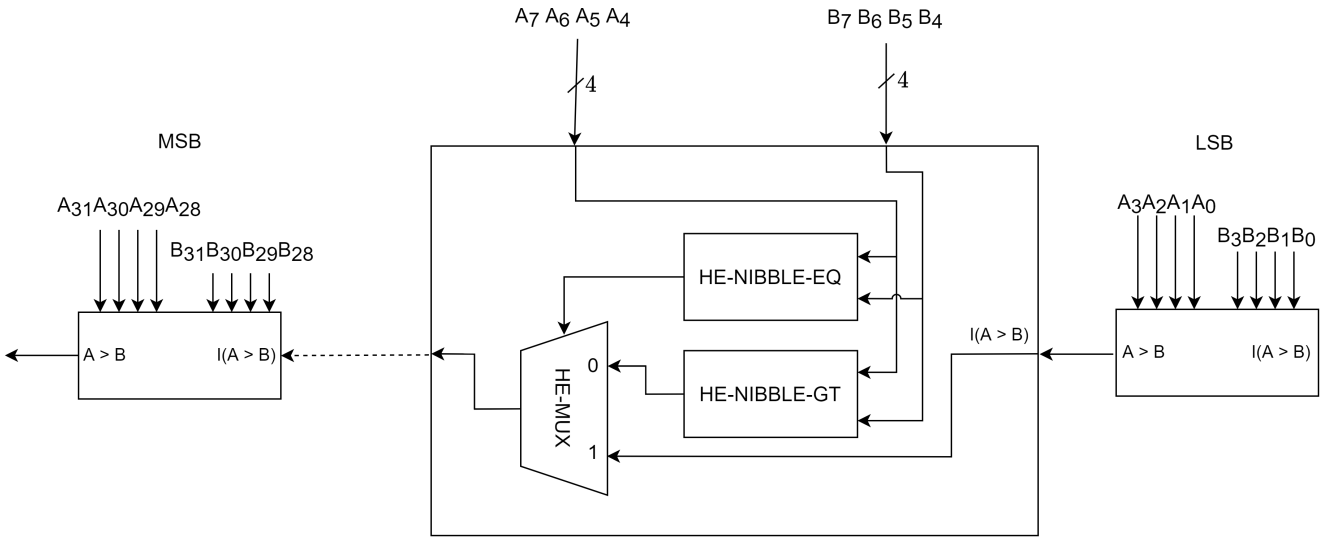


Figure 8. Encrypted 32-bit comparator circuit by cascading 4-bit encrypted comparator.

5. Applications of Private Integer Comparison: Decision Trees

In this section, we consider the application of our private integer comparison to decision trees.

5.1. Private Evaluation of Decision Trees

Recall from Figure 4 that evaluating the decision tree requires comparing each node threshold with one of the feature vector values. This comparison is made using our PVTCMP function, resulting in a tree with encrypted comparison results. For each node i, we assign $b_i \leftarrow \text{PVTCMP}(x_{\sigma(i)}, \tau_i)$.

After computing all the decision values in the internal nodes, we assign a cost to each edge similar to Tai et al. [21], as shown in Figure 9. If the value of b_i is 1, then the path cost on its right edge will be 0, whereas the path cost on the left edge will be 1. In other words, the traversal will have a penalty of 1 for continuing on the left edge. Similarly, if the value of b_i is 0, then the path cost on its left edge will be 0, whereas the path cost on the right edge will be 1. The value $1 - b_i$ is computed using programmable bootstrapping with the function below, which inverts the input bit.

$$f(x) = \begin{cases} 1 & x = 0 \\ 0 & x \neq 0 \end{cases} \tag{11}$$

For each leaf node, we construct a path from the root node consisting of all the edges from it to the leaf node. The cost of a path is computed by adding the path cost of all the edges in its path. We make use of our PARALLEL-PREFIX-SUM algorithm to compute the path cost. If the length is less than 2^m, we append zero encryption to the path. If the length is greater than 2^m, then we split the path into chunks of size 2^m and compute the path cost recursively by adding the path cost of the chunks. The paths of interest are those with a path cost of zero. Hence, we invert the path cost using a programmable bootstrapping function; the paths with zero path cost return one, and those with non-zero path cost return zero. The details of the algorithm are shown in Algorithm 4. The outputs of IS-ZERO-PATH-COST are multiplied with the corresponding leaf value v_i, and the terms are added to obtain the final output.

Figure 10 shows the complete private decision tree evaluation. The client sends the encrypted integer as a sequence of encrypted nibble $[x_1]$, $[x_2]$, ... $[x_n]$ to the server. The model owner also encrypts the decision tree thresholds $[\tau_1], [\tau_2], \ldots [\tau_m]$ to the server. The server computes the private comparison values $[b_1]$, $[b_2]$, ... $[b_m]$ for each of the m interior nodes. The server then computes the edge costs $[ec]$ of each node. For all the paths P_i

where $i \in \{1, m+1\}$ from root to the leaf node, we calculate its edge path cost pc_i using the PARALLEL-PREFIX-SUM algorithm. The $\widetilde{pc_i}$ is calculated by inverting the pc_i bit to indicate whether the path has zero edge path cost. The final result is obtained by pairwise multiplying $\widetilde{pc_i}$ with v_i to obtain the final v value. The server sends the encrypted $[v]$ to the client, and the client decrypts the $[v]$ to obtain the final classification.

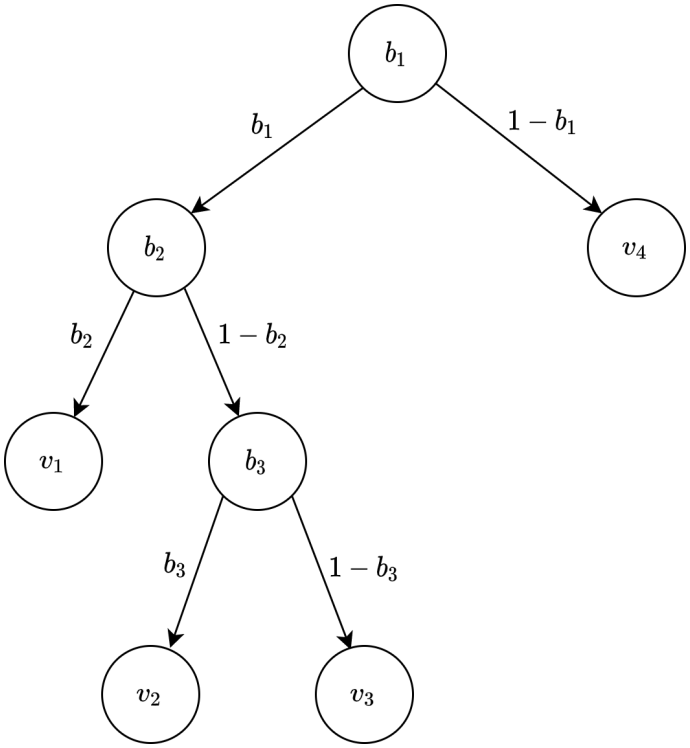


Figure 9. Decision tree evaluation with threshold comparison results and edge costs.

Algorithm 4 IS-ZERO-PATH-COST

Input: An encrypted bit array $\mathbb{X} \in [\![\mathbb{B}]\!]^n$, encryption of zero $[\![0]\!]$
Output: An encrypted bit $b \in [\![\mathbb{B}]\!]$, such that $b \leftarrow [\![1]\!]$ if $[\![1]\!] \in \mathbb{X}$
Split the array $\mathbb{X}$ into $\mathbb{X}_0, \mathbb{X}_1, \ldots \in [\mathbb{B}]^k$, where $k \leq n$ and $k = 2^m$. Append $[0]$ to the last split if $k \nmid n$.
for all $\mathbb{X}_i$ **do**
    $partial\_sum[i] \leftarrow$ PARALLEL-PREFIX-SUM$(\mathbb{X}_i)$
    $partial\_sum[i] \leftarrow \text{PBS}\left(partial\_sum[i], \lambda(x) = \begin{cases} 1 & x = 0 \\ 0 & x \neq 0 \end{cases}\right)$
    $partial\_sum[i] \leftarrow \text{KS}(partial\_sum[i])$
end for
if *partial_sum* has more than one element **then**
    return IS-ZERO-PATH-COST$(partial_sum)$
else
    return $partial\_sum[0]$
end if

5.2. Extensions to Private Decision Tree Evaluation

In our protocol, we used label encoding to encode the class labels. Each leaf node has the output class encoded in v_i as an integer value. This encoding is efficient for binary classification problems, or the number of classes is small. The scalar multiplication of v_i with $\widetilde{pc_i}$ using Equation (7) is only feasible if the scaling of the noise is small. Therefore, we must use a different approach for multi-class problems and regression trees.

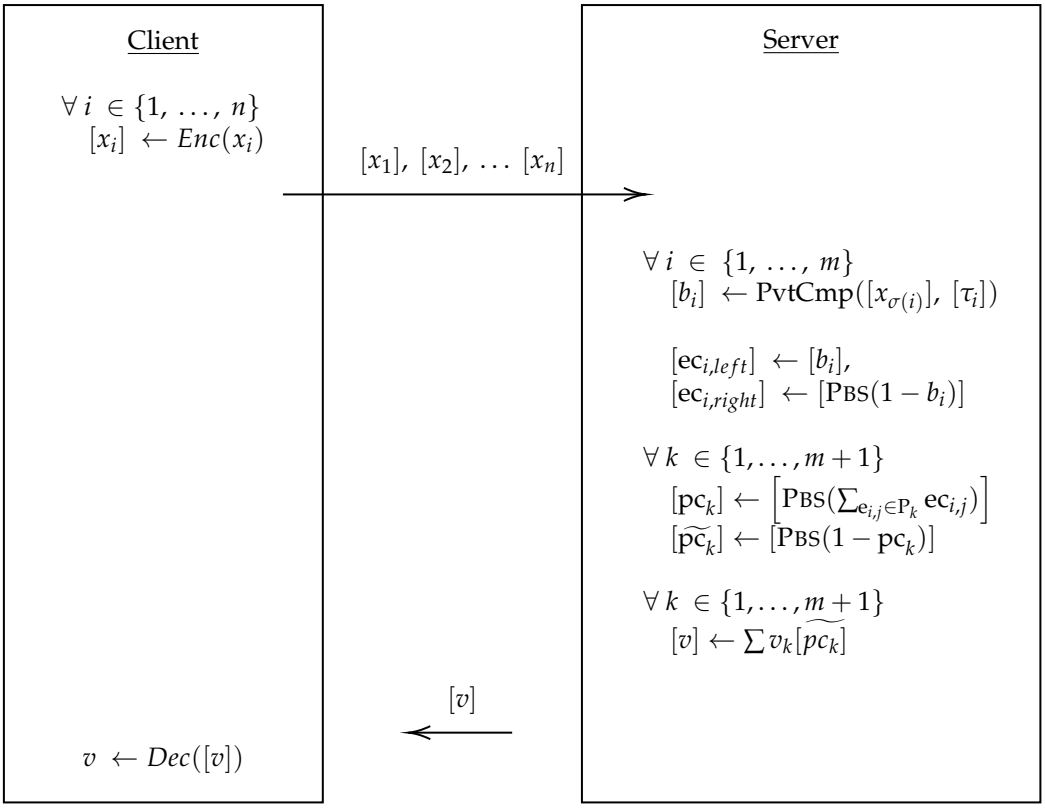


Figure 10. Private decision tree evaluation using our private comparison function.

5.2.1. Multi-Class Decision Trees and Regression Trees

A one-hot encoding scheme encodes the class labels for multi-class problems. The one-hot encoding is a vector where the ith element is one if the ith class is selected and zero otherwise. We encode the output classes v_i in the leaf nodes as shown in Figure 11. The scalar multiplication with $\widetilde{pc_i}$ is done element-wise using the vector. The vectors are then added element-wise using PARALLEL-PREFIX-SUM to obtain the final v as an encrypted one-hot vector.

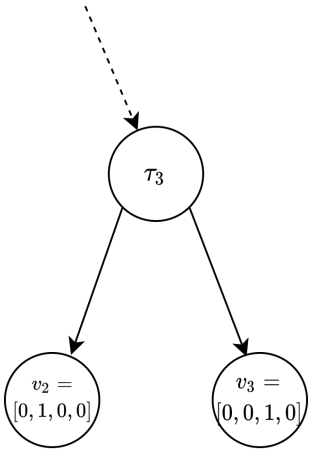


Figure 11. Decision tree with one-hot encoded labels.

We use a similar approach for regression trees, where the output values are encoded as a binary vector. The final value after scalar multiplication and element-wise addition is the binary value of the regression tree.

5.2.2. Random Forest Classification

A random forest is a collection of decision trees, where the final decision is made by taking the majority vote of the decision trees. Even though the random forest algorithm can be used for regression, we focus on classification problems. Like multi-class classification, each decision tree in the random forest returns its output class as a one-hot encoded vector.

Once the server obtains the individual decision tree results, the server computes the final result by adding element-wise again using the PARALLEL-PREFIX-SUM algorithm. The final result vector is then sent to the client. The client decrypts the result, and the class with the highest vote is selected as the final classification.

6. Evaluation

We evaluate the computational complexity and the performance of our private decision tree evaluation.

6.1. Computational Complexity

Let n be the number of entries in the feature vector and m be the number of decision nodes in the decision tree. Let these numbers be represented as t-bit integers. Since we encrypt all the integers as a sequence of 4-bit nibbles, encrypting n entries in the feature vector takes $O(n \times \lceil t/4 \rceil)$ steps. Similarly, comparing m decision node thresholds takes $O(m \times \lceil t/4 \rceil)$ steps. The time taken for the edge path cost is negligible compared to the threshold comparison.

We compare our protocol with recent protocols by Bost et al. [19], Wu et al. [22], and Tai et al. [21]. Table 1 shows all protocols' computational complexity.

Table 1. Computational complexity summary.

Protocol	Complexity		Rounds	Scheme	Leakage	
	Client	Server			Client	Server
Bost et al. [19]	$O((n+m)t)$	$O(mt)$	≥ 6	Levelled-FHE	None	m
Wu et al. [22]	$O((n+m)t+d)$	$O(mt+2^d)$	6	AHE	None	m, d
Tai et al. [21]	$O((n+m)t)$	$O(mt)$	4	AHE	None	m
This work	$O(n \times \lceil t/4 \rceil)$	$O(m \times \lceil t/4 \rceil)$	2	TFHE w PBS	None	None

6.2. Experimental Results

We implemented the comparison function and the classifiers in Rust using the concrete (https://github.com/zama-ai/concrete/tree/concrete-0.1.11 (accessed on 4 August 2022)) library [23] for the TFHE implementation. Our performance evaluations were run on a server with Intel Xeon Platinum 8170 (64-bit) processors running at 3.7 GHz and 192 GB RAM.

We chose the statistical security parameter λ to be 128 and instantiated the TFHE scheme with the following parameters by the LWE estimator [24]. We used the default value for $q = 2^{64}$ given by the library. The number of padding bits $\bar{\omega}$ was set to 4 while encoding the 4-bit integers. Hence, the value of $p = 2^{4+4}$, i.e., uses 4-bits precision for representing the number and another 4-bits for padding.

The LWE key has a dimension of 750 and a noise with a standard deviation of 2^{-18}. For the RLWE key, we used a polynomial of size 2048 and a standard deviation of 2^{-52}. The bootstrapping key has base 2^7 and level 3. The key-switching key has base 2^2 and level 7.

We ran benchmarks for the performance of our comparison function and XCMP [6] non-interactive private method. Specifically, we measured the computation time for evaluating comparison, excluding the encryption and decryption time. The integer input size varied from 12 bits to 32 bits. The value of bit length for XCMP could only be measured to 19 bits because of the constraints of HElib [25]. The rest of the values for XCMP were extrapolated for higher bit length. The results are shown in Figure 12. Our comparison function for 32-bit and 64-bit integers is 26% faster than the naive TFHE implementation.

A more efficient version of XCMP was mentioned in [26], but we could not compare its performance since its source code is not open sourced.

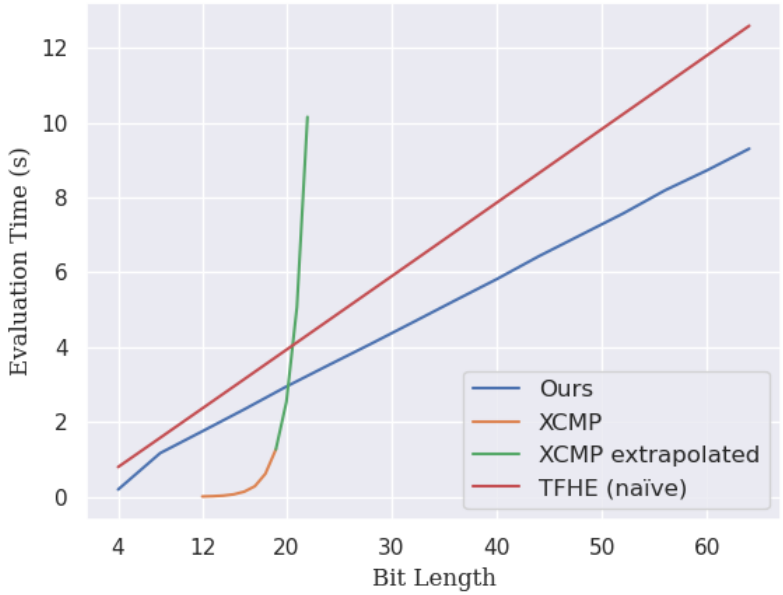


Figure 12. Evaluation time of our comparison method, TFHE [3] and XCMP [6], for comparing two encrypted value aspects to various bit lengths.

We measured the path cost computation time for long paths to see the impact on evaluation trees with high depth. The computation time grows linearly with the length of the path in every step of 4. The results are shown in Figure 13.

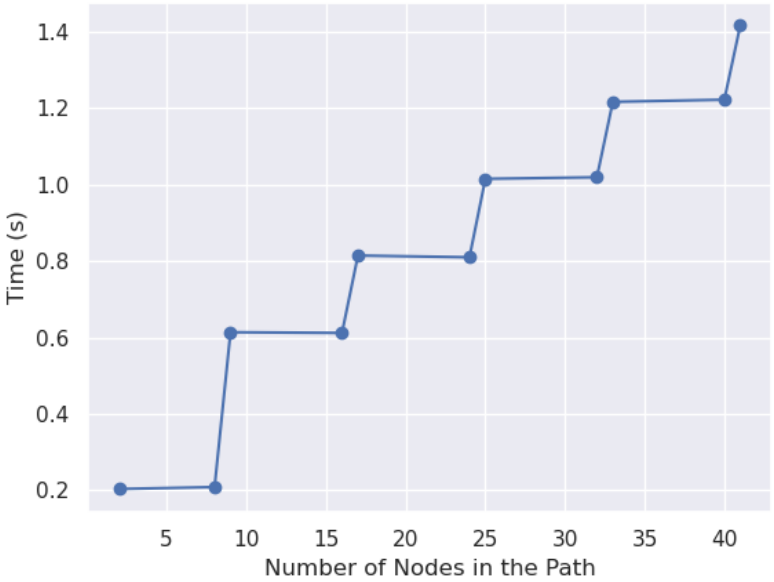


Figure 13. Time taken for path cost calculation.

We trained our decision tree models on real datasets from the UCI repository [5] using the scikit-learn library [27]. The training phase was done on plain data without any encryption. The number of features, decision nodes and the depth of the trained models are summarized in Table 2. We measured the performance of our secure evaluation after both the client's input and the decision tree thresholds were encrypted.

The performance gain achieved by the XCMP protocol is hindered by the fact that the feature vectors and decision tree thresholds cannot be more than 13 bits long. The work

of Tai et al. [21], on the other hand, is an interactive protocol where the server needs to communicate with the client to determine the result.

Table 2. Performance of secure evaluation on real datasets from UCI repository.

Dataset	n	d	m	Evaluation Time (s)			
				This Work		XCMP [6]	Tai et al. [21]
				32-bit	64-bit	13-bit	64-bit
Heart-disease	13	3	5	24.29	47.52	0.59	0.25
Credit-screening	15	4	5	24.29	47.52	–	0.27
Breast-cancer	9	8	12	58.30	114.04	–	0.34
Housing	13	13	92	446.98	874.29	10.27	1.98
Spambase	57	17	58	281.79	551.19	6.88	1.80

The final LWE encrypted classification result is serialized and sent to the client. Hence, the download bandwidth is a constant size, irrespective of the structure of the decision tree. The JSON serialization of LWE ciphertext of dimension 750 takes 17 KB of space. This size is less than half the download bandwidth required by Wu et al. [22] for the heart-disease and credit-screening datasets. The download bandwidth for Wu et al. for other real datasets in Table 2 is more than a hundred times larger than our result ciphertext.

7. Conclusions and Future Work

This paper introduced a new non-interactive integer comparison protocol using TFHE with programmable bootstrapping. This protocol was tested with 128-bit security and scaled linearly with the input bit length. As an application, we developed an efficient protocol for the private evaluation of decision trees. The evaluation is done in a non-interactive manner, where there is no interaction between the client and server. The model owner can delegate the evaluation to a third-party evaluator, and the client can be offline after sending the feature vector to the evaluator. We experimented with widely used real datasets from the UCI machine learning repository and compared them with the existing protocols. The protocol can be extended to perform more complex inferences using random forest classification.

An important future extension to the protocol is to support ensemble and gradient boosting methods. This extension is possible if we can extend the protocol with homomorphic operations, such as division on encrypted numbers. Our solution evaluates the tree encrypted with the client's public key. Therefore, it requires a key distribution mechanism to share the client's public key with the model owner. An exciting direction for future work is to use multi-key homomorphic encryption [28,29] to encrypt the tree and the feature vector with different keys. This technique eliminates the need for key distribution between the client and the model owner.

Author Contributions: J.P.: Conceptualization, Methodology, Software, Validation, Writing—Original Draft; B.H.M.T.: Conceptualization, Methodology, Writing—Review and Editing; B.V.: Supervision, Writing—Review and Editing; K.M.M.A.: Supervision, Resources, Funding Acquisition. All authors have read and agreed to the published version of the manuscript.

Funding: Jestine Paul, Benjamin Hong Meng Tan and Khin Mi Mi Aung are supported by A*STAR under its RIE2020 Advanced Manufacturing and Engineering (AME) Programmatic Programme (Award A19E3b0099).

Data Availability Statement: The dataset used for this study is publicly available at UCI Machine Learning Repository [5].

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Fredrikson, M.; Jha, S.; Ristenpart, T. Model inversion attacks that exploit confidence information and basic countermeasures. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 1322–1333.
2. Gentry, C. Fully homomorphic encryption using ideal lattices. In Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing, Bethesda, MD, USA, 31 May– 2 June 2009; pp. 169–178.
3. Chillotti, I.; Gama, N.; Georgieva, M.; Izabachène, M. TFHE: Fast fully homomorphic encryption over the torus. *J. Cryptol.* **2020**, *33*, 34–91. [CrossRef]
4. Chillotti, I.; Joye, M.; Paillier, P. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Proceedings of the International Symposium on Cyber Security Cryptography and Machine Learning, Be'er Sheva, Israel, 8–9 July 2021; Springer: Berlin/Heidelberg, Germany; pp. 1–19.
5. Dua, D.; Graff, C. *UCI Machine Learning Repository*; University of California: Berkeley, CA, USA, 2017.
6. Lu, W.J.; Zhou, J.J.; Sakuma, J. Non-interactive and output expressive private comparison from homomorphic encryption. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security, Incheon, Korea, 4–8 June 2018; pp. 67–74.
7. Iliashenko, I.; Zucca, V. Faster homomorphic comparison operations for BGV and BFV. *Proc. Priv. Enhancing Technol.* **2021**, *2021*, 246–264. [CrossRef]
8. Agrawal, R.; Srikant, R. Privacy-preserving data mining. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, 16–18 May 2000; pp. 439–450.
9. Du, W.; Zhan, Z. *Building Decision Tree Classifier on Private Data*; Syracuse University: New York, NY, USA, 2002.
10. Yao, A.C. Protocols for secure computations. In Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (SFCS 1982), Chicago, IL, USA, 3–5 November 1982; pp. 160–164.
11. Brickell, J.; Porter, D.E.; Shmatikov, V.; Witchel, E. Privacy-preserving remote diagnostics. In Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 28–31 October 2007; pp. 498–507.
12. Barni, M.; Failla, P.; Kolesnikov, V.; Lazzeretti, R.; Sadeghi, A.R.; Schneider, T. Secure evaluation of private linear branching programs with medical applications. In Proceedings of the European Symposium on Research in Computer Security, Saint-Malo, France, 21–23 September 2009; Springer: Berlin/Heidelberg, Germany; pp. 424–439.
13. Agrawal, R.; Kiernan, J.; Srikant, R.; Xu, Y. Order preserving encryption for numeric data. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France, 13–18 June 2004; pp. 563–574.
14. Boneh, D.; Lewi, K.; Raykova, M.; Sahai, A.; Zhandry, M.; Zimmerman, J. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, 26–30 April 2015; Springer: Berlin/Heidelberg, Germany; pp. 563–594.
15. Naveed, M.; Kamara, S.; Wright, C.V. Inference attacks on property-preserving encrypted databases. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 644–655.
16. Damgård, I.; Geisler, M.; Krøigaard, M. Efficient and secure comparison for on-line auctions. In Proceedings of the Australasian conference on Information Security and Privacy, Townsville, Australia, 2–4 July 2007; Springer: Berlin/Heidelberg, Germany; pp. 416–430.
17. Damgård, I.; Geisler, M.; Krøigaard, M. A correction to "Efficient and Secure Comparison for On-Line Auctions". *Cryptol. ePrint Arch.* **2008**, *2008*, 321. [CrossRef]
18. Veugen, T. Improving the DGK comparison protocol. In Proceedings of the 2012 IEEE International Workshop on Information Forensics and Security (WIFS), Tenerife, Spain, 2–5 December 2012; pp. 49–54.
19. Bost, R.; Popa, R.A.; Tu, S.; Goldwasser, S. Machine learning classification over encrypted data. *Cryptol. ePrint Arch.* **2014**, *2014*, 331.
20. Hillis, W.D.; Steele, G.L., Jr. Data parallel algorithms. *Commun. ACM* **1986**, *29*, 1170–1183. [CrossRef]
21. Tai, R.K.; Ma, J.P.; Zhao, Y.; Chow, S.S. Privacy-preserving decision trees evaluation via linear functions. In Proceedings of the European Symposium on Research in Computer Security, Oslo, Norway, 11–15 September 2017; Springer: Berlin/Heidelberg, Germany; pp. 494–512.
22. Wu, D.J.; Feng, T.; Naehrig, M.; Lauter, K. Privately Evaluating Decision Trees and Random Forests. *Proc. Priv. Enhancing Technol.* **2016**, *4*, 335–355. [CrossRef]
23. Chillotti, I.; Joye, M.; Ligier, D.; Orfila, J.B.; Tap, S. CONCRETE: Concrete operates on ciphertexts rapidly by extending TfhE. In Proceedings of the WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Virtual Event, 15 December 2020; Volume 15.
24. Albrecht, M.R.; Player, R.; Scott, S. On the concrete hardness of learning with errors. *J. Math. Cryptol.* **2015**, *9*, 169–203. [CrossRef]
25. Halevi, S.; Shoup, V. Design and implementation of HElib: A homomorphic encryption library. *Cryptol. ePrint Arch.* **2020**, *2020*, 1481.
26. Ishimaki, Y.; Yamana, H. Non-interactive and fully output expressive private comparison. In Proceedings of the International Conference on Cryptology in India, New Delhi, India, 9–12 December 2018; Springer: Berlin/Heidelberg, Germany; pp. 355–374.
27. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.

28. Chen, H.; Dai, W.; Kim, M.; Song, Y. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 395–412.
29. Chen, H.; Chillotti, I.; Song, Y. Multi-key homomorphic encryption from TFHE. In Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, 8–12 December 2019; Springer: Berlin/Heidelberg, Germany; pp. 446–472.