Oblivious (Un)Learning of Extremely Randomized Trees

Anonymous Authors

***Abstract*—While the use of homomorphic encryption (HE) for encrypted inference has received considerable attention, its application for the training of machine learning (ML) models remains comparatively underexplored, primarily due to the high computational overhead traditionally associated with fully homomorphic encryption (FHE). In this work, we address this challenge by leveraging the inherent connection between inference and training in the context of Extremely Randomized Trees (ERT), thereby enabling efficient training directly over encrypted data. More precisely, we instantiate this approach by the training of ERT within the TFHE framework. Our implementation demonstrates that it is possible to train ERTs on encrypted datasets with a runtime significantly lower than current state-of-the-art methods for training Random Forests in the encrypted domain while achieving comparable predictive accuracy. This result highlights a promising direction for practical privacy-preserving machine learning using FHE. Our second main contribution consists in leveraging the properties of ERTs to create the first ML model that enables *private unlearning*. This approach makes the unlearning process indistinguishable from training, thus allowing clients to conceal the true nature of the operations being conducted on the model.**

***Index Terms*—Privacy-preserving Machine Learning, Random Forests, Extremely Randomized Trees, Homomorphic Encryption, TFHE, Private Unlearning, Oblivious queries.**

1. Introduction

Decision trees are known for their interpretability and applicability across a wide range of domains—from finance [1] and healthcare [2] to cybersecurity and industrial automation [3]. In particular, by decomposing complex decisions into hierarchical sequences of simple rules, decision trees are inherently interpretable, making them highly suitable for settings in which explainability is essential. Over time, their performance has been significantly enhanced through ensemble learning techniques, such as bagging and boosting. In this landscape, Extremely Randomized Trees (ERTs) [4], a stochastic variant of the classic Random Forests algorithm, have emerged as a powerful model. By introducing additional randomness in the split selection process, ERTs promote higher variance across individual trees, which in turn can improve generalization and robustness.

As machine learning systems increasingly process sensitive data, ensuring the privacy of training and inference has become a central concern [5]. Fully Homomorphic Encryption (FHE) schemes offer a cryptographic solution to this challenge, enabling computation directly over encrypted data without ever decrypting it. While substantial advancements in applying FHE to inference tasks have been made recently, allowing encrypted predictions using pretrained models, the training of machine learning models over encrypted data remains an underexplored frontier [6]. This is due to the prohibitive computational overhead associated with FHE operations, despite the progress made in schemes like TFHE [7], which support efficient programmable bootstrapping and bit-level logic gates.

In this work, we bridge this gap by presenting a novel protocol to train ERTS on encrypted datasets by leveraging the capabilities of TFHE. Additionally, our protocol is oblivious, in the sense that both the tree structure and model outputs reveal no information about the underlying training data. To realize this, we tackle several challenges, including the encrypted computation of histograms, randomized split point selection and secure tree construction—entirely in the encrypted domain. Our construction is also designed to support machine unlearning, allowing the efficient removal of the impact of individual data points from the encrypted model. To the best of our knowledge, this work is the first realization of oblivious unlearning within a homomorphic encryption framework, making it a significant milestone towards adaptive privacy-preserving machine learning.

2. Related Work

Privacy-preserving decision tree evaluation has been an active research area for a decade, with most approaches leveraging advanced cryptographic primitives such as Secure Multiparty Computation (SMPC) [8], [9] and Homomorphic Encryption (HE) [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26] for model training and/or inference over encrypted or secret-shared data. Among HE-based protocols, schemes like BFV [27], CKKS [28] and TFHE [7] are particularly prominent due to their rich arithmetic capabilities and expressiveness.

Private decision tree inference. The most advanced solutions for inference over encrypted data typically exploit the batching features of HE schemes. For instance, Cong and collaborators [23] proposed a highly efficient protocol using batching-friendly cryptosystems. Their scheme leverages parallelism for significant throughput gains in inference while reducing comparison time by up to 72× compared to prior state-of-the-art approaches such as [21]. Within the TFHE ecosystem, Probonite [19] introduced the first

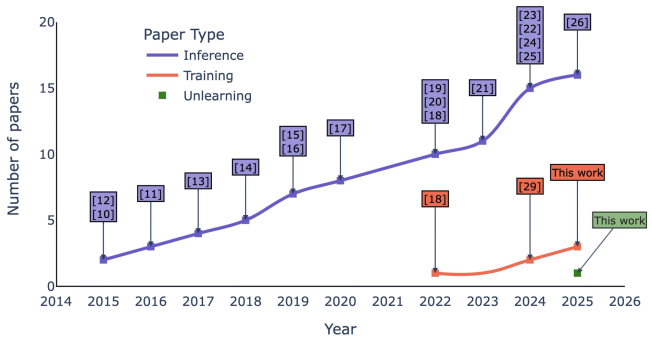


Fig. 1. Chronology of related work on tree-based model training and inference using HE.

protocol performing a number of non-interactive comparisons linear in the tree depth, substantially accelerating evaluation. SortingHat [20] further improved performance by integrating a new comparison technique and a transciphering layer that minimizes ciphertext size. More recently, Frery and co-authors [22] made substantial use of programmable bootstrapping to implement efficient tree traversal and comparisons, pushing the limits of bit-level encrypted inference.

Private decision tree training. Unlike inference, private training of decision trees has been less studied (see Figure 1) due to the challenge of selecting the best splits without leaking data. Akavia and collaborators [18] proposed one of the few interactive training schemes based on TFHE, using polynomial approximations for encrypted comparisons and achieving communication complexity linear in the tree depth. While efficient, their method requires interaction and incurs costly comparison evaluations. Our approach departs from this paradigm by employing a novel data encoding technique (detailed in Section 5.1), which eliminates the need for explicit comparisons, enabling a fully non-interactive and client-offline training procedure.

Recently, Shin and co-authors [29] proposed a random forest training scheme based on CKKS, which employs approximate sign functions for comparisons while leveraging SIMD parallelism. While CKKS-based batching is powerful, our lightweight construction and optimized TFHE-level comparison make our approach competitive, even without batching, especially in the context of ERTs in which simplicity and randomness reduce training complexity.

Private decision tree unlearning. Driven by privacy regulations such as the GDPR's and the "right to be forgotten" [30], the area of machine unlearning has gained traction [31], [32], [33]. While some efforts have examined unlearning on ERTs in the clear domain [34], our work is the first to introduce private unlearning for ERTs specifically in the context where both the training data and the unlearned instances remain encrypted throughout the process. More broadly, our proposal constitutes the first private unlearning protocol for an ML model under homomorphic encryption, setting a precedent for regulatory-compliant, encrypted model adaptability.

3. System and Adversary Models

We consider the traditional outsourcing setting in which a client holds private data and delegates machine learning tasks to a cloud server. Specifically, the client may request training, unlearning or inference operations on an ERT model hosted by the server. Our model also extends to a federated setting involving multiple clients, each holding their own private dataset. These clients collaboratively contribute to a shared classifier maintained by the server, enabling joint training, unlearning and inference—while preserving the privacy of individual datasets.

We consider a semi-honest server that follows the protocol but attempts to extract as much information as possible from client interactions. Specifically, when handling training, unlearning or inference requests, the server tries to infer sensitive information about the provided input features. In addition to data confidentiality, we also aim at protecting the type of query submitted by the client (*e.g.*, training vs. unlearning), which can also be privacy-sensitive. Since a bigger training data set is typically associated with better model utility, a malicious server may have incentives to ignore, reject or even simply defer unlearning requests in order to retain all samples in the model—thus violating the client's right to be forgotten.

To capture this risk, we introduce a stronger adversary model, in which in addition to inferring the client's data, the adversary also aims to distinguish the nature of each query. Our protocol is therefore designed to ensure both data privacy and query indistinguishability, preventing the server from identifying whether a given request corresponds to training, unlearning or inference. This extended adversary model shares a strong similarity with assumptions commonly made in the context of Oblivious RAM (ORAM) [35]. In ORAM, protocols are often designed not only to hide the memory access patterns, but also to make the type of access (*e.g.*, read vs. write) indistinguishable to a semi-honest server [36]. Analogously, in our setting, our objective is to conceal whether a client is performing a read operation on the ERT forest (*i.e.*, inference) or a write/update operation (*i.e.*, training or unlearning). This query indistinguishability is thus essential to prevent the server from inferring intent or denying unlearning requests.

4. Background

Let p be a power of 2. We denote by $\mathbb{Z}_p$ the message space, and by $[\![m]\!]$ the TFHE encryption of a message $m \in \mathbb{Z}_p$. Additional notations are defined as they appear in the text. In this section, we introduce the necessary background on the TFHE cryptosystem, decision trees as well as ERTs to understand our approach.

4.1. The TFHE Cryptosystem

The TFHE encryption scheme, introduced in 2016 [37], [38], is built upon the hardness of the Learning With Errors (LWE) problem and its ring-based variant, Ring-LWE (RLWE).

4.1.1. Ciphertext Types.. The TFHE cryptosystem defines several types of ciphertexts, depending on the plaintext format and the encryption method used. Below, we outline the different ciphertext types and the associated notations used throughout this paper.

LWE Ciphertexts. A message $m \in \mathbb{Z}_p$ can be encrypted as an LWE ciphertext $(\vec{a}, b)$, in which $b = \sum_{i=0}^{n-1} a_i \cdot s_i + \Delta m + e$. Here, $\vec{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_2^n$ is a randomly sampled vector, $\vec{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_2^n$ is the secret key, Δ is the message scaling factor, and e is a noise term drawn from a Gaussian distribution with standard deviation σ_{LWE}.

RLWE Ciphertexts. A tuple of messages $(m_0, \ldots, m_{N-1}) \in \mathbb{Z}_p^N$ can be represented as a polynomial $M(X)$ and encrypted as an RLWE ciphertext $(A(X), B(X))$, in which $B(X) = A(X) \cdot S(X) + \Delta M(X) + E(X)$. Here, $A(X)$ is a polynomial with randomly sampled coefficients, $S(X)$ is the secret key polynomial and $E(X)$ is an error polynomial whose coefficients are drawn from a Gaussian distribution with standard deviation σ_{RLWE}.

LUT Ciphertexts. An additional ciphertext type, known as a Look-Up Table (LUT) ciphertext, was later introduced in the literature. LUT ciphertexts are a specific form of RLWE ciphertexts with structured redundancy: each coefficient of the message polynomial $M(X)$ is repeated $\frac{N}{p}$ times, allowing the encryption of up to p messages within a single ciphertext.

In this paper, we use bracket notation to distinguish between ciphertext types. For example, $[\![M]\!]_{\text{LUT}} = [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}}$ denotes a LUT encryption of message vector $M = (m_0, \ldots, m_{p-1})$; $[\![m]\!]_{\text{LWE}}$ denotes a standard LWE ciphertext of scalar message m and $[m]_{\text{LWE}}$ indicates a trivially encrypted LWE ciphertext (*i.e.*, one with zero noise and zero mask).

4.1.2. TFHE's operations. The TFHE cryptosystem offers several essential primitives for performing homomorphic operations on ciphertexts. The main operations used throughout this paper are described below:

- **Blind Rotation (BR):** $([\![\star]\!]_{\text{LWE}}, [\![\star]\!]_{\text{LUT}}) \rightarrow [\![\star]\!]_{\text{RLWE}}$.
  This operation, central to programmable bootstrapping, enables the homomorphic rotation of a polynomial $M(X)$, encrypted as an RLWE ciphertext, by an amount specified in an LWE ciphertext $[\![i]\!]_{\text{LWE}}$, without revealing this rotation index.
- **Sample Extraction (SE):** $(\star, [\![\star]\!]_{\text{RLWE}}) \rightarrow [\![\star]\!]_{\text{LWE}}$.
  This operation extracts a coefficient from the polynomial $M(X) = \sum_{i=0}^{N-1} m_i X^i$ encrypted as an RLWE ciphertext, to produce an LWE ciphertext $[\![m_j]\!]_{\text{LWE}}$. It operates by isolating and re-encoding one coefficient while discarding the rest.
- **Key Switching (KS):** $[\![\star]\!]_{\text{LWE}} \rightarrow [\![\star]\!]_{\text{LWE}}$.
  This operation transforms an LWE ciphertext encrypted under one secret key into a ciphertext under a different secret key, allowing ciphertext reuse across different contexts or components of the system.
- **Pub. Functional Key Switch (PFKS):** $\{[\![\star]\!]_{\text{LWE}}\} \rightarrow [\![\star]\!]_{\text{RLWE}}$.
  Introduced in [39] (Algorithm 2), this operation allows for the compact representation of multiple LWE ciphertexts into a single RLWE ciphertext, effectively packing several LWE ciphertexts into one.

In our implementation, each Blind Rotation is preceded by a Key Switching operation, while Sample Extraction is omitted due to its negligible execution time and lack of noise accumulation. Among all TFHE primitives, the most computationally significant are the *Blind Rotation* (BR) and the *Public Functional Key Switch (PFKS)*. As such, we quantify the cost of our protocol primarily in terms of these two operations. Specifically, we denote t_{BR} as the combined execution time of a Blind Rotation and its associated Key Switch, and t_{PFKS} as the time required to apply PFKS to a single LWE ciphertext. Sample Extraction is excluded from our analysis for its minimal impact on performance and noise budget.

4.2. RevoLUT operations

RevoLUT [40] is a library built on top of `tfhe-rs` [41] that handles LUT ciphertexts as first-class objects and provides a set of operations specifically designed for their manipulation. Below, we briefly review a subset of these operations that are relevant to this work.

4.2.1. Byte LWE ciphertexts. One of the key challenges in using TFHE lies in the scalability of its cryptographic parameter sizes. While commodity hardware typically processes data in byte-sized units (*i.e.*, $p = 256$), using such a high message modulus directly with `tfhe-rs`'s native encryption would result in impractically large ciphertexts and keys. A practical workaround is to decompose large messages into smaller digits modulo p. In this work, we adopt a radix-2 representation of LWE ciphertexts. Specifically, by setting $p = 2^4$, each ciphertext encodes a 4-bit value or *nibble*. Consequently, a full byte can be encrypted using a pair of LWE ciphertexts. For instance, this representation has been previously leveraged in the design of an 8-bit general-purpose TFHE processor [42].

Definition 1 (Byte-LWE ciphertexts)**.** A message $m \in \mathbb{Z}_{2^8}$, decomposed as $m = 2^4 \cdot m_1 + m_0$ with $m_0, m_1 \in \mathbb{Z}_{2^4}$, is encrypted as:

$$[\![m]\!]_{\text{BLWE}} = ([\![m_1]\!]_{\text{LWE}}, [\![m_0]\!]_{\text{LWE}})$$

This encoding naturally leads to the definition of new LUT structures in the RevoLUT library. In this paper, we focus on a specific LUT variant—$\text{LUT}_{1,2}$—introduced in [40], which we refer to as the Nibble-Byte-LUT.

Definition 2 (Nibble-Byte-LUT ciphertexts)**.** A vector $M = (m_i)_{i=0}^{p-1}$ of p bytes can be encrypted into a $\text{LUT}_{1,2}$ ciphertext as follows :

$$[\![M]\!]_{\text{LUT}_{1,2}} = ([\![M_h]\!]_{\text{LUT}}, [\![M_l]\!]_{\text{LUT}})$$

in which M_h (respectively M_l) is the vector composed of the higher (respectively lower) 4-bits of each $m_i \in M$.

We refer to this structure as a Nibble-Byte-LUT since it allows LUTs to be indexed by a single LWE nibble while the corresponding outputs are full BLWE ciphertexts. In the next section, we present the key operations from the RevoLUT library leveraging this representation.

4.2.2. Oblivious operations. The RevoLUT library offers a suite of oblivious read and write primitives. In particular, it enables the oblivious selection of the $[\![i]\!]_{\mathrm{LWE}}$-th element from a set of LWE ciphertexts without revealing the access index. When the set contains at most p LWE ciphertexts, they can be compactly represented as a single LUT ciphertext via the `PFKS` operation. The BlindRead procedure is then implemented using a combination of `BlindRotation` followed by `SampleExtraction` as detailed in Algorithm 8.

The same approach extends naturally to operations on Nibble-Byte-LUT ciphertexts, such as oblivious reads and writes. For example, the `BlindRead` operation can be applied separately to both components of a $\mathrm{LUT}_{1,2}$ ciphertext—once for the high nibble and once for the low nibble. To simplify notation, we will refer to these generalized operations as if they were applied to standard LUT ciphertexts. However, it is important to note that the number of required Blind Rotation operations (t_{BR}) depends on the structure of the LUT: :

$$t_{\mathrm{BlindRead}} = \begin{cases} t_{BR} & \text{if applied to } [\![M]\!]_{\mathrm{LUT}} \\ 2t_{BR} & \text{if applied to } [\![M]\!]_{\mathrm{LUT}_{1,2}} \end{cases}$$

An `Argmax` operator is also required in our protocol. We have implemented it in the RevoLUT library using a fold-based approach, which allows the computation of both the maximum value and its corresponding index over a vector of BLWE ciphertexts. In this paper, we focus on the specific case in which the input vector contains at most p BLWE ciphertexts, which is described in Algorithm 9. Its computational complexity, expressed in terms of `BlindRotation` and `Public Functional Key Switch` operations, is given by:

$$t_{\mathrm{BlindArgmax}} = (12t_{BR} + 4t_{PFKS}) \times (p-1).$$

Another essential operation provided by RevoLUT is an oblivious counting mechanism developed in [43], which computes the number of encrypted elements in a vector of BLWE or LWE ciphertexts. We refer to this procedure as BlindCount. The time complexity of this operation, in terms of `BlindRotation` and `Public Functional Key Switch` operations, is given by:

$$t_{\mathrm{BlindCount}} = (5t_{BR} + 2t_{PFKS}) \times m,$$

in which m denotes the number of elements in the input vector.

4.3. Decision Trees and Random Forests

Decision trees [44], [45] are a widely used machine learning algorithm that recursively partitions the feature space to produce a prediction. Each internal node $\mathcal{N}$ of a decision tree $\mathcal{T}$ is associated with a specific feature, and in case of numerical features, a threshold θ that is used to guide the decision at that node. Starting from the root node, the tree is traversed by evaluating the conditions at each node, branching accordingly, until a terminal node—referred to as a *leaf*—is reached, which determines the final prediction. Due to their intuitive structure and interpretability, decision trees have been successfully applied to a wide range of tasks, including classification and regression. Their simplicity also makes them computationally efficient and well-suited for scenarios requiring explainable model behavior. Their performance can be significantly enhanced by combining multiple decision trees into an ensemble such as a random forest [46]. This approach aggregates the outputs of multiple trees, typically by majority voting or averaging, leading to improved accuracy and robustness.

Training. The training of a decision tree is an iterative procedure that aims to partition the dataset into increasingly homogeneous subsets with respect to the target variable. At each step in the tree-building process, the current dataset is split into two or more subsets according to a selected feature and, if applicable, a threshold value (*i.e.*, for continuous features). The optimal split is determined by a scoring criterion that seeks to maximize the homogeneity (or minimize the impurity) of the resulting subsets. Common impurity measures include the Gini Index [47] and Information Gain [48]. The process proceeds recursively by creating a new node for each subset and continuing the splitting until a stopping condition is met. Typically, training halts when all samples in a node belong to the same class, though practical implementations often use additional stopping criteria, such as limiting the tree depth or requiring a minimum number of samples per node. Each leaf node maintains class counters, which record the distribution of training samples that reach that leaf. These counters are then used during inference to determine the predicted class label, as well as possibly the confidence level.

Inference. The inference in a decision tree consists in applying the trained model to predict outcomes for previously unseen input data. This process is deterministic and mirrors the tree's structure: starting from the root node, the input feature values are evaluated against the node's threshold (for numerical features) or category (for categorical features). Based on the result of this comparison, the traversal proceeds to the left or right child node, and this process is repeated recursively until a leaf node is reached. In the case of a single decision tree, the predicted class corresponds to the majority class of the training samples stored in that leaf. For an ensemble of trees, such as in a random forest or ERTs, the final prediction is typically determined by aggregating the outputs of individual trees—most commonly via majority voting in classification tasks or averaging in regression.

Unlearning. Driven by privacy regulations and the increasing demand for user control over personal data, significant research efforts have focused on developing effective machine unlearning methods [49]. More precisely, the goal of unlearning is to remove the influence of a specific data sample from a trained model, effectively "forgetting" that the sample was ever part of the training set. A naive approach to unlearning would be to retrain the model from scratch on the original dataset minus the target sample. However, this makes the computational cost of unlearning nearly equivalent to full retraining, which is often prohibitive. Consequently, unlearning techniques aim at being more computationally efficient compared to retraining. One of the most well-known approaches is SISA (Sharded, Isolated, Sliced, and Aggregated training) [32], a framework that supports unlearning for arbitrary model ensembles. In SISA, the training data is partitioned into independent slices and models are trained in isolation on these slices. To unlearn a sample, only the relevant model trained on the affected slice needs to be retrained. While this reduces the cost compared to retraining the entire ensemble, it still requires some retraining, albeit on a smaller portion of the data.

4.4. Extremely Randomized Trees

Introduced by Geurts, Ernst and Wehenkel in 2006 [4], ERTs offer an efficient and highly randomized variant of tree-based ensemble learning. The key idea behind ERTs is to inject randomness at two levels of the tree-building process: both the choice of features and the associated thresholds are randomized. Specifically, for each node, a subset of k' features is randomly selected from the total set of k features. The best feature is then chosen among this subset based on a scoring criterion. When $k' = 1$, the resulting tree is entirely random, with both the selected features and thresholds completely decoupled from the training data.

ERT training involves constructing a predefined number of trees, each of a fixed depth d. For each tree, the internal nodes are randomly assigned features and thresholds before any data is processed. The training data is then passed through the trees, updating class counters in the corresponding leaves. During inference, the model aggregates predictions from all trees—typically using majority voting—to obtain the final output.

Despite their simplicity and lack of fine-tuned splits, ERTs have demonstrated strong empirical performance across a range of tasks. For example, they have been used to achieve up to 99.27% accuracy in breast cancer prediction by integrating feature ranking with ERTs [50], and have outperformed other learning algorithms such as neural networks, random forests and support vector machines in fault detection for wireless sensor networks [51]. An additional advantage of ERTs, particularly relevant in privacy-sensitive applications, is their natural support for machine unlearning. Indeed, unlike standard decision trees—in which removing a training sample might change the optimal split at each node—ERTs construct their structure independently of the data. Thus, instead of retraining or modifying the tree structure, one can simply reprocess the target sample to locate the corresponding leaf and decrement the associated class counter, which makes ERTs especially well-suited for cryptographic protocols that require both learning and unlearning over encrypted data.

5. Our Proposal

In this section, we present the main contribution of this work: a novel protocol for training, untraining and inference of ERTs, along with its practical implementation under the TFHE cryptosystem. First, we describe the client-side pre-processing that facilitates several key operations in our protocol, before detailing the full construction.

5.1. Client Processing

Client-side preprocessing plays a crucial role in both the efficiency and accuracy of our protocol. In this section, we describe the operations performed by the client, including TFHE parameter selection, data encoding and quantization, along with the rationale behind these design choices.

TFHE parameters choice. As discussed earlier, TFHE natively supports encrypted messages with precision limited to $\log_2(p)$ bits. While the `tfhe-rs` library [41] can support values up to $p = 2^8$ by adjusting parameters without compromising security, increasing p significantly degrades performance. Beyond $p = 2^6$, key sizes become prohibitively large and operation runtimes slow down considerably. We selected $p = 2^4$ as a balanced choice, for the following reasons: first, it enables modeling trees as a hierarchy of LUTs, which is central to our efficient traversal protocol described in the next section. Second, it supports the use of Byte-LWE ciphertexts, allowing efficient manipulation of encrypted bytes via two 4-bit ciphertexts. Third, in `tfhe-rs`, the `BlindRotation` operation—heavily used in our protocol—is most optimized for $p = 2^4$; and finally, the resulting key size remains manageable (approximately 65 MB), making the protocol more practical for deployment. The parameter set used in our implementation is summarized in Table 1.

Table 1: The parameters used in our experiments giving 4bits of precision and ensuring 128-bit of security. The TFHE parameters notations used are the ones in TFHE's original paper [39].

Parameter	Value
LWE dimension	761
RLWE polynomial degree (N)	2048
LWE standard deviation (σ_{LWE})	2^{-40}
RLWE standard deviation (σ_{RLWE})	2^{-2}
Decomp params bootstrapping (g, ℓ)	$(2^{23}, 1)$
Decomp params KS (g, ℓ)	$(2^3, 5)$
Decomp params PFKS (g, ℓ)	$(2^{23}, 1)$
Ciphertext modulus (q)	2^{64}
Plaintext modulus (p)	2^4

Data encoding. A key factor contributing to the protocol's efficiency lies in how the client encodes its input features. Let $s = ([f_0, \ldots, f_{k-1}], l)$ denote a data sample, in which f_i is the i-th feature and l the class label. Each feature f_i is represented as a unary vector F_i of size N defined as:

$$F_{i,j} = \begin{cases} 0 & \text{if } j \in [0, f_i - 1] \\ 1 & \text{if } j \in [f_i, N-1]. \end{cases}$$

For instance, with $N = 4$ and $f_i = 3$, this yields $F_i = [0, 0, 0, 1]$. Each vector F_i is then encrypted as an RLWE ciphertext, producing an encrypted feature vector $F = (\llbracket F_0 \rrbracket_{\mathrm{RLWE}}, \ldots, \llbracket F_{k-1} \rrbracket_{\mathrm{RLWE}})$. This encoding, inspired by [22], allows the server to evaluate comparisons without explicitly performing them. If a decision node specifies feature index I and threshold θ, the server retrieves $\llbracket F_I \rrbracket_{\mathrm{RLWE}}$ and performs a Sample Extraction at index θ to obtain $\llbracket \theta < f_I \rrbracket_{\mathrm{LWE}}$. Since Sample Extraction is one of the fastest operations in TFHE, this method greatly accelerates comparison logic. Repeating this process across all nodes yields a "comparison tree" or *compiled tree*, described in the next section. For the label $l \in \mathbb{Z}_\gamma$, we use a one-hot encoding $l = (l_0, \ldots, l_{\gamma-1})$. The label vector is then adapted according to the query type—training, unlearning or inference—as follows:

$$l_i = \begin{cases} 1, & \text{if Training and } l_i \neq 0 \\ 2, & \text{if Unlearning and } l_i \neq 0 \\ 0, & \text{if Inference and } l_i \neq 0 \end{cases} \tag{1}$$

Although -1 could have represented unlearning, we use 2 for efficiency in the BAAT operation (see Algorithm 11) used for leaf updates. Finally, each l_i is encrypted as an LWE ciphertext, yielding the encrypted label vector $L = (\llbracket l_0 \rrbracket_{\mathrm{LWE}}, \ldots, \llbracket l_{\gamma-1} \rrbracket_{\mathrm{LWE}})$. In summary, the client encrypts a sample as $S = (F, L)$.

Quantization. The quantization process is a crucial step in our protocol, as it allows us to represent continuous feature values with a limited number of bits. To quantize the dataset, we first normalize each feature value to $[0, 1]$, based on the minimum and maximum values of the feature, before scaling this normalized value to the range of integers that can be represented with the specified number of bits. The result is then rounded to the nearest integer. Since the feature values are encoded in a unary vector, the precision of the data is now limited to $log_2(N)$ bits. In particular, the classical TFHE parameters with $p = 2^4$ that provides 128-bit of security in `tfhe-rs` sets N to 2^{11}. Therefore, we can quantize the feature values to 11 bits. This process ensures that the quantized values are uniformly distributed across the available range, preserving the relative differences between feature values while reducing the data size. This is particularly important for our protocol, as the client may encode just one or few samples (*e.g*, for inference).

5.2. Building Blocks

This section introduces the main components upon which our protocol is built.

5.2.1. Trees Structure. We adopt a specialized representation of decision trees tailored to the requirements of our protocol. As introduced earlier in the context of client-side encoding, one such representation is the *compiled tree*. In addition, we define two other variants: the *training tree* and the *inference tree*. All tree variants share a common structure, which can be expressed as:

$$T = \left\{\mathcal{N}_i\right\}_{i=0}^{2^{d+1}-1} \cup \left\{\mathcal{L}_i\right\}_{i=0}^{2^d-1}$$

in which $\mathcal{N}_i$ denotes a decision node and $\mathcal{L}_i$ a leaf node. The main difference between each tree type is the internal representation of these nodes and their associated logic. To provide an intuitive understanding, we illustrate the three tree types in Figure 2, and proceed to formally define them in the following subsections.

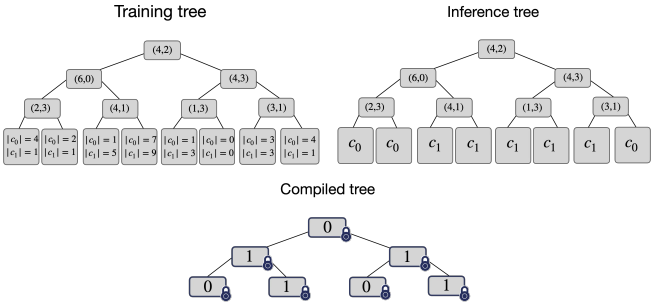


Fig. 2. Illustration of the training tree, inference tree and compiled tree.

Definition 3 (Training tree)**.** In a training tree $\mathcal{T}^{tr}$, each decision node $\mathcal{N}_i$ is stored in the clear and contains a pair (θ_i, I_i), in which θ_i is a threshold and I_i is the index of the feature. Each leaf $\mathcal{L}_i$ holds γ BLWE ciphertexts, in which the j-th ciphertext encrypts the count of training samples that reached the leaf and belong to class j. Formally, we have:

$$\mathcal{N}_i = (\theta_i, I_i) \quad \text{and} \quad \mathcal{L}_i = (\llbracket c_0 \rrbracket_{\mathrm{BLWE}}, \ldots, \llbracket c_{\gamma-1} \rrbracket_{\mathrm{BLWE}}),$$

in which $\llbracket c_j \rrbracket_{\mathrm{BLWE}}$ represents the encrypted count of class-j samples at leaf $\mathcal{L}_i$.

This representation enables the leaf nodes to store class counts up to a maximum value of 256, due to the byte-size encryption. If more than 256 samples from the same class reach a leaf, the counter wraps around modulo 256. The implications of this constraint on model accuracy are discussed in Section 6.2.

After the training phase, a new structure called the *inference tree* is derived from the training tree, by computing the majority class in each leaf $\mathcal{L}_i$ of the training tree, which then serves as the prediction for that leaf. The majority vote is carried out using the $\mathrm{TreeMajority}$ algorithm, presented in Algorithm 1.

Definition 4 (Inference tree)**.** In an inference tree $\mathcal{T}^{inf}$, the decision nodes $\mathcal{N}_i$ are identical to those in the training tree. However, each leaf $\mathcal{L}_i$ now contains a single LWE

ciphertext that encrypts the majority class label. Formally, we have:

$$\mathcal{N}_i = (\theta_i, I_i) \quad \text{and} \quad \mathcal{L}_i = [\![c]\!]_{\text{LWE}}$$

, in which c is the class with the highest count in the corresponding leaf of the training tree, serving as the prediction for that leaf.

Once the inference trees are constructed, the server uses them to process incoming inference queries. The predictions from each tree are then aggregated using a majority vote to produce the final output. This voting procedure is carried out by the Majority algorithm, detailed in Algorithm 3.

Algorithm 1: Majority vote in a leaf (TreeMajority)

Input : A training tree $\mathcal{T}^{tr}$ with 2^d leaves $\mathcal{L}_i = ([\![c_0]\!]_{\text{BLWE}}, \ldots, [\![c_{\gamma-1}]\!]_{\text{BLWE}})$ in which d is the depth of the tree and γ is the number of classes.

Output: An inference tree $\mathcal{T}^{inf}$ with 2^d leaves $\mathcal{L}_i = [\![c]\!]_{\text{LWE}}$ in which c is the majority class.

```
// Get the majority class for each leaf
for i ∈ [0, 2^d − 1] do
    [[c]]_LWE ← BlindArgmax(L_i)
    T^inf.L_i ← [[c]]_LWE
end
return T^inf
```

The TreeMajority algorithm operates by invoking the BlindArgmax procedure from RevoLUT on each leaf of the training tree $\mathcal{T}^{tr}$. Its overall time complexity is therefore given by:

$$\begin{aligned} t_{\text{TreeMajority}} &= 2^d \times t_{\text{BlindArgmax}} \\ &= 12 \times 2^d \times t_{BR} + 4 \times 2^d \times t_{PFKS} \end{aligned}$$

in which d is the depth of the training tree $\mathcal{T}^{tr}$.

Definition 5 (Compiled tree)**.** A compiled tree $\mathcal{T}^{comp}$ is an encrypted structure used to traverse the decision tree obliviously. Each decision node $\mathcal{N}_i$ contains a single encrypted comparison bit $[\![b_i]\!]_{\text{LWE}}$ computed via the Compile procedure (see Algorithm 2). The leaves $\mathcal{L}_i$ are left empty, as the sole purpose of the compiled tree is to determine the encrypted index of the reached leaf. Formally, we define:

$$\mathcal{N}_i = ([\![b_i]\!]_{\text{LWE}}) \quad \text{and} \quad \mathcal{L}_i = \varnothing$$

in which $[\![b_i]\!]_{\text{LWE}}$ denotes the encrypted result of the comparison at node i.

Since the Compile operation relies solely on the SampleExtraction primitive—which is fast and incurs no additional noise—its time complexity t_{Compile} is negligible compared to other operations used in our protocol.

Algorithm 2: Compilation process (Compile)

Input : A balanced decision tree $\mathcal{T} = \{Level_0, \ldots, Level_{d-1}\}$ in which $Level_i$ is the i-th level of the tree. The i-th level is composed of 2^i nodes (θ, I) in which θ is the threshold and I is the index of the feature.

Output: A compiled tree $\mathcal{T}^{comp}$.

```
T^comp ← ∅
for i ∈ [0, d − 1] do
    // Compile the i-th level
    Level_c ← ∅
    for (θ, I) ∈ Level_i do
        b ← SampleExtraction(θ, [[F_I]]_RLWE)
        Level_c ← Level_c ∪ b
    end
    T^comp ← T^comp ∪ Level_c
end
return T^comp
```

Algorithm 3: Majority vote of a set of LWE ciphertexts (Majority)

Input : A set of m LWE ciphertexts $C = \{[\![c_0]\!]_{\text{LWE}}, \ldots, [\![c_{m-1}]\!]_{\text{LWE}}\}$ representing the classification of a sample by m inference trees.

Output: A majority class c

```
// Get the count of each class
R ← BlindCount(C)
// Get the majority class
c ← BlindArgmax(R)
return c
```

The Majority algorithm, which consists essentially of two RevoLUT operations—BlindCount followed by BlindArgmax—has a total time complexity given by:

$$\begin{aligned} t_{\text{Majority}} &= t_{\text{BlindCount}} + t_{\text{BlindArgmax}} \\ &= (5m + 12(p-1)) \times t_{BR} \\ &\quad + (2m + 4(p-1)) \times t_{PFKS} \end{aligned}$$

where m is the number of inference trees and $p = 16$.

5.2.2. Shallow Tree Traversal. Once the compiled tree is constructed, we traverse it level by level using successive BlindRead operations on LUT ciphertexts that represent each level. We restrict ourselves to perfectly balanced trees of depth $d \leq \log_2(p)$, so that the i-th level $N_{j\,j=0}^{2^i-1}$ can be packed into a single LUT ciphertext. During traversal, an encrypted selector s_i is computed iteratively to track the path through the tree. This selector is an LWE ciphertext defined by the recurrence:

$$s_i = \begin{cases} [\![b_0]\!]_{\text{LWE}} & \text{if } i = 0 \\ [\![b_i]\!]_{\text{LWE}} + 2s_{i-1} & \text{if } i \geq 1 \end{cases}$$

Here, $[\![b_i]\!]_{\text{LWE}}$ is the encrypted comparison bit obtained via BlindRead on the i-th level. Recall that in a compiled tree $\mathcal{T}^{comp}$, the j-th node contains $[\![b_j]\!]_{\text{LWE}} = [\![\theta < f_I]\!]_{\text{LWE}}$, the result of comparing the encrypted feature to the threshold. At the end of the process, s_d is an LWE ciphertext encrypting the index ℓ of the reached leaf.

This traversal method is detailed in Algorithm 4, and it can be generalized to any balanced tree of depth $d \leq \log_2(p)$, even when comparisons are computed homomorphically instead of using precompiled values.

Algorithm 4: Traversal process (Traverse)

Input : A compiled tree $\mathcal{T}^{comp} = \{Level_0, \ldots, Level_{d-1}\}$ in which $Level_i$ is the i-th level of the tree containing 2^i LWE ciphertexts $[\![b_i]\!]_{\text{LWE}}$ representing the comparison bits.

Output: The selector s of the reached leaf.

```
// Initialize the selector to the first level
```
$s \leftarrow Level_0.b_0$

```
// Traverse the tree
```
for $i \in [1, d-1]$ **do**
  $[\![Level_i]\!]_{\text{LUT}} \leftarrow \text{PFKS}(Level_i)$
  $b \leftarrow \text{BlindRead}([\![Level_i]\!]_{\text{LUT}}, s)$
  $s \leftarrow b + 2s$
end
return s

Since traversal uses one BlindRead per level and one PFKS per node (except at the root), the total complexity is:

$$\begin{aligned} t_{\text{Traverse}} &= d \times t_{\text{BlindRead}} + \Big(\sum_{i=1}^{d-1} 2^i\Big) \times t_{\text{PFKS}} \\ &= d \times t_{BR} + 2^d \times t_{\text{PFKS}} \end{aligned}$$

in which d is the depth of the compiled tree $\mathcal{T}^{comp}$.

5.2.3. Updating The Leaves. During training (respectively unlearning), the leaf nodes of a tree $\mathcal{T}$ are updated by incrementing (respectively decrementing) the class counts they contain. More precisely, at a given iteration, a leaf $\mathcal{L}_i$ holds encrypted class counters $[\![c_0]\!]_{\text{BLWE}}, \ldots, [\![c_{\gamma-1}]\!]_{\text{BLWE}}$, in which each $[\![c_j]\!]_{\text{BLWE}}$ encodes the number of samples of class j that have reached that leaf. Given a sample (F, L) and its associated encrypted label vector L, the update rule is:

$$[\![c_j]\!]_{\text{BLWE}} = \begin{cases} [\![c_j]\!]_{\text{BLWE}} + 1 & \text{if Training and } L_j \neq 0 \\ [\![c_j]\!]_{\text{BLWE}} - 1 & \text{if Unlearning and } L_j \neq 0 \\ [\![c_j]\!]_{\text{BLWE}} & \text{if Inference} \end{cases}$$

To preserve the obliviousness of the protocol, this update must not reveal the label nor the nature of the query (training vs. unlearning). This is handled by the Update procedure, which uses the BAAT primitive (Blind Add Ternary) from RevoLUT [40], detailed in Algorithm 11. The objective BAAT is to add an encrypted ternary value (*i.e*, 0 , 1 or -1) to a Nibble-Byte-LUT at an encrypted index ℓ computed via the Traverse algorithm.

Algorithm 5: Updating the leaves (Update)

Input : A training tree $\mathcal{T}$ with 2^d leaves $\mathcal{L}_i = ([\![c_0]\!]_{\text{BLWE}}, \ldots, [\![c_{\gamma-1}]\!]_{\text{BLWE}})$ in which d is the depth of the tree. A selector $[\![\ell]\!]_{\text{LWE}}$ encrypted as a LWE ciphertext denoting the index of the leaf reached. A label L encrypted as a vector of LWE ciphertexts.

Output: The updated training tree $\mathcal{T}$.

```
// Pack each class counts into a Nibble-Byte LUT
```
for $j \in [0, \gamma-1]$ **do**
  $v \leftarrow \varnothing$
  for $i \in [0, 2^d - 1]$ **do**
    $v \leftarrow v \cup [\![c_j]\!]_{\text{BLWE}}$
  end
  $[\![C_j]\!]_{\text{LUT}_{1,2}} \leftarrow \text{PFKS}(v)$
end

```
// Update the leaves
```
for $j \in [0, \gamma-1]$ **do**
  $[\![C_j]\!]_{\text{LUT}_{1,2}} \leftarrow \text{BAAT}([\![C_j]\!]_{\text{LUT}_{1,2}}, [\![\ell]\!]_{\text{LWE}}, L_j)$
end
return $\mathcal{T}$

In our implementation, the packing step is omitted by storing the class counters as prepacked Nibble-Byte-LUT ciphertexts. While the previous notation $\mathcal{L}_i = ([\![c_0]\!]_{\text{BLWE}}, \ldots, [\![c_{\gamma-1}]\!]_{\text{BLWE}})$ is kept for clarity, the actual data structure is γ precomputed Nibble-Byte-LUTs. As a result, the time complexity of the Update operation—excluding the packing step—is:

$$\begin{aligned} t_{\text{Update}} &= \gamma \times t_{\text{BAAT}} \\ &= 15\gamma \times t_{BR} + 11\gamma \times t_{PFKS}, \end{aligned}$$

in which γ denotes the number of classes.

5.3. Partially Oblivious Protocol

In this section, we detail the core components of our protocol, including the procedures for training, unlearning and inference. We also analyze the extent to which each operation preserves obliviousness and discuss the trade-offs between efficiency and privacy in our design.

5.3.1. Training And Unlearning Processes. The training and unlearning processes are described in Algorithm 6. They take as input a dataset $\mathcal{D}$ and either generates a new forest $\mathcal{F}$ composed of ERTs or operates on a pre-existing forest—typically the case during unlearning. For each sample $(F, L) \in \mathcal{D}$, each tree $\mathcal{T} \in \mathcal{F}$ is first compiled using the Compile function, which computes the encrypted comparison bits. Then, the Traverse function is used to

derive the encrypted index ℓ of the reached leaf. Finally, the Update procedure adjusts the class counters in the leaf accordingly, depending on the nature of the query (*i.e.*, training or unlearning).

After all samples have been processed, the leaves of each tree in the forest undergo a majority vote using the TreeMajority algorithm to prepare the trees for inference. This overall process is illustrated in Figure 3.

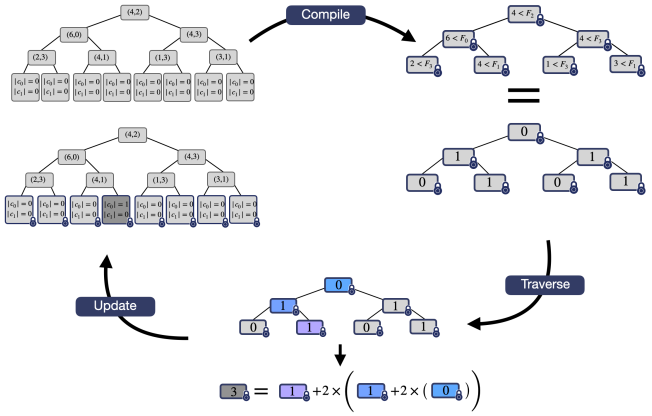


Fig. 3. Overview of the Training Process at the beginning of the training, in which leaves are set to $\{[\![0]\!]_{\text{BLWE}}\}_{j=0}^{\gamma-1}$ for $\gamma = 2$. The feature vector used for updating the leaves is $f = [5, 3, 7, 1, 0]$ and the label is $l = 0$ encoded and encrypted as explained in Section 5.1.

Algorithm 6: Training/Unlearning process

Input : A dataset $\mathcal{D} = \left[\left(F^i, L^i\right)\right]_{i=1}^{n}$ in which $F^i = ([\![F_0^i]\!]_{\text{RLWE}}, \ldots, [\![F_{k-1}^i]\!]_{\text{RLWE}})$ is the the encrypted features of the i-th sample and L^i is its one-hot encoded label encrypted as a vector of LWE ciphertexts.
Output: A forest $\mathcal{F} = (\mathcal{T}_0, \ldots, \mathcal{T}_{m-1})$

// Generate random trees
for $i \in [0, m-1]$ **do**
  $\mathcal{T}_i \leftarrow \text{RandomTree}()$
end
// Train/Unlearn the trees
for $(F, L) \in \mathcal{D}$ **do**
  for $\mathcal{T} \in \mathcal{F}$ **do**
    $\mathcal{C} \leftarrow \text{Compile}(\mathcal{T}, F)$
    $\ell \leftarrow \text{Traverse}(\mathcal{C})$
    $\mathcal{T} \leftarrow \text{Update}(\mathcal{T}.leaves, L, \ell)$
  end
end
// Compute the forest ready for inference
for $\mathcal{T} \in \mathcal{F}$ **do**
  $\mathcal{T} \leftarrow \text{TreeMajority}(\mathcal{T})$
end
return $\mathcal{F}$

Remark that, due to the randomized nature of ERTs, some leaves may remain unvisited during training. In such cases, the TreeMajority function performs an argmax over all-zero counts, defaulting to the last class, which can lead to erroneous predictions. To mitigate this, we append an additional encrypted zero-counter as an *abstention class* before executing TreeMajority. This ensures that if a leaf is unvisited, the model abstains from making a prediction rather than guessing arbitrarily.

The total cost of the training or unlearning phase, expressed in terms of cryptographic operations, is given by:

$$t_{tr} = n \cdot m \cdot (t_{\text{Compile}} + t_{\text{Traverse}} + t_{\text{Update}}) + m \cdot t_{\text{TreeMajority}}, \tag{2}$$

in which n is the number of data samples and m is the number of trees in the forest.

5.3.2. Inference. Once the forest $\mathcal{F}$ has been prepared for inference, the prediction process becomes relatively straightforward. For each tree $\mathcal{T} \in \mathcal{F}$, the client sample F is first compiled using the Compile function. The compiled tree is then traversed via the Traverse function, yielding an encrypted index ℓ corresponding to the reached leaf. The prediction associated with this leaf is then retrieved using the BlindRead function. After collecting predictions from all trees in the forest, a final majority vote is performed using the Majority function to determine the predicted class. The detailed procedure is provided in Algorithm 7.

To provide a complete view, the total computational cost of the inference procedure is expressed as:

$$t_{inf} = n \cdot m \cdot (t_{\text{Compile}} + t_{\text{Traverse}} + t_{\text{BlindRead}}) + n \cdot t_{\text{Majority}} \tag{3}$$

Algorithm 7: Inference process

Input : A feature vector $F = ([\![F_0]\!]_{\text{RLWE}}, \ldots, [\![F_{k-1}]\!]_{\text{RLWE}})$. A vector of encyption of 0 serving as label $L = ([\![0]\!]_{\text{LWE}})_{i=0}^{\gamma-1}$. A forest $\mathcal{F} = (\mathcal{T}_0, \ldots, \mathcal{T}_{m-1})$ with majority vote in the leaves.
Output: A classification c

// Get the classification of each tree
$R \leftarrow \varnothing$
for $(F, L) \in \mathcal{D}$ **do**
  for $\mathcal{T} \in \mathcal{F}$ **do**
    $C \leftarrow \text{Compile}(\mathcal{T}, F)$
    $\ell \leftarrow \text{Traverse}(\mathcal{C})$
    $[\![r]\!]_{\text{LWE}} \leftarrow \text{BlindRead}(\mathcal{T}.leaves, \ell)$
    $R \leftarrow R \cup [\![r]\!]_{\text{LWE}}$
  end
end
// Get the majority vote
$c \leftarrow \text{Majority}(R)$
return c

5.3.3. Discussion. The oblivious nature of the protocol stems from the shared use of the Compile and Traverse functions across all query types—training, unlearning and

inference. The only divergence occurs after the traversal, in which the Update function is invoked for training and unlearning to modify the leaves while $\mathrm{BlindRead}$ is used during inference to retrieve a prediction. This architectural separation necessitates distinct tree representations for training/unlearning and for inference. As such, we refer to this construction as a *Partially Oblivious Protocol*. It guarantees that training and unlearning queries are indistinguishable from each other, preserving the client's right to be forgotten. However, inference queries can still be differentiated by the server. Consequently, there is no need for the client to include a label L when requesting inference, as shown in Algorithm 7 and supported by the encoding logic in Equation 1. In the following section, we demonstrate how this protocol can be extended to achieve full query-type obliviousness through a few additional operations.

5.4. Fully Oblivious Protocol

To make the client's queries—whether for training, unlearning, or inference—indistinguishable to the server, we explore three progressively more efficient strategies to achieve full query-type obliviousness.

First and foremost, the client encodes the labels according to Equation 1. The first and most straightforward method consists in having the server execute both the training/unlearning and inference processes for each incoming sample. This guarantees perfect query indistinguishability but comes at the cost of significant overhead with a total runtime that becomes:

$$t_{tr} + t_{inf} + (n-1) \cdot m \cdot t_{\mathrm{TreeMajority}}$$

To reduce this overhead, we consider a second approach: augmenting Algorithm 6 with inference computations. For every sample, the server computes the inference trees by applying $\mathrm{TreeMajority}$ to each tree in $\mathcal{F}$, followed by a $\mathrm{BlindRead}$ to retrieve the prediction. If the query corresponds to inference, the tree structure remains unchanged since the label vector L consists of $[\![0]\!]_{\mathrm{LWE}}$. The final prediction is computed via a majority vote ($\mathrm{Majority}$). Although this method better aligns with the ERT design, it still incurs considerable cost due to repeated and sometimes unnecessary $\mathrm{TreeMajority}$ and $\mathrm{Majority}$ operations. Its total runtime is:

$$t_{tr} + n \cdot m \cdot t_{\mathrm{BlindRead}} + n \cdot t_{\mathrm{Majority}} + (n-1) \cdot m \cdot t_{\mathrm{TreeMajority}}$$

A more efficient alternative leverages batch processing. Here, the forest $\mathcal{F}$ comprises both training trees $\mathcal{T}^{tr}$ and inference trees $\mathcal{T}^{inf}$, the latter being derived from the former via periodic application of $\mathrm{TreeMajority}$. Every β queries, the inference trees are updated. During the interval, all samples—regardless of query type—are processed using the current inference trees and updates are performed only on training trees. This way, inference queries do not alter the model (as their labels are zero vectors), but predictions are still correctly computed. Training and unlearning queries continue to update the training trees. The detailed process is given in Algorithm 12 in Appendix 5.4 and its resulting runtime is:

$$t_{tr} + n \cdot m \cdot t_{\mathrm{BlindRead}} + n \cdot t_{\mathrm{Majority}} + \frac{n}{\beta} \cdot m \cdot t_{\mathrm{TreeMajority}}$$

Hence, the server benefits from this third method when $\beta > n-1$, as it significantly amortizes the cost of $\mathrm{TreeMajority}$ over multiple samples.

6. Experimental Evaluation

In this section, we describe the experiments conducted to validate our approach. All experiments were conducted on a machine running Ubuntu 24.04, equipped with an Intel i9-11900KF CPU at 3.5 GHz and 64 GB of RAM. We evaluated our protocol on three standard datasets from the UCI Machine Learning Repository: Iris [52], Wine [53] and Breast Cancer [54]. The main characteristics of these datasets are summarized in Table 2.

Dataset ($\mathcal{D}$)	**# samples** (n)		**# features** (k)	**# classes** (γ)
	Train	**Test**		
Iris	120	30	4	3
Wine	142	36	13	3
Breast Cancer	455	114	30	2

Table 2: Datasets used in the experiments.

These three datasets are particularly relevant as they each exhibit distinct characteristics that impact different aspects of the protocol, such as runtime, accuracy and bandwidth. For example, while both the Iris and Wine datasets contain a small number of samples, they feature a relatively high number of classes. In contrast, the Breast Cancer dataset has a larger sample size but fewer classes. Additionally, the Wine dataset differs from the Iris dataset by having a greater number of features. These variations provide a diverse testbed, allowing us to assess the performance of our protocol under different conditions.

6.1. Time complexity

The use of ERTs significantly accelerates the training process, as the construction of the internal nodes $\mathcal{N}_i$ is instantaneous—contrary to classical decision trees, which require data-dependent computations. The total training times for Algorithm 6, applied to the three datasets summarized in Table 2, are reported in Figure 5. As illustrated, training time is not solely dictated by the number of samples but also by the number of output classes. For a fixed number of trees, the number of samples directly scales the cost of the $\mathrm{Compile}$, $\mathrm{Traverse}$ and Update operations, while the number of classes primarily impacts the complexity of $\mathrm{TreeMajority}$, Update and $\mathrm{Majority}$ (see Table 5 in Appendix C). For instance, given an equal number of samples, training on the Breast Cancer dataset—which has fewer classes—is expected to be approximately 1.5× less expensive than on the Iris or Wine datasets. Nevertheless, the results in Figure 5 show that the Cancer dataset incurs the

highest training cost. This is explained by its significantly larger sample size—roughly three times that of the Iris and Wine datasets—making sample count the dominant factor in total training time for this particular case.

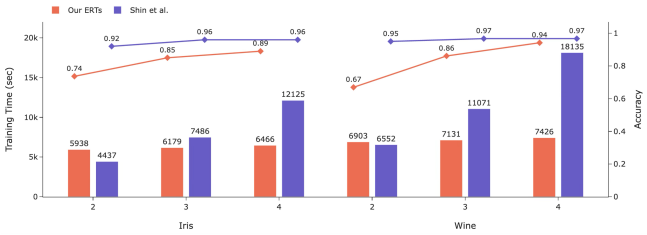


Fig. 4. A comparison of training times and accuracy between our private ERT forest and the private Random Forest model from [29], evaluated across different tree depths. For a fair comparison, we took the same amount of samples for both models, which correspond to those reported in Table 3 of [29]. Both models have a forest size of 64.

Regarding the comparison with the privacy-preserving Random Forest model based on the CKKS scheme, as proposed by [29], the runtime figures reported in their paper (shown in Figure 4) indicate that our protocol achieves a speedup of approximately $1.2\times$ to $2.4\times$ for tree depths starting from 3, despite a marginal trade-off in accuracy. It is important to emphasize that, unlike [29], our reported training time explicitly includes the cost of the $\mathrm{TreeMajority}$ operation, which is used to finalize predictions by computing the majority vote in the leaves. This further underscores the practical efficiency of our approach, particularly in scenarios where low latency is critical.

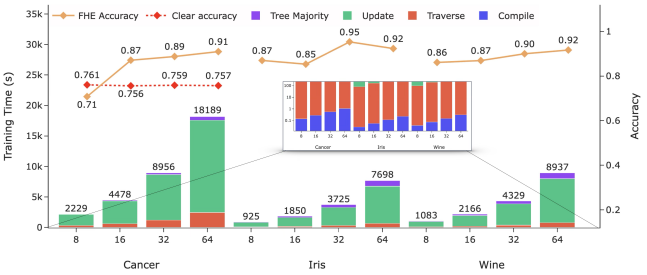


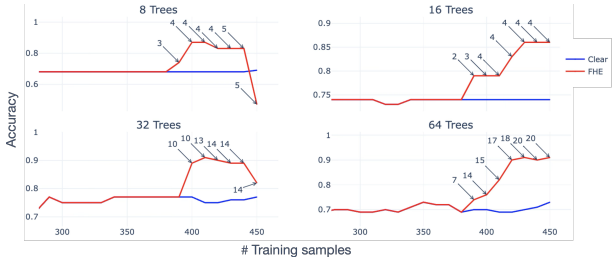
Fig. 5. Training time and accuracies across various forest sizes $\mathcal{F}$ (*i.e*, $8, 16, 32$ and 64) and the different datasets used. The results presented were obtained with the training set sizes listed in Table 2

6.2. Accuracy analysis

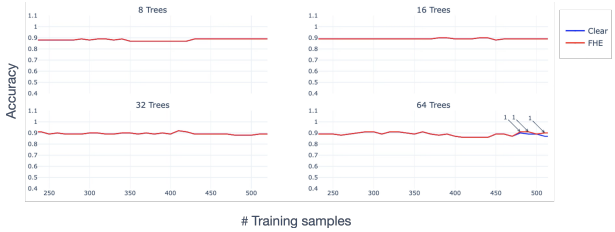
The accuracy of ERTs is inherently variable due to the randomized nature of feature and threshold selection. This work does not aim to improve ERT accuracy but instead to analyze how applying ERTs to encrypted data affects predictive performance. To do so, we first identified the optimal splits (θ, I) on unencrypted data by averaging results over 100 trials. We then retrained the model using the same data but encrypted. Our results show that the accuracy of encrypted models depends not only on the dataset but also on the forest size and the encryption constraints. As depicted in Figure 5, the Iris dataset yielded the highest accuracy, which is expected due to its small size (few samples and features), which facilitates generalization and prevents overfitting. In several runs, the encrypted model achieved perfect accuracy (100%). The Wine dataset, despite having more features, performed similarly, demonstrating the protocol's resilience to increased input dimensionality. For these two datasets, the number of training samples per class was small enough to avoid exceeding the capacity of a BLWE ciphertext (limited to a maximum count of 256). As a result, the encrypted and unencrypted models exhibited comparable accuracy.

In contrast, the Cancer dataset includes more samples than can be stored in an encrypted leaf counter, which is limited to 256 by the BLWE ciphertext capacity. This leads to a mismatch between the encrypted and unencrypted model accuracies when more than 256 samples of the same class reach the same leaf. A straightforward way to address this issue is to increase the radix representation of the counters by using more LWE components in each BLWE ciphertext, thereby extending the counting range. However, this comes at the cost of reduced performance. Interestingly, this overflow effect can sometimes improve accuracy. As shown in Figure 6a, the encrypted model's accuracy on the Cancer dataset initially aligns with the clear-text version and even improves slightly after overflows begin. This behavior stems from a form of implicit pruning: when excessive samples from the majority class (typically c_0) flood a poorly split leaf (often the rightmost one), the counter wraps around, possibly flipping the predicted class to c_1. Such "corrupted" trees—whose splits are too coarse to be useful—are effectively neutralized or reversed in their decision. In binary classification tasks, this can increase the diversity of the ensemble and correct for biased trees, improving the overall majority vote. However, this advantage is fragile. If more than half of the trees in the forest are corrupted due to overflow, the ensemble's decision may become unreliable, and accuracy drops below the baseline of the clear-text model. As shown in Figure 6a, this threshold is critical for forests of limited size (*e.g.*, 8 trees). Hence, increasing the forest size can help mitigate the risks of overflow by diluting the influence of any single corrupted tree and maintaining better ensemble stability.

We hypothesize that if the dataset exhibits high class overlap —meaning that samples from different classes are not well-separated in feature space— this overflow phenomenon tends to be less frequent. For example, the Japanese Credit Screening dataset [55], despite containing more samples than the Cancer dataset, shows fewer overflows. This is likely due to the lower separability of its classes, as evidenced by the Principal Component Analysis (PCA) plots in Figure 7b, in which class boundaries are less distinct. Consequently, the FHE accuracy closely mirrors the clear-text accuracy across various forest sizes (see Figure 6b).



(a) Cancer dataset



(b) Japanese Credit Screening dataset

Fig. 6. Accuracy of various forests during the training phase on the Cancer and Japanese Credit Screening datasets [55]. The arrows $(\rightarrow)$ indicate the count of "corrupted" trees in which leaf overflow occurred, meaning more than 256 samples of the same class reached a leaf.

6.3. Bandwidth analysis

Our protocol involves two distinct phases from a bandwidth perspective:

- *Offline Phase*: The client generates and transmits the necessary cryptographic material to the server. This includes three types of keys: the bootstrapping key (BSK); the key switching key (KSK) and the public functional key switching key (PFKSK).
- *Online Phase*: After the keys are established, the client submits encrypted data to the server using ciphertexts under its secret key. Each query consists of a mix of RLWE and LWE ciphertexts.

The sizes of the ciphertexts and cryptographic keys used in our protocol are detailed in Table 3 and the bandwidth consumption for each dataset is summarized in Table 4.

LWE	RLWE	BSK	KSK	PFKSK
5.9 KB	32 KB	47.6 MB	7.5 MB	8 MB

Table 3: Size of the ciphertexts and the keys used in our protocol.

Phase	Dataset	Bandwidth consumption
Online Phase	Iris	145.7 KB
	Wine	433.7 KB
	Cancer	971.8 KB
Offline Phase		63.1 MB

Table 4: Bandwidth consumption during the online phase for *one sample* of each dataset. The offline phase bandwidth consumption is not depending on the datasets

As expected, the Cancer dataset yields the largest bandwidth per sample due to its higher feature dimensionality, despite having fewer classes. Indeed, the two primary factors that influence the size of a sample are the number of features and the number of classes. To address this overhead, we leverage a well-established compression technique [56] for RLWE and LWE ciphertexts. These ciphertexts typically consist of two components: $(\vec{a}, b)$ for LWE, and $(A(X), B(X))$ for RLWE. By assuming a shared cryptographically secure pseudo-random generator (PRG) between client and server, we can transmit only the seed used to generate the random part (*i.e.*, $\vec{a}$ or $A(X)$). The server then reconstructs the full ciphertext using the PRG and the received seed plus b or $B(X)$. Applying this compression reduces the size of a single Cancer dataset sample from 971.8 KB to approximately 480 KB—nearly a 2x reduction.

7. Conclusion

In this paper, we presented an innovative method for training and unlearning tree-based models on encrypted data. Specifically, we employed the Extremely Randomized Trees (ERTs) framework, whose structure naturally aligns the training process of a sample with its inference process. This structural property allowed us to bridge the gap between private inference, a widely studied topic in Fully Homomorphic Encryption (FHE), and the previously unexplored challenge of private training and unlearning.

A major contribution of our work is the first protocol enabling privacy-preserving unlearning in machine learning. Thanks to the random and independent nature of ERT splits, data points can be efficiently removed from the model without retraining it from scratch—opening the door to practical and compliant data removal under encryption.

Our protocol provides two levels of privacy: a partially oblivious version, in which the server cannot distinguish training from unlearning, and a fully oblivious version, in which inference queries are also hidden. This indistinguishability is especially valuable in settings in which clients wish to conceal the nature of their interaction with the model. Experimentally, our approach shows that accuracy remains comparable to cleartext models in most settings, particularly when encrypted class counts remain within representable bounds. We also observe that the overflow of encrypted counters can paradoxically improve accuracy by pruning corrupted trees, a phenomenon we explain and quantify.

In terms of efficiency, our method outperforms existing FHE-based Random Forest approaches in training time by a factor of up to $2.4\times$, while maintaining competitive accuracy. Future work will aim to improve scalability to larger models, enhance the management of counter overflows and explore the homomorphic elimination of corrupted trees, enabling encrypted forests to self-correct without leaking information or requiring decryption.

References

[1] F. Butaru, Q. Chen, B. Clark, S. Das, A. W. Lo, and A. Siddique, "Risk and risk management in the credit card industry," *Journal of Banking & Finance*, vol. 72, pp. 218–239, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378426616301340

[2] A. Khemphila and V. Boonjing, "Comparing performances of logistic regression, decision trees, and neural networks for classifying heart disease patients," in *2010 international conference on computer information systems and industrial management applications (CISIM)*. IEEE, 2010, pp. 193–198.

[3] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: techniques, datasets and challenges," *Cybersecur.*, vol. 2, no. 1, p. 20, 2019. [Online]. Available: https://doi.org/10.1186/s42400-019-0038-7

[4] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, p. 3–42, Apr. 2006. [Online]. Available: https://doi.org/10.1007/s10994-006-6226-1

[5] Apple Machine Learning Research. (2024, Oct.) Combining machine learning and homomorphic encryption in the apple ecosystem. [Online]. Available: https://machinelearning.apple.com/research/homomorphic-encryption

[6] C. Hong, "Recent advances of privacy-preserving machine learning based on (fully) homomorphic encryption," *Security and Safety*, vol. 4, p. 2024012, 2025.

[7] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[8] Lindell and Pinkas, "Privacy Preserving Data Mining," *Journal of Cryptology*, vol. 15, no. 3, pp. 177–206, Jun. 2002. [Online]. Available: http://link.springer.com/10.1007/s00145-001-0019-2

[9] J. Vaidya, B. Shafiq, W. Fan, D. Mehmood, and D. Lorenzi, "A random decision tree framework for privacy-preserving data mining," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 5, pp. 399–411, 2014.

[10] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine learning classification over encrypted data," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/machine-learning-classification-over-encrypted-data

[11] D. J. Wu, T. Feng, M. Naehrig, and K. E. Lauter, "Privately evaluating decision trees and random forests," *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 4, pp. 335–355, 2016. [Online]. Available: https://doi.org/10.1515/popets-2016-0043

[12] A. Khedr, G. Gulak, and V. Vaikuntanathan, "Shield: scalable homomorphic implementation of encrypted data-classifiers," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2848–2858, 2015.

[13] R. K. Tai, J. P. Ma, Y. Zhao, and S. S. Chow, "Privacy-preserving decision trees evaluation via linear functions," in *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*. Springer, 2017, pp. 494–512.

[14] W.-j. Lu, J.-J. Zhou, and J. Sakuma, "Non-interactive and output expressive private comparison from homomorphic encryption," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 67–74.

[15] Á. Kiss, M. Naderpour, J. Liu, T. Schneider, and N. Asokan, "Sok: Modular and efficient private decision tree evaluation," *Proceedings on Privacy Enhancing Technologies*, vol. 2, 2019.

[16] A. Tueno, F. Kerschbaum, and S. Katzenbeisser, "Private evaluation of decision trees using sublinear cost," *Proceedings on Privacy Enhancing Technologies*, 2019.

[17] A. Tueno, Y. Boev, and F. Kerschbaum, "Non-interactive private decision tree evaluation," in *Data and Applications Security and Privacy XXXIV: 34th Annual IFIP WG 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25–26, 2020, Proceedings 34*. Springer, 2020, pp. 174–194.

[18] A. Akavia, M. Leibovich, Y. S. Resheff, R. Ron, M. Shahar, and M. Vald, "Privacy-preserving decision trees training and prediction," *ACM Trans. Priv. Secur.*, vol. 25, no. 3, May 2022. [Online]. Available: https://doi.org/10.1145/3517197

[19] S. Azogagh, V. Delfour, S. Gambs, and M.-O. Killijian, "Probonite: Private one-branch-only non-interactive decision tree evaluation." New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3560827.3563377

[20] K. Cong, D. Das, J. Park, and H. V. Pereira, "Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 563–577. [Online]. Available: https://doi.org/10.1145/3548606.3560702

[21] R. Akhavan Mahdavi, H. Ni, D. Linkov, and F. Kerschbaum, "Level up: Private non-interactive decision tree evaluation using levelled homomorphic encryption," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2945–2958. [Online]. Available: https://doi.org/10.1145/3576915.3623095

[22] J. Frery, A. Stoian, R. Bredehoft, L. Montero, C. Kherfallah, B. Chevallier-Mames, and A. Meyre, "Privacy-preserving tree-based inference with tfhe," in *Mobile, Secure, and Programmable Networking*, S. Bouzefrane, S. Banerjee, F. Mourlin, S. Boumerdassi, and É. Renault, Eds. Cham: Springer Nature Switzerland, 2024, pp. 139–156.

[23] K. Cong, J. Kang, G. Nicolas, and J. Park, "Faster private decision tree evaluation for batched input from homomorphic encryption," in *Security and Cryptography for Networks*, C. Galdi and D. H. Phan, Eds. Cham: Springer Nature Switzerland, 2024, pp. 3–23.

[24] R. Ko, R. A. Mahdavi, B. Yoon, M. Onizuka, and F. Kerschbaum, "Silentwood: Private inference over gradient-boosting decision forests," *arXiv preprint arXiv:2411.15494*, 2024.

[25] Z. Zhang, H. Zhang, X. Song, J. Lin, and F. Kong, "Secure outsourcing evaluation for sparse decision trees," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[26] S. Fukui, L. Wang, and S. Ozawa, "Efficient and privacy-preserving decision tree inference via homomorphic matrix multiplication and leaf node pruning," *Applied Sciences*, vol. 15, no. 10, p. 5560, 2025.

[27] Z. Brakerski and V. Vaikuntanathan, "Leveled fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[28] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2017.

[29] H. Shin, J. Choi, D. Lee, K. Kim, and Y. Lee, "Fully homomorphic training and inference on binary decision tree and random forest," in *Computer Security – ESORICS 2024: 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16–20, 2024, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 217–237. [Online]. Available: https://doi.org/10.1007/978-3-031-70896-1_11

[30] A. Mantelero, "The eu proposal for a general data protection regulation and the roots of the 'right to be forgotten'," *Computer Law & Security Review*, vol. 29, no. 3, pp. 229–235, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0267364913000654

[31] Y. Cao and J. Yang, "Towards making systems forget with machine unlearning," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 463–480.

[32] L. Bourtoule, V. Chandrasekaran, C. A. Choquette-Choo, H. Jia, A. Travers, B. Zhang, D. Lie, and N. Papernot, "Machine unlearning," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 141–159.

[33] T. T. Nguyen, T. T. Huynh, Z. Ren, P. L. Nguyen, A. W.-C. Liew, H. Yin, and Q. V. H. Nguyen, "A survey of machine unlearning," 2024. [Online]. Available: https://arxiv.org/abs/2209.02299

[34] S. Wang, Z. Shen, X. Qiao, T. Zhang, and M. Zhang, "Dynfrs: An efficient framework for machine unlearning in random forest," 2025. [Online]. Available: https://arxiv.org/abs/2410.01588

[35] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, p. 431–473, May 1996. [Online]. Available: https://doi.org/10.1145/233551.233553

[36] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *Advances in Cryptology – CRYPTO 2010*, T. Rabin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 502–519.

[37] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. H. Cheon and T. Takagi, Eds., vol. 10031, 2016, pp. 3–33. [Online]. Available: https://doi.org/10.1007/978-3-662-53887-6_1

[38] ——, "TFHE: fast fully homomorphic encryption over the torus," *IACR Cryptol. ePrint Arch.*, p. 421, 2018. [Online]. Available: https://eprint.iacr.org/2018/421

[39] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[40] S. Azogagh, Z. A. Birba, M.-O. Killijian, F. Larose-Gervais, and S. Gambs, "RevoLUT : Rust efficient versatile oblivious look-up-tables," Cryptology ePrint Archive, Paper 2024/1935, 2024. [Online]. Available: https://eprint.iacr.org/2024/1935

[41] Zama, "TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data," 2022, https://github.com/zama-ai/tfhe-rs.

[42] D. Trama, A. Boudguiga, P.-E. Clet, R. Sirdey, and N. Ye, "Designing a general-purpose 8-bit (t) fhe processor abstraction," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 2, pp. 535–578, 2025.

[43] S. Azogagh, M.-O. Killijian, and F. Larose-Gervais, "A non comparison oblivious sort and its application to k-nn," *Proceedings on Privacy Enhancing Technologies*, 2025.

[44] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, pp. 81–106, 1986.

[45] W.-Y. Loh, "Classification and regression trees," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 1, no. 1, pp. 14–23, 2011.

[46] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[47] C. W. Gini, "Variability and mutability, contribution to the study of statistical distributions and relations," *Studi Economico-Giuridici della R. Universita de Cagliari*, 1912. [Online]. Available: https://cir.nii.ac.jp/crid/1574231874891159808

[48] Y.-Y. Song and L. Ying, "Decision tree methods: applications for classification and prediction," *Shanghai archives of psychiatry*, vol. 27, no. 2, p. 130, 2015.

[49] H. Xu, T. Zhu, L. Zhang, W. Zhou, and P. S. Yu, "Machine unlearning: A survey," *ACM Comput. Surv.*, vol. 56, no. 1, Aug. 2023. [Online]. Available: https://doi.org/10.1145/3603620

[50] T. E. Mathew, "An optimized extremely randomized tree model for breast cancer classification," *Journal of Theoretical and Applied Information Technology*, 2022.

[51] U. Saeed, S. U. Jan, Y.-D. Lee, and I. Koo, "Fault diagnosis based on extremely randomized trees in wireless sensor networks," *Reliability Engineering and System Safety*, vol. 205, p. 107284, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095183202030781X

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

[53] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Wine," UCI Machine Learning Repository, 2009, DOI: https://doi.org/10.24432/C53Y8H.

[54] M. Lichman *et al.*, "Breast Cancer," UCI Machine Learning Repository, 2013, DOI: https://doi.org/10.24432/C50P49.

[55] C. Sano, "Japanese Credit Screening," UCI Machine Learning Repository, 1992, DOI: https://doi.org/10.24432/C5259N.

[56] M. Joye, "Sok: Fully homomorphic encryption over the [discretized] torus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, p. 661–692, Aug. 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9836

Appendix

1. RevoLUT operations

1.1. Blind Read. Introduced as Blind Array Access in [19], the operation of reading an encrypted value in an encrypted array at an encrypted index was further denoted as Blind Read in RevoLUT paper [40] because of its extension to multidimensional arrays.

Algorithm 8: Blind Read operation (BlindRead)

Input : A LUT ciphertext $[\![M]\!]_{\mathrm{LUT}}$ and a LWE ciphertext $[\![i]\!]_{\mathrm{LWE}}$.
Output: The $[\![i]\!]_{\mathrm{LWE}}$-th element of $[\![M]\!]_{\mathrm{LUT}}$: $[\![m_i]\!]_{\mathrm{LWE}}$.

$[\![M]\!]_{\mathrm{LUT}} \leftarrow \mathrm{BlindRotation}([\![i]\!]_{\mathrm{LWE}}, [\![M]\!]_{\mathrm{LUT}})$
$r \leftarrow \mathrm{SampleExtraction}(0, [\![M]\!]_{\mathrm{LUT}})$
return r

1.2. Blind Argmax. The proposed algorithm to compute the maximum value of a vector of BLWE ciphertexts is a single-pass scan with max/argmax update.

Algorithm 9: Blind Argmax of a vector of BLWE ciphertexts

Input : A vector of BLWE ciphertexts $M = ([\![m_0]\!]_{\mathrm{BLWE}}, \ldots, [\![m_{p-1}]\!]_{\mathrm{BLWE}})$
Output: A LWE ciphertext encrypting the index of the maximum value of M.

// Initialize the maximum value and its index
$max \leftarrow [\![m_0]\!]_{\mathrm{BLWE}}$
$argmax \leftarrow [0]_{\mathrm{LWE}}$
// Single-pass scan with max/argmax update
for $i \in [1, p-1]$ **do**
  $e \leftarrow [\![m_i]\!]_{\mathrm{BLWE}}$
  $b \leftarrow [\![max > e]\!]_{\mathrm{LWE}}$
  $A \leftarrow \mathrm{PFKS}([i]_{\mathrm{LWE}}, argmax)$
  $argmax \leftarrow \mathrm{BlindRead}(A, b)$
  // Update the higher and lower parts of the max value
  $H \leftarrow \mathrm{PFKS}(e_h, max_h)$
  $L \leftarrow \mathrm{PFKS}(e_l, max_l)$
  $max \leftarrow \mathrm{BlindRead}([\![H, L]\!]_{\mathrm{LUT}_{1,2}}, b)$
end
return $(argmax)$

1.3. BLWE ternary addition. Adding a ternary value to a BLWE ciphertext is not as straightforward as performing a simple addition on each LWE ciphertexts composing the BLWE ciphertext. This is because it involves managing carry boundaries. For instance, incrementing an encryption of the hex value $(1F)_{16}$ should result with a carry bit of 1 because $(1F)_{16} + (1)_{16} = (20)_{16}$ and not $(10)_{16}$. This carry bit have to remain encrypted and be used to update the next BLWE ciphertext only in the boundary cases like $(1F)_{16}$, $(10)_{16}$ and $(FF)_{16}$. In this paper, we need to blindly add a ternary value to an element in a Nibble-Byte-LUT ($[\![A]\!]_{\mathrm{LUT}_{1,2}}$) at an encrypted index. Due to this reason, we developed the BAAT function, which is presented in Algorithm 11. To better understand the BAAT function, we first present an oblivious functional switch-case that we developed in RevoLUT for this paper. In a nutshell, given an encrypted selector ℓ and encrypted data $[\![x]\!]_{\mathrm{LWE}}$, the protocol evaluates exactly one among multiple candidate functions homomorphically, while keeping both the selected branch and the data secret. This function is presented in Algorithm 10 with 3 cases but can be extended to p cases.

Algorithm 10: Oblivious functional switch-case 3 (OSW)

Input : A selector ℓ encrypted as an LWE ciphertext. An encrypted data $[\![x]\!]_{\mathrm{LWE}}$. A list of candidate functions (f_0, f_1, f_2) encrypted as LUT ciphertexts.
Output: The result of the selected function f_ℓ applied to $[\![x]\!]_{\mathrm{LWE}}$

$R \leftarrow \varnothing$
for $i \in [0, 2]$ **do**
  $R \leftarrow R \cup \mathrm{BlindRead}([\![f_i]\!]_{\mathrm{LUT}}, [\![x]\!]_{\mathrm{LWE}})$
end
$[\![R]\!]_{\mathrm{LUT}} \leftarrow \mathrm{PFKS}(R)$
$r \leftarrow BlindRead([\![R]\!]_{\mathrm{LUT}}, \ell)$
return r

Before defining the BAAT function, we need to define the function f_{Id} as the identity function, f_{+1} as the function that adds 1 to the input modulo p and f_{-1} as the function that subtracts 1 from the input modulo p.

2. Fully Oblivious Protocol

We detail our Fully Oblivious Protocol in Algorithm 12 as presented in Section 5.4.

3. Operations average time

We show in Table 5 the average time in seconds for the main operations in our protocol for the three datasets used in the experiments. This shows the impact of the dataset characteristics (number of features, number of classes, etc.) on the protocol runtime.

4. PCA of the datasets

The Figure 7 present the PCA of the Cancer and Credit datasets. The high separability of the classes in the Cancer dataset compared to the Credit dataset is clearly visible and can explain the overflow phenomenon observed in Figure 6.

Algorithm 11: Blind Array Add Ternary (BAAT)

Input : A Nibble-Byte-LUT $[\![A]\!]_{\text{LUT}_{1,2}}$. A encrypted index ℓ and a ternary value $[\![b]\!]_{\text{LWE}}$ where $b \in \{0, 1, 2\}$

Output: The Nibble-Byte-LUT $[\![A]\!]_{\text{LUT}_{1,2}}$ with the ternary value $[\![b]\!]_{\text{LWE}}$ added at the ℓ-th position

// Get the element to change in the LUT

$[\![x]\!]_{\text{BLWE}} \leftarrow \text{BlindRead}([\![A]\!]_{\text{LUT}}, \ell)$

// 1st round of switch-case

$r_l \leftarrow \text{OSW}(\ell, [\![x_l]\!]_{\text{LWE}}, ([\![f_{Id}]\!]_{\text{LUT}}, [\![f_{+1}]\!]_{\text{LUT}}, [\![f_{-1}]\!]_{\text{LUT}}))$

// 2nd round of switch-case

$t \leftarrow \text{OSW}(\ell, [\![x_l]\!]_{\text{LWE}}, ([\![0]\!]_{\text{LUT}}, [\![[0, \ldots, 1]]\!]_{\text{LUT}}, [\![[2, \ldots, 0]]\!]_{\text{LUT}}))$

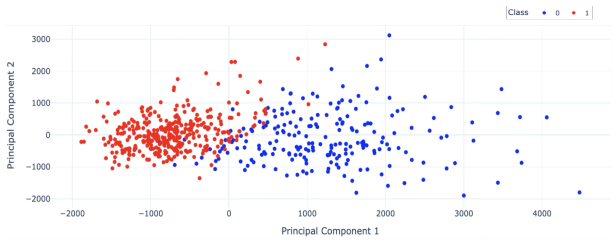
// 3rd round of switch-case

$r_h \leftarrow \text{OSW}(t, [\![x_h]\!]_{\text{LWE}}, ([\![f_{Id}]\!]_{\text{LUT}}, [\![f_{+1}]\!]_{\text{LUT}}, [\![f_{-1}]\!]_{\text{LUT}}))$

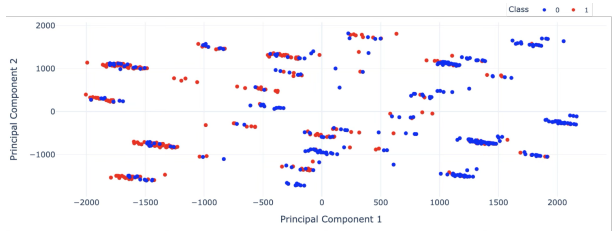
// Update the LUT

$[\![A]\!]_{\text{LUT}_{1,2}} \leftarrow \text{BlindWrite}([\![A]\!]_{\text{LUT}_{1,2}}, [\![\ell]\!]_{\text{LWE}}, (r_h, r_l))$

return $[\![A]\!]_{\text{LUT}_{1,2}}$



(a) Cancer dataset



(b) Japanese Credit Screening dataset

Fig. 7. Principal Component Analysis (PCA) in 2D of the Cancer and Credit datasets.

Algorithm 12: Fully Oblivious Protocol

Input : One or more samples $\mathcal{D} = \left[\left(F^i, L^i\right)\right]_{i=1}^{n}$ in which $F^i = ([\![F_0^i]\!]_{\text{RLWE}}, \ldots, [\![F_{k-1}^i]\!]_{\text{RLWE}})$ is the the encrypted features of the i-th sample and L^i is the label encoded and encrypted according to Equation 1. A forest with two types of trees $\mathcal{F} = (\mathcal{T}_0^{tr}, \ldots, \mathcal{T}_{m-1}^{tr}) \cup (\mathcal{T}_0^{inf}, \ldots, \mathcal{T}_{m-1}^{inf})$ and a current state $j \in [0, \beta - 1]$ before the next update of $(\mathcal{T}_i^{inf})_{i=0}^{m-1}$.

Output: One or more classifications $C = ([\![c]\!]_{\text{LWE}})_{i=1}^{n}$

$C \leftarrow \varnothing$

$i \leftarrow j$

for $(F, L) \in \mathcal{D}$ **do**

  // Update the model if the batch is complete

  if $i == \beta$ **then**

    for $(\mathcal{T}^{inf}, \mathcal{T}^{tr}) \in \mathcal{F}$ **do**

      $\mathcal{T}^{tr} \leftarrow \text{TreeMajority}(\mathcal{T}^{inf})$

    end

    $i \leftarrow 0$

  end

  // Get the prediction with $\mathcal{T}^{inf}$ and update the count of $\mathcal{T}^{tr}$

  for $(\mathcal{T}^{tr}, \mathcal{T}^{inf}) \in \mathcal{F}$ **do**

    $\mathcal{C} \leftarrow \text{Compile}(\mathcal{T}^{tr}, F)$

    $\ell \leftarrow \text{Traverse}(\mathcal{C})$

    $\mathcal{T}^{tr} \leftarrow \text{Update}(\mathcal{T}^{tr}.leaves, \ell)$

    $[\![r]\!]_{\text{LWE}} \leftarrow \text{BlindRead}(\mathcal{T}^{inf}.leaves, \ell)$

    $R \leftarrow R \cup [\![r]\!]_{\text{LWE}}$

  end

  $[\![c]\!]_{\text{LWE}} \leftarrow \text{Majority}(R)$

  $C \leftarrow C \cup [\![c]\!]_{\text{LWE}}$

  $i \leftarrow i + 1$

end

return C

Dataset	m	Compile	Traverse	Update	TreeMajority	BlindRead	BlindCount	BlindArgmax	Majority
Iris	8	0.000032	0.089215	0.807610	15.102815	0.073092	0.581338	0.687836	1.269222
	16	0.000032	0.092778	0.829066	15.056918	0.068593	1.156436	0.655479	1.811960
	32	0.000037	0.090193	0.822244	15.068344	0.069083	2.385299	0.658220	3.043561
Wine	8	0.000041	0.089131	0.820877	15.043914	0.068837	0.566600	0.657254	1.223895
	16	0.000040	0.089463	0.820111	15.054799	0.069017	1.176779	0.657477	1.834303
	32	0.000040	0.089278	0.821859	15.042277	0.068908	2.357210	0.660547	3.017799
Cancer	8	0.000042	0.089257	0.540498	9.762715	0.068869	0.565306	0.326632	0.891981
	16	0.000044	0.089342	0.539478	9.749358	0.069208	1.163370	0.327015	1.490422
	32	0.000042	0.089166	0.538513	9.758339	0.068781	2.363703	0.328953	2.692696

Table 5: Average time in seconds for the operations in our protocol, standardized to the same number of samples across different datasets and varying forest sizes m of $\mathcal{F}$. Note that the $\mathrm{BlindRead}$ operation includes the packing of 2^4 leaves with the PFKS operation and that $\mathrm{Majority}$ operations is for the whole forest $\mathcal{F}$.