Oblivious Exact (Un)Learning of Extremely Randomized Trees

Sofiane Azogagh, Zelma Aubin Birba, Sébastien Gambs, Marc-Olivier Killijian
Université du Québec à Montréal, Montréal, Canada
azogagh.sofiane@uqam.ca

***Abstract*—Recent regulations such as the GDPR have given the right to be forgotten to users, which requires that they can ask for the deletion of their data. Yet, enforcing such deletions for machine learning (ML) models remains challenging, especially when servers are untrusted or may ignore requests. To address this issue, we present the first ML model to support *oblivious exact unlearning*, in which the deletion is computationally indistinguishable from regular training or inference. This ensures that unlearning can be enforced without revealing its occurrence to the server. Our construction is based on *Extremely Randomized Trees* (ERTs), which are well-suited for encrypted training and efficient unlearning. More precisely, their randomized data-independent structure enables exact sample removal without retraining. We instantiate our protocol within the *TFHE* framework by designing a non-interactive procedure for encrypted updates, traversals and inference. Our implementation shows that encrypted ERTs train up to 2.4× faster than prior encrypted random forests while maintaining a comparable accuracy.**

***Index Terms*—Privacy-preserving ML, Extremely Randomized Trees, Homomorphic Encryption, TFHE, Exact Unlearning, Obliviousness.**

I. 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 [4] and boosting [5]. In particular, Extremely Randomized Trees (ERTs) [6], a stochastic variant of the classical 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.

However, ERTs and more generally ML models are often training on sensitive data, which makes the protection of privacy during training and inference a critical issue. Fully Homomorphic Encryption (FHE) schemes offer a cryptographic solution to this challenge, enabling computation directly over encrypted data without ever decrypting it [7]. While substantial advancements in applying FHE to inference tasks have been made recently, allowing encrypted predictions using pretrained models, the training of ML models over encrypted data remains an underexplored frontier [8]. This can be explained in part by the prohibitive computational overhead associated with FHE operations, despite the progress made in schemes like TFHE [9], which support efficient programmable bootstrapping and bit-level logic gates. To circumvent this issue, in this work we leverage the inherent connection between training and inference in ERTs, and instantiate this idea within the TFHE framework to present a novel protocol that enables efficient training of a tree-based model over encrypted datasets. To achieve this, we tackle several challenges, including the encrypted computation of histograms, randomized split point selection and secure tree construction—entirely in the encrypted domain.

As an additional fundamental contribution, we introduce the first protocol enabling oblivious exact unlearning in ML. Motivated by recent reports that companies like Google do not comply with the right to be forgotten [10], the need to conceal the nature of the operation being performed on the model is a way to enforce unlearning requests. Thanks to the random and independent nature of ERT splits, data points can be efficiently removed from the model—opening the door to practical and compliant data removal under encryption.

a) Outline*:* The paper is organized as follows. First, in Section II we review the related work on training and inference of tree-based models over encrypted data as well as private unlearning in a broader machine learning context beyond tree-based models. Then, in Section III, we describe the system and adversary models before in Section IV providing the necessary background notions. Afterwards, we detail our proposal in Section V before presenting the experimental results in Section VI and concluding in Section VII.

II. Related Work

Privacy-preserving decision tree learning has been an active research area for a decade as highlighted in Figure 1, with most approaches leveraging advanced cryptographic primitives such as Secure Multiparty Computation (SMPC) [11], [12] and Homomorphic Encryption (HE) [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29] for training and/or inference over encrypted or secret-shared data. Among HE-based protocols, schemes like BFV [30], CKKS [31] and TFHE [9] are particularly prominent due to their rich arithmetic capabilities and expressiveness.

a) 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

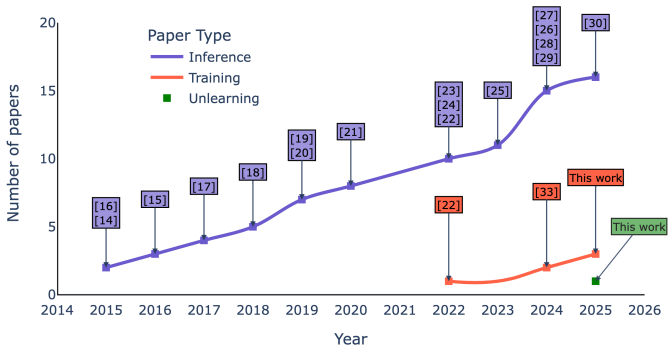


Fig. 1. Chronology of related work on tree-based model training and inference using HE.

collaborators [26] 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 [24]. Within the TFHE ecosystem, Probonite [22] introduced the first protocol performing a number of non-interactive comparisons linear in the tree depth, substantially accelerating evaluation. SortingHat [23] further improved performance by integrating a new comparison technique and a transciphering layer that minimizes ciphertext size. More recently, Frery and co-authors [25] made substantial use of programmable bootstrapping to implement efficient tree traversal and comparisons, pushing the limits of bit-level encrypted inference.

b) 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 information about the data. Akavia and collaborators [21] proposed one of the few interactive training schemes based on CKKS, using polynomial approximations for encrypted comparisons and achieving communication complexity linear in the tree depth. While efficient, their method requires interaction and costly comparison evaluations. Our approach departs from this paradigm by employing a novel data encoding technique (detailed in Section V-A), which eliminates the need for explicit comparisons, enabling a fully non-interactive and client-offline training procedure. Recently, Shin and co-authors [32] 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.

c) Private unlearning*:* Driven by privacy regulations such as the GDPR and the associated "right to be forgotten" [33], the area of machine unlearning has gained significant attention in recent years [34], [35], [36]. For instance, Wang and collaborators [37] have introduced a method for private *approximate* unlearning that relies on retraining the model over a carefully constructed dataset designed to eliminate the influence of the data sample to be unlearned. To the best of our knowledge, this is the only existing approach that explicitly acknowledges the sensitivity of unlearned profiles and proposes an approximate mechanism to preserve their privacy.

Prior research, including [38], [39], has investigated methods for verifying that a data sample has been effectively unlearned upon request. Consistent with our assumptions, these approaches acknowledge the potential for a server to deny an unlearning request, thereby underscoring the importance of unlearning verification mechanisms. Some efforts have examined unlearning *in the clear domain* on gradient-boosted decision trees [40], [41] and random forests [42], [43] and ERTs [44], focusing on making the unlearning process cheaper than a mere complete retraining without the unlearned data, our work is the first to introduce *oblivious exact* unlearning for ERTs, specifically in the context in which both the training data and the unlearned instances remain encrypted throughout the process. More broadly, our proposal constitutes the first *oblivious and exact* unlearning protocol for an ML model under homomorphic encryption, setting a precedent for regulatory-compliant, encrypted model adaptability.

III. System and Adversary Models

We consider the traditional outsourcing setting in which a client holds private data and delegates ML tasks to a cloud server. Specifically, the client may request training, unlearning, or inference operations on an ERT ensemble model hosted by the server, by sending data encrypted under their key. 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 which are encrypted under their respective keys in a multi-key scheme [45].

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 dataset is typically associated with better model utility, a 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 mitigate 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 is similar in spirit with assumptions commonly made in the context of Oblivious RAM (ORAM) [46]. 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 versus write) indistinguishable to a semi-honest server [47]. Analogously, in our setting, our objective is to conceal whether a client is performing a read operation on the ERTs (*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.

IV. Background

In this section, we introduce the background on decision trees, ERTs and on the TFHE cryptosystem necessary to understand our work.

A. Decision Trees and Random Forests

Decision trees [48], [49] are widely used ML models that recursively partition the feature space to produce predictions. Each internal node $\mathcal{N}$ of a tree $\mathcal{T}$ is associated with a feature, and, for numerical features, a threshold θ that guides the decision at that node. The tree is traversed from the root by evaluating each node's condition, branching accordingly until a terminal *leaf* node is reached, which outputs the prediction of the associated majority label. Thanks to their intuitive structure and interpretability, decision trees can be applied for classification and regression and are inherently interpretable by design. Performance can be further improved by combining multiple trees into ensembles such as Random Forests [50], which aggregate outputs—typically by majority vote or averaging—resulting in better accuracy and robustness.

a) Training*:* The training of a decision tree consists in recursively partitioning the dataset into increasingly homogeneous subsets with respect to the target variable. At each step, the data is split based on a selected feature and, for numerical features, a threshold. The split is chosen using a scoring criterion that aims to reduce impurity—typically Gini Index [51] or Information Gain [52]. The process continues by creating a new node for each subset and repeating the split until a stopping condition is met, such as all samples sharing the same class label, reaching a maximum tree depth or having too few samples. Typically, each leaf stores class counters summarizing the label distribution of samples within it. These counters are used during inference to determine the predicted label and possibly a confidence score.

b) Inference*:* The inference in a decision tree consists in predicting the output for new input data by traversing the tree structure. More precisely, starting from the root, the value of attribute associated with the current node is compared against the node's threshold (for numerical features) or categories (for categorical ones), directing the traversal left or right until a leaf is reached. The predicted class then corresponds to the majority label of the samples in that leaf. In ensembles such as Random Forests or ERTs, predictions from multiple trees are aggregated—typically by majority vote for classification, or averaging for regression.

c) Unlearning*:* Machine unlearning aims to remove the influence of a specific training sample, effectively "forgetting" it [53]. The naive approach—retraining from scratch without the target sample—is often computationally prohibitive and thus more efficient unlearning approaches have been designed. For instance, SISA (Sharded, Isolated, Sliced and Aggregated training) [35] is a model-agnostic approach that partitions the training data into independent slices, each used to train a separate model. To unlearn, only the model learnt on the relevant slice must be retrained, significantly reducing computation—but still requiring access to part of the data and non-negligible effort. SISA supports *exact* unlearning as the resulting model behaves as if the sample had never been included.

B. Extremely Randomized Trees

Introduced by Geurts, Ernst and Wehenkel in 2006 [6], ERTs offer an efficient and highly randomized variant of random forests. The key idea behind ERTs is to inject randomness in both the choice of features and the associated thresholds. 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 [54], and have outperformed other learning algorithms such as neural networks, (classical) random forests and support vector machines in fault detection for wireless sensor networks [55]. 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, completely randomized ERTs ($k' = 1$) construct their structure independently of the data. Thus, one can reprocess the target sample to locate the corresponding leaf and decrement the associated class counter to perform unlearning, which makes ERTs especially well-suited for cryptographic protocols that require both learning and unlearning over encrypted data. We emphasize that while unlearning does not require choosing the best split for each node again, the tree traversal and the selection of the right class counter remain significant challenges that we address to make our solution efficient.

Hereafter, a forest of ERTs denotes an ensemble of ERTs constructed independently (features and thresholds sampled independently), for which a classification decision is made by taking a majority vote between the individual decisions of the constituent trees.

C. The TFHE Cryptosystem

The TFHE encryption scheme, introduced in 2016 [56], [57], is built upon the hardness of the Learning With Errors (LWE) problem and its ring-based variant, Ring-LWE (RLWE). 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.

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.

a) 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, $\Delta = q/2p$ is the message scaling factor and e is a noise term drawn from a Gaussian distribution with standard deviation σ_{LWE}.

b) 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}.

c) 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, $[\![M]\!]_{\mathrm{LUT}} = [\![m_0, \ldots, m_{p-1}]\!]_{\mathrm{LUT}}$ denotes a LUT encryption of message vector $M = (m_0, \ldots, m_{p-1})$; $[\![m]\!]_{\mathrm{LWE}}$ denotes a standard LWE ciphertext of scalar message m and $[m]_{\mathrm{LWE}}$ indicates a trivially encrypted LWE ciphertext (*i.e.*, one with zero noise and zero mask).

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):** $([\![\star]\!]_{\mathrm{LWE}}, [\![\star]\!]_{\mathrm{LUT}}) \rightarrow [\![\star]\!]_{\mathrm{RLWE}}$. 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 $[\![i]\!]_{\mathrm{LWE}}$, without revealing this rotation index.
- **Sample Extraction (SE):** $(\star, [\![\star]\!]_{\mathrm{RLWE}}) \rightarrow [\![\star]\!]_{\mathrm{LWE}}$. 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 $[\![m_j]\!]_{\mathrm{LWE}}$. It operates by isolating and re-encoding one coefficient while discarding the rest.
- **Key Switching (KS):** $[\![\star]\!]_{\mathrm{LWE}} \rightarrow [\![\star]\!]_{\mathrm{LWE}}$. 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):** $\{[\![\star]\!]_{\mathrm{LWE}}\} \rightarrow [\![\star]\!]_{\mathrm{RLWE}}$. Introduced in [58] (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 BR is preceded by a KS operation, while SE is omitted due to its negligible execution time and lack of noise accumulation. Among all TFHE primitives, the most computationally significant are the BR and the 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.

D. RevoLUT Operations

RevoLUT [59] is a library built on top of TFHE-RS [60] 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.

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 [61].

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

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

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—$\mathrm{LUT}_{1,2}$—introduced in [59], 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 $\mathrm{LUT}_{1,2}$ ciphertext as follows

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

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.

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 $\mathrm{BlindRead}$ procedure is then implemented using a combination of BR followed by SE as detailed in Algorithm 6.

The same approach extends naturally to operations on Nibble-Byte-LUT ciphertexts, such as oblivious reads and writes. For example, the $\mathrm{BlindRead}$ 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

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

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 8. Its computational complexity, expressed in terms of BLINDROTATION and PFKS operations, is given by

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

Another essential operation provided by RevoLUT is an oblivious counting mechanism developed in [62], which computes the number of encrypted elements in a vector of BLWE or LWE ciphertexts. We refer to this procedure as $\mathrm{BlindCount}$. The time complexity of this operation is given by

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

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

V. 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.

A. 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.

a) TFHE parameters choice*:* As discussed earlier, TFHE natively supports encrypted messages with precision limited to $\log_2(p)$ bits. While the TFHE-RS library [60] 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 BR 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 is summarized in Appendix A).

b) 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 $F = ([\![F_0]\!]_{\mathrm{RLWE}}, \ldots, [\![F_{k-1}]\!]_{\mathrm{RLWE}})$. This encoding, inspired by [25], allows the server to evaluate comparisons without explicitly performing them. If a decision node specifies feature index I and threshold θ, the server retrieves $[\![F_I]\!]_{\mathrm{RLWE}}$ and performs a Sample Extraction at index θ to obtain $[\![\theta < f_I]\!]_{\mathrm{LWE}}$. 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

$$l_i = \begin{cases} 1, & \text{if training and } l_i \neq 0 \\ 2, & \text{if unlearning and } l_i \neq 0 \\ 0, & \text{if inference and } l_i \neq 0 \end{cases} \quad (1)$$

Although -1 could have represented unlearning, we use 2 for efficiency in the BAAT operation (see Algorithm 10) 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 $S = (F, L)$.

c) 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 chosen TFHE parameters sets N to 2^{11} (see Table III), which enables us to 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 a few samples (*e.g*, for inference).

B. Building Blocks

This section introduces the main components upon which our protocol is built. We first describe the different types of trees employed in our construction, followed by the key algorithms used to manipulate and operate on them.

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

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

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 different tree types in Figure 2, and proceed to formally define them in the following subsections.

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

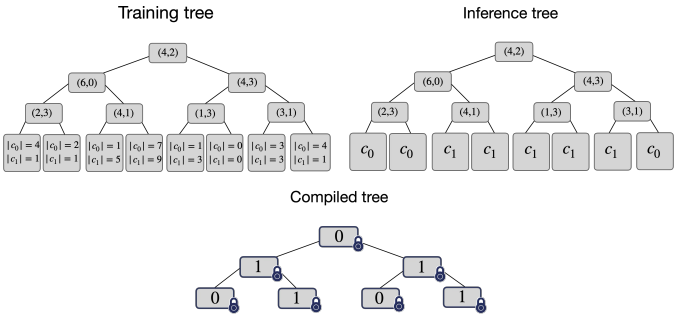


Fig. 2. Illustration of the training tree $\mathcal{T}^{tr}$, inference tree $\mathcal{T}^{inf}$ and compiled tree $\mathcal{T}^{comp}$.

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 further discussed in Section VI-B.

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 11.

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:

$$\mathcal{N}_i = (\theta_i, I_i) \quad \text{and} \quad \mathcal{L}_i = \llbracket c \rrbracket_{\mathrm{LWE}}$$

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 $\mathrm{Majority}$ algorithm, detailed in Algorithm 12.

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

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

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 1). 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.

Algorithm 1: 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 $\mathcal{T}^{comp}$.

$\mathcal{T}^{comp} \leftarrow \varnothing$
for $i \in [0, d-1]$ **do**
  // Compile the i-th level
  $Level_c \leftarrow \varnothing$
  for $(\theta, I) \in Level_i$ **do**
    $b \leftarrow \text{SampleExtraction}(\theta, [\![F_I]\!]_{\text{RLWE}})$
    $Level_c \leftarrow Level_c \cup b$
  end
  $\mathcal{T}^{comp} \leftarrow \mathcal{T}^{comp} \cup Level_c$
end
return $\mathcal{T}^{comp}$

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

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

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

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

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 2, 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 2: 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

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

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

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 [59], detailed in Algorithm 10. The objective BAAT is to add an encrypted ternary value (*i.e*, 0 , 1 or -1) to a

Algorithm 3: Updating the leaves (Update)

Input : A training tree $\mathcal{T}$ with 2^d leaves $\mathcal{L}_i = (\llbracket c_0 \rrbracket_{\text{BLWE}}, \ldots, \llbracket c_{\gamma-1} \rrbracket_{\text{BLWE}})$ in which d is the depth of the tree. A selector $\llbracket \ell \rrbracket_{\text{LWE}}$ 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 ∈ [0, γ − 1] do
    v ← ∅
    for i ∈ [0, 2^d − 1] do
        v ← v ∪ ⟦c_j⟧_BLWE
    end
    ⟦C_j⟧_LUT1,2 ← PFKS(v)
end
// Update the leaves
for j ∈ [0, γ − 1] do
    ⟦C_j⟧_LUT1,2 ← BAAT(⟦C_j⟧_LUT1,2, ⟦ℓ⟧_LWE, L_j)
end
return T
```

Nibble-Byte-LUT at an encrypted index ℓ computed via the Traverse algorithm.

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

$$\begin{aligned} t_{\text{Update}} &= \gamma \times t_{\text{BAAT}} \\ &= 10\gamma \times t_{\text{BR}} + 4\gamma \times t_{\text{PFKS}}, \end{aligned}$$

in which γ denotes the number of classes.

C. Partially Oblivious Protocol

In this section, we detail the core components of our first protocol, which we call partially oblivious as we demonstrate the indistinguishability of the training and unlearning procedures. This protocol will be further refined in the follow-up section to make it fully oblivious. Additionally, we describe the inference process and examine the balance between efficiency and privacy within our design.

1) Training and Unlearning Processes: The training and unlearning processes are described in Algorithm 4. They take as input a dataset $\mathcal{D}$ and either generate a new forest $\mathcal{F}$ composed of ERTs or operate 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.

Algorithm 4: Training/Unlearning process

Input : A dataset $\mathcal{D} = \left[\left(F^i, L^i\right)\right]_{i=1}^{n}$ in which $F^i = (\llbracket F^i_0 \rrbracket_{\text{RLWE}}, \ldots, \llbracket F^i_{k-1} \rrbracket_{\text{RLWE}})$ 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 ∈ [0, m − 1] do
    T_i ← RandomTree()
end
// Train/Unlearn the trees
for (F, L) ∈ D do
    for T ∈ F do
        C ← Compile(T, F)
        ℓ ← Traverse(C)
        T ← Update(T.leaves, L, ℓ)
    end
end
// Compute the forest ready for inference
for T ∈ F do
    T ← TreeMajority(T)
end
return F
```

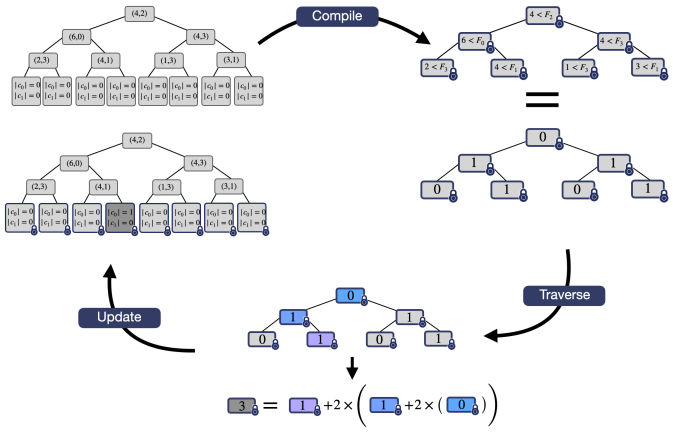


Fig. 3. Overview of the training process at the beginning, in which leaves are set to $\{\llbracket 0 \rrbracket_{\text{BLWE}}\}_{j=0}^{\gamma-1}$ for $\gamma = 2$. The feature vector used for updating the leaves is $f = [5, 3, 7, 1, 0]$ and the label is $l = 0$ encoded and encrypted as explained in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

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 $\mathrm{TreeMajority}$. 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

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

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

Theorem 1 (Indistinguishability of training and unlearning)**.** The training and unlearning processes executed by Algorithm 4 are indistinguishable.

Proof. The proof is given in Appendix D. □

Theorem 2 (Exact unlearning)**.** The unlearning process executed by Algorithm 4 on a forest $\mathcal{F}$ and a sample (F, L) yields the same forest as if (F, L) had never been included in the training set.

Proof. The proof follows from the fact that the traversal is determinist and that the training process increments the class counters in the leaves of the tree while the unlearning process decrements them. □

2) Inference and Discussion: 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 $\mathrm{Compile}$ function. The compiled tree is then traversed via the $\mathrm{Traverse}$ function, yielding an encrypted index ℓ corresponding to the reached leaf. The prediction associated with this leaf is then retrieved using the $\mathrm{BlindRead}$ function. After collecting predictions from all trees in the forest, a final majority vote is performed using the $\mathrm{Majority}$ function to determine the predicted class. The detailed procedure is provided in Algorithm 5. The total computational cost of the inference procedure is expressed as

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

The oblivious nature of the protocol stems from the shared use of the $\mathrm{Compile}$ and $\mathrm{Traverse}$ 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* as it guarantees that training and unlearning queries are indistinguishable from each other. 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 5 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.

Algorithm 5: Inference process

Input : A feature vector $F = (\llbracket F_0 \rrbracket_{\mathrm{RLWE}}, \ldots, \llbracket F_{k-1} \rrbracket_{\mathrm{RLWE}})$.
A vector of encyption of 0 serving as label $L = (\llbracket 0 \rrbracket_{\mathrm{LWE}})_{i=0}^{\gamma-1}$.
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 ← ∅
for (F, L) ∈ D do
    for T ∈ F do
        C ← Compile(T, F)
        ℓ ← Traverse(C)
        [[r]]_LWE ← BlindRead(T.leaves, ℓ)
        R ← R ∪ [[r]]_LWE
    end
end
// Get the majority vote
c ← Majority(R)
return c
```

D. Fully Oblivious Protocol

To make the client's queries—whether for training, unlearning or inference—indistinguishable to the server, we propose three different strategies to achieve full query-type obliviousness that have different privacy-utility trade-off.

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 4 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 $\llbracket 0 \rrbracket_{\mathrm{LWE}}$. 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 with a total runtime of

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

One way to amortize this cost is to leverage batch processing as in continual learning. 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 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 13 in Appendix V-D and its resulting runtime is

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

Theorem 3 (Obliviousness)**.** The Algorithm 13 gives obliviousness between training, unlearning and inference.

Proof. The proof is given in Appendix D. □

VI. Experimental Evaluation

In this section, we describe the experiments conducted to validate our approach with our code publicly available on GitHub[1]. 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 ML Repository: Iris [63], Wine [64] and Breast Cancer [65] whose characteristics are summarized in Table I.

Table I: Datasets used in the experiments.

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
Japanese Credit	523	130	15	2

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 non-binary 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.

A. 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 4, applied to the three datasets summarized in Table I, are reported in Figure 5. As observed, the 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 Compile, Traverse and Update operations, while the number of classes primarily impacts the complexity of TreeMajority, Update and Majority (see Table V in Appendix E). 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.

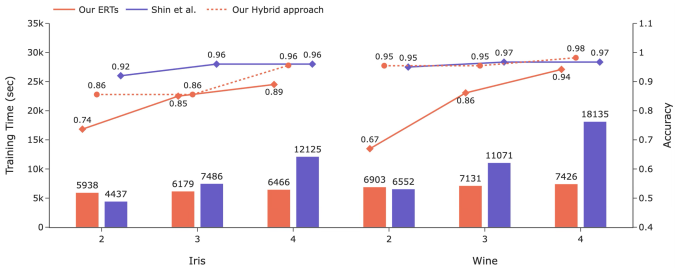


Fig. 4. A comparison of training times and accuracy between our private ERTs and the private Random Forest model from [32], evaluated across different tree depths. The hybrid approach is described latter in Section VI-B. For a fair comparison, we took the same amount of samples for both models, which correspond to those reported in Table 3 of [32]. Both models have a forest size of 64.

Regarding the comparison with the privacy-preserving Random Forest model employing the CKKS scheme [32], the runtime reported in their paper (see Figure 4) shows that our protocol achieves a speedup ranging from 1.2× to 2.4× for tree depths of 3 or more, with only a slight reduction in accuracy. We reported an hybrid approach (discussed in Section VI-B) which makes it possible to attain comparable accuracy to the baseline, further narrowing this gap. We emphasize that, unlike [32], our reported training time explicitly includes the cost of the 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.

B. Accuracy Analysis

The accuracy of ERTs is inherently variable due to the randomized feature and threshold selections. Recall that our objective is not to improve the ERT accuracy but rather to analyze how applying ERTs to encrypted data affects the 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

[1]https://github.com/sofianeazogagh/oblivious_unlearning

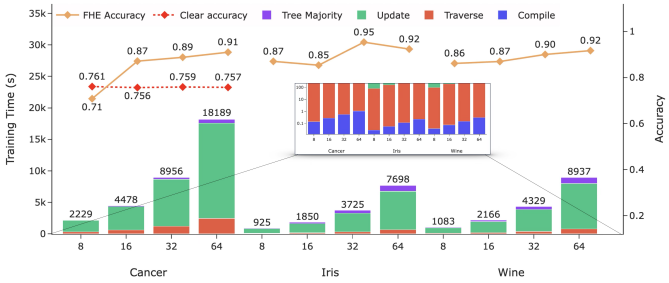
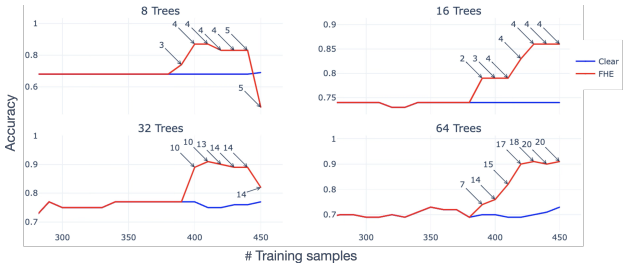


Fig. 5. Training time and accuracies across various ensemble sizes $\mathcal{F}$ (*i.e*, $8, 16, 32$ and 64) and the different datasets used. The results presented were obtained with the training set sizes listed in Table I

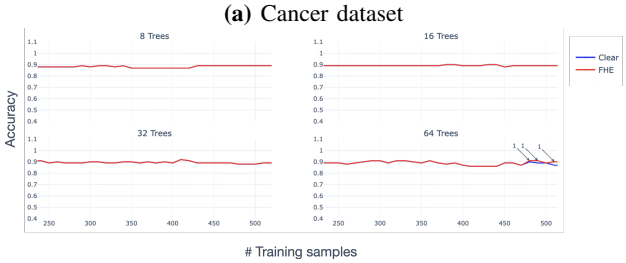
data but encrypted. Our results show that the accuracy of encrypted models depends not only on the dataset but also on the forest size and the encryption constraints. As depicted in Figure 5, the Iris dataset yielded the highest accuracy, which is expected due to its small size (few samples and features), 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. One simple solution to this limitation is to increase the radix of the counters by adding more LWE components to each radix ciphertext, thus expanding the representable count range. By doing so, only the Update and TreeMajority operations would be affected, and their runtime would grow linearly with the number of LWE components, while the counting range would increase exponentially. For instance, increasing the counter from 2 to 3 LWE components raises the maximum count from $16^2 = 256$ to $16^3 = 4096$, incurring an estimated overhead of 50%.

Interestingly, this overflow effect can sometimes improve accuracy. For instance, as shown in Figure 6a, the encrypted model's accuracy on the Cancer dataset initially aligns with the cleartext 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. For instance, when 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.



(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 [66]. 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.

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 [66], 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 8b, 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) ***Hybrid approach****:* Recognizing that selecting the optimal model structure is unrealistic, we introduce a hybrid approach. In this approach, the server initially trains an ERT model on a set of auxiliary public data D_0 (it can also be its own private data), with a feature subset size $k' > 1$ (recall that k' denotes the number of candidate features considered for each split). Using a Gini-index-based criterion and D_0, the best split is chosen among the k' candidates at each node. These selected nodes are then treated as fixed hyperparameters of the model. Afterwards, they can receive encrypted queries

forming a new dataset D_1 to further fine-tune the model, with these encrypted samples only updating the leaves counters, not the internal splits. Experimental results shown in Figure 7 that this hybrid approach achieves a better accuracy than the former approach and even better than [32].

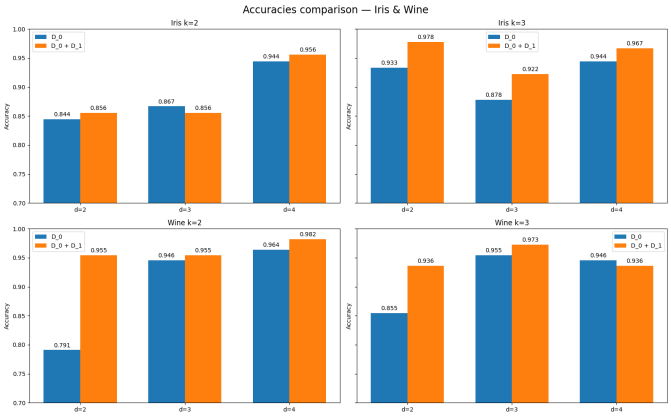


Fig. 7. Accuracy evolution in the hybrid approach on Iris and Wine datasets: initial Gini-based split selection on D_0 with $k' \in \{2, 3\}$ random candidate splits per node, then fine-tuning on encrypted data D_1 by updating only leaf counters.

C. 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 including public keys.
- *Online Phase*: The client submits encrypted data consisting of a mix of RLWE and LWE.

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

Table II: Bandwidth consumption during the online phase for *one sample* of each dataset. The offline phase bandwidth consumption does not depend on the characteristics of the datasets.

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

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. One way to reduce this overhead is to use a standard compression technique for LWE and RLWE ciphertexts [67]. Since such ciphertexts consist of two components—$(\vec{a}, b)$ for LWE and $(A(X), B(X))$ for RLWE, we assume a shared cryptographically secure pseudo-random generator (PRG) between client and server. Only the seed generating the random component ($\vec{a}$ or $A(X)$) is transmitted, while the server reconstructs the full ciphertext from the seed and the corresponding b or $B(X)$. This approach reduces the size of a single Cancer dataset sample from 971.8 KB to about 480 KB, achieving nearly a 2× compression.

VII. Conclusion

In this paper, we have proposed an innovative method for training and unlearning tree-based models on encrypted data using the Extremely Randomized Trees (ERTs) framework. The structural alignment between training and inference in ERTs enabled us to bridge private inference with the previously unexplored challenge of private training and exact unlearning, making our work the first protocol enabling oblivious exact unlearning in ML. Thanks to the random and independent nature of ERT splits, data points can be efficiently removed without retraining from scratch, enabling practical and compliant data removal under encryption.

We have proposed two variants: a *partially oblivious protocol* where the server cannot distinguish training from unlearning, and a *fully oblivious* one where inference queries are also hidden. Experimentally, our approach achieves accuracy comparable to cleartext models when the number of training samples remain within representable bounds. We further introduced a *hybrid approach* that initially trains on auxiliary public data using Gini-based split selection, then fine-tunes on encrypted data by updating only leaf counters. This hybrid method achieves better accuracy than our baseline approach and even outperforms existing FHE-based Random Forest methods, while our protocol achieves training speedups of up to $2.4\times$ compared to existing approaches. Future work will focus on improving scalability, enhancing counter overflow management, and exploring homomorphic removal of corrupted trees.

LLM usage considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

Acknowledgments

This work is supported by the DEEL Project CRDPJ 537462-18 funded by the National Science and Engineering Research Council of Canada (NSERC) and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ), together with its industrial partners Thales Canada inc, Bell Textron Canada Limited, CAE inc and Bombardier inc[2]. Sébastien Gambs is also supported by the Canada Research Chair program and a Discovery Grant from NSERC while Marc-Olivier Killijian is supported by Discovery Grants from NSERC.

[2]https://deel.quebec

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] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[5] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[6] 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

[7] 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

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

[9] 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.

[10] T. C. Press, "Google refusing to comply with "right to be forgotten" de-listing decision," *National Post*, August 2025. [Online]. Available: https://nationalpost.com/news/canada/google-not-complying-with-canadas-right-to-be-forgotten

[11] 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

[12] 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.

[13] 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

[14] 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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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

[33] 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

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

[35] 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.

[36] 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

[37] W. Wang, Z. Tian, C. Zhang, and S. Yu, "Oblivious unlearning by learning: Machineunlearning without exposing erased data," 2024.

[38] X. Gao, X. Ma, J. Wang, Y. Sun, B. Li, S. Ji, P. Cheng, and J. Chen, "Verifi: Towards verifiable federated unlearning," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 6, pp. 5720–5736, 2024.
[39] K. Wu and W. H. Wang, "Verification of incomplete graph unlearning through adversarial perturbations," in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, ser. KDD '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 3204–3215. [Online]. Available: https://doi.org/10.1145/3711896.3737179
[40] H. Lin, J. W. Chung, Y. Lao, and W. Zhao, "Machine unlearning in gradient boosting decision trees," ser. KDD '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1374–1383. [Online]. Available: https://doi.org/10.1145/3580305.3599420
[41] Z. Wu, J. Zhu, Q. Li, and B. He, "Deltaboost: Gradient boosting decision trees with efficient machine unlearning," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–26, 2023.
[42] J. Brophy and D. Lowd, "Machine unlearning for random forests," in *International Conference on Machine Learning*. PMLR, 2021, pp. 1092–1104.
[43] 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
[44] S. Schelter, S. Grafberger, and T. Dunning, "Hedgecut: Maintaining randomised trees for low-latency machine unlearning," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1545–1557.
[45] H. Chen, I. Chillotti, and Y. Song, "Multi-key homomophic encryption from TFHE," Cryptology ePrint Archive, Paper 2019/116, 2019. [Online]. Available: https://eprint.iacr.org/2019/116
[46] 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
[47] 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.
[48] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, pp. 81–106, 1986.
[49] W.-Y. Loh, "Classification and regression trees," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 1, no. 1, pp. 14–23, 2011.
[50] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.
[51] 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
[52] 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.
[53] 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
[54] T. E. Mathew, "An optimized extremely randomized tree model for breast cancer classification," *Journal of Theoretical and Applied Information Technology*, 2022.
[55] 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
[56] 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
[57] ——, "TFHE: fast fully homomorphic encryption over the torus," *IACR Cryptol. ePrint Arch.*, p. 421, 2018. [Online]. Available: https://eprint.iacr.org/2018/421
[58] 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.
[59] 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
[60] 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.
[61] 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.
[62] 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.
[63] R. A. Fisher, "Iris," UCI Machine Learning Repository, 1936, DOI: https://doi.org/10.24432/C56C76.
[64] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Wine," UCI Machine Learning Repository, 2009, DOI: https://doi.org/10.24432/C53Y8H.
[65] M. Lichman *et al.*, "Breast Cancer," UCI Machine Learning Repository, 2013, DOI: https://doi.org/10.24432/C50P49.
[66] C. Sano, "Japanese Credit Screening," UCI Machine Learning Repository, 1992, DOI: https://doi.org/10.24432/C5259N.
[67] 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 A
TFHE PARAMETERS

Table III: 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 [58].

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

With the chosen TFHE parameters, each item exchanged between clients and the server—whether ciphertexts (LWE or RLWE) or keys (bootstrapping key (BSK), keyswitching key (KSK), or packing key (PFKSK))—has a fixed size. Table IV provides a summary of the sizes of these different objects.

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

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

APPENDIX B
REVOLUT OPERATIONS

A. Blind Read

Introduced as Blind Array Access in [22], the operation of reading an encrypted value in an encrypted array at an encrypted index was renamed as Blind Read in the RevoLUT paper [59] due to its extension to multidimensional arrays.

Algorithm 6: Blind Read operation (BlindRead)

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

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

B. Blind Write

The operation of adding an encrypted value to an encrypted array at an encrypted index was first introduced in [62] and renamed as Blind Write in the RevoLUT paper [59] due to its extension to multidimensional arrays.

Algorithm 7: Blind Write operation (BlindWrite)

Input : A LUT ciphertext $[\![M]\!]_{\text{LUT}}$, a LWE ciphertext $[\![i]\!]_{\text{LWE}}$ and a LWE ciphertext $[\![v]\!]_{\text{LWE}}$.
Output: The $[\![M]\!]_{\text{LUT}}$ with the $[\![i]\!]_{\text{LWE}}$-th element set to $[\![m_i + v]\!]_{\text{LWE}}$.

$[\![V]\!]_{\text{LUT}} \leftarrow \text{PFKS}([\![v]\!]_{\text{LWE}}, 0, \ldots, 0)$
$[\![M]\!]_{\text{LUT}} \leftarrow \text{BlindRotation}([\![i]\!]_{\text{LWE}}, [\![V]\!]_{\text{LUT}})$
$[\![M]\!]_{\text{LUT}} \leftarrow [\![M]\!]_{\text{LUT}} + [\![V]\!]_{\text{LUT}}$
return $[\![M]\!]_{\text{LUT}}$

C. Blind Argmax

Our algorithm described in Algorithm 8 computes the maximum value of a vector of BLWE ciphertexts is a single-pass scan with max/argmax update.

D. 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 LWE 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 ($[\![A]\!]_{\text{LUT}_{1,2}}$) at an encrypted index. Due to this reason, we developed the BAAT function, which is presented in Algorithm 10. 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 an encrypted data $[\![x]\!]_{\text{LWE}}$, this evaluate a function among multiple candidate functions homomorphically, while keeping both the selected branch and the data secret. This function is presented in Algorithm 9 with 3 cases but can be extended to p cases.

Algorithm 8: Blind Argmax of a vector of BLWE ciphertexts

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

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

Algorithm 9: Oblivious functional switch-case 3 (OSW)

Input : A selector $\ell = [\![s]\!]_{\text{LWE}}$. An encrypted data $[\![x]\!]_{\text{LWE}}$. A list of candidate functions (f_0, f_1, f_2) encrypted as LUT ciphertexts.
Output: The result of the selected function f_s applied to $[\![x]\!]_{\text{LWE}}$ i.e. $[\![f_s(x)]\!]_{\text{LWE}}$

$R \leftarrow \varnothing$
for $i \in \{0, 1, 2\}$ **do**
  $R \leftarrow R \cup \text{BlindRead}([f_i]_{\text{LUT}}, [\![x]\!]_{\text{LWE}})$
end
$[\![R]\!]_{\text{LUT}} \leftarrow \text{PFKS}(R)$
$r \leftarrow \text{BlindRead}([\![R]\!]_{\text{LUT}}, \ell)$
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.

1) Majority votes: Two Majority vote algorithms are presented in this section. The first one is used to compute the majority class in a leaf of a training tree (Algorithm 11 called TreeMajority), the second one is used to compute the majority class in a set of LWE ciphertexts (Algorithm 12 called Majority).

APPENDIX C
FULLY OBLIVIOUS PROTOCOL

In this section, we detail our Fully Oblivious Protocol in Algorithm 13 as presented in Section V-D.

Algorithm 10: Blind Array Add Ternary (BAAT)

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

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

// Get the element to change in the LUT

$[\![x]\!]_{\mathrm{BLWE}} \leftarrow \mathrm{BlindRead}([\![A]\!]_{\mathrm{LUT}_{1,2}}, [\![\ell]\!]_{\mathrm{LWE}})$

// Get the lower and higher parts of $[\![x]\!]_{\mathrm{BLWE}}$

$[\![x_l]\!]_{\mathrm{LWE}}, [\![x_h]\!]_{\mathrm{LWE}} \leftarrow Split([\![x]\!]_{\mathrm{BLWE}})$

// 1st round of switch-case

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

// 2nd round of switch-case

$L_1 \leftarrow [\![[0,\ldots,1]]\!]_{\mathrm{LUT}}$

$L_2 \leftarrow [\![[2,\ldots,0]]\!]_{\mathrm{LUT}}$

$t \leftarrow \mathrm{OSW}([\![\ell]\!]_{\mathrm{LWE}}, [\![x_l]\!]_{\mathrm{LWE}}, ([\![0]\!]_{\mathrm{LUT}}, L_1, L_2))$

// 3rd round of switch-case

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

// Update the LUT

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

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

Algorithm 11: Majority vote in a leaf (TreeMajority)

Input : A training tree $\mathcal{T}^{tr}$ with 2^d leaves $\mathcal{L}_i = ([\![c_0]\!]_{\mathrm{BLWE}}, \ldots, [\![c_{\gamma-1}]\!]_{\mathrm{BLWE}})$ 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 $\mathcal{L}_i = [\![c]\!]_{\mathrm{LWE}}$ in which c is the majority class.

// Get the majority class for each leaf

for $i \in [0, 2^d - 1]$ **do**

  $[\![c]\!]_{\mathrm{LWE}} \leftarrow \mathrm{BlindArgmax}(\mathcal{L}_i)$

  $\mathcal{T}^{inf}.\mathcal{L}_i \leftarrow [\![c]\!]_{\mathrm{LWE}}$

end

return $\mathcal{T}^{inf}$

Appendix D
Proofs

In the following, we prove the indistinguishability of the Training and Unlearning processes claimed in Theorem 1 and the indistinguishability of the Fully Oblivious Protocol claimed in Theorem 3. Indeed, to ensure that this indistinguishability holds, it is crucial that an adversary cannot discern *which* leaf was altered and *how* it was altered (*i.e.*, whether the leaf was incremented or decremented). Those two aspects are managed by the BAAT function. Specifically, the identification of the modified leaf is managed by both the BlindRead function and the BlindWrite function, while the nature of the modification is managed by the OSW function presented in Algorithm 9. Hence, we first prove that the output of the BAAT function is computationally indistinguishable from a random Nibble-Byte-LUT.

Lemma 1 (Indistinguishability of BAAT output)**.** Let

Algorithm 12: Majority vote of a set of LWE ciphertexts (Majority)

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

Output: A majority class c

// Get the count of each class

$R \leftarrow \mathrm{BlindCount}(C)$

// Get the majority class

$c \leftarrow \mathrm{BlindArgmax}(R)$

return c

Algorithm 13: Fully Oblivious Protocol

Input : One or more samples $\mathcal{D} = \left[\left(F^i, L^i\right)\right]_{i=1}^{n}$ in which $F^i = ([\![F^i_0]\!]_{\mathrm{RLWE}}, \ldots, [\![F^i_{k-1}]\!]_{\mathrm{RLWE}})$ 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 $\mathcal{F} = (\mathcal{T}^{tr}_0, \ldots, \mathcal{T}^{tr}_{m-1}) \cup (\mathcal{T}^{inf}_0, \ldots, \mathcal{T}^{inf}_{m-1})$ and a current state $j \in [0, \beta - 1]$ before the next update of $(\mathcal{T}^{inf}_i)_{i=0}^{m-1}$.

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

$C \leftarrow \varnothing$

$i \leftarrow j$

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**

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

    end

    $i \leftarrow 0$

  end

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

  $R \leftarrow \varnothing$

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

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

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

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

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

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

  end

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

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

  $i \leftarrow i + 1$

end

return C

$[\![A]\!]_{\mathrm{LUT}_{1,2}}$ be a Nibble-Byte-LUT encrypting a vector $A = (a_i)_{i=0}^{2^d-1}$ in which each $a_i \in \{0x00, \ldots, 0xFF\}$. Let $[\![\ell]\!]_{\mathrm{LWE}}$ be an encrypted index and $[\![b]\!]_{\mathrm{LWE}}$ an encrypted ternary value belonging to $\{0, 1, 2\}$. Then the output of $\mathrm{BAAT}([\![A]\!]_{\mathrm{LUT}_{1,2}}, [\![\ell]\!]_{\mathrm{LWE}}, [\![b]\!]_{\mathrm{LWE}})$ is computationally indistinguishable from a random Nibble-Byte-LUT.

Proof of Lemma 1. We argue by a standard sequence of hybrid games against a PPT distinguisher $\mathcal{D}'$ that receives the encrypted output Nibble-Byte-LUT.

- Game H_0 (real): $\mathcal{D}'$ is given the real output of $\mathrm{BAAT}([\![A]\!]_{\mathrm{LUT}_{1,2}}, [\![\ell]\!]_{\mathrm{LWE}}, [\![b]\!]_{\mathrm{LWE}})$.
- Game H_1 (randomize b): Replace $[\![b]\!]_{\mathrm{LWE}}$ by $[\![\tilde{b}]\!]_{\mathrm{LWE}}$ in which $\tilde{b} \leftarrow \{0, 1, 2\}$ is uniform. By IND-CPA of LWE, $H_0 \approx H_1$.
- Game H_2 (randomize ℓ): Replace $[\![\ell]\!]_{\mathrm{LWE}}$ by $[\![\tilde{\ell}]\!]_{\mathrm{LWE}}$ in which $\tilde{\ell} \leftarrow \mathbb{Z}_p$ is uniform. Because $\mathrm{BlindRead}$, OSW, and $\mathrm{BlindWrite}$ apply homomorphic operations that re-randomize ciphertexts, all intermediate and final ciphertexts produced by BAAT are distributed as fresh encryptions independent of the concrete values of ℓ and of the internal selection path. Hence $H_1 \approx H_2$.
- Game H_3 (randomize $[\![A]\!]_{\mathrm{LUT}_{1,2}}$) : Sample $\tilde{A} \leftarrow \{0x00, \ldots, 0xFF\}^{2^d}$ uniformly and encrypt it to get $[\![\tilde{A}]\!]_{\mathrm{LUT}_{1,2}}$. Replace the input ciphertext $[\![A]\!]_{\mathrm{LUT}_{1,2}}$ by $[\![\tilde{A}]\!]_{\mathrm{LUT}_{1,2}}$. By IND-CPA security, this single replacement is indistinguishable, therefore $H_2 \approx H_3$.

In H_3, the BAAT pipeline operates on a uniformly random packed Nibble-Byte-LUT and on independent uniform $\tilde{\ell}, \tilde{b}$. By the re-randomizing nature of the subroutines, the ciphertext of the output Nibble-Byte-LUT is distributed as a fresh encryption that is independent of the original (A, ℓ, b). Thus, the distribution seen by $\mathcal{D}'$ in H_3 is that of an encryption of a uniformly random Nibble-Byte-LUT. Chaining the hybrid steps yields $H_0 \approx H_3$, completing the proof. □

Proof of Theorem 1. Let $\mathcal{F}$ be a forest of trees $\mathcal{F} = (\mathcal{T}_0^{tr}, \ldots, \mathcal{T}_{m-1}^{tr})$ trained on a dataset $\mathcal{D}$. Let (F, L) be a sample (in $\mathcal{D}$ or not) and $\tilde{\mathcal{F}} = (\tilde{\mathcal{T}}_0^{tr}, \ldots, \tilde{\mathcal{T}}_{m-1}^{tr})$ be the forest after calling Algorithm 4 with $\mathcal{F}$ and (F, L). Then, by Lemma 1, since the leaves of the forest $\tilde{\mathcal{F}}$ are the output of the BAAT function applied to the leaves of the forest $\mathcal{F}$ and the sample (F, L) where L contains the ternary value to add to the leaves of the forest $\mathcal{F}$, the leaves $(\tilde{\mathcal{T}}_i^{tr}.\mathcal{L}_j)_{i=0,j=0}^{m-1,\gamma-1}$ of the forest $\tilde{\mathcal{F}}$ are indistinguishable from a random set of leaves for an adversary that sees the encrypted output of the training and unlearning processes. □

Proof of Theorem 3. Let $\mathcal{F}$ be a forest of two types of trees $\mathcal{F} = (\mathcal{T}_0^{tr}, \ldots, \mathcal{T}_{m-1}^{tr}) \cup (\mathcal{T}_0^{inf}, \ldots, \mathcal{T}_{m-1}^{inf})$ trained on a dataset $\mathcal{D}$. Let (F, L) encoded and encrypted as in Equation 1. Let j the current state of the forest $\mathcal{F}$ before the next update of $(\mathcal{T}_i^{inf})_{i=0}^{m-1}$. Let $\tilde{\mathcal{F}} = (\tilde{\mathcal{T}}_0^{tr}, \ldots, \tilde{\mathcal{T}}_{m-1}^{tr}) \cup (\tilde{\mathcal{T}}_0^{inf}, \ldots, \tilde{\mathcal{T}}_{m-1}^{inf})$ be the forest after calling Algorithm 13 with $\mathcal{F}$ and (F, L). Two cases can arise:

- If the sample (F, L) is for a training or unlearning query, then the server blindly updates the corresponding leaves in $\mathcal{T}_i^{tr}$ *and* blindly reads the corresponding leaves in $\mathcal{T}_i^{inf}$. Since, the $\mathrm{BlindRead}$ re-randomizes the ciphertexts, the forest leaves of $\tilde{\mathcal{F}}$ are indistinguishable from a random set of leaves.
- If the sample (F, L) is for an inference query, then L is a vector of $[\![0]\!]_{\mathrm{LWE}}$ according to Equation 1. Therefore the server blindly reads the corresponding leaves in $\mathcal{T}_i^{inf}$ *and* blindly adds 0 to the corresponding leaves in $\mathcal{T}_i^{tr}$ using the BAAT function. Since, the $\mathrm{BlindRead}$ and BAAT re-randomizes the ciphertexts, the forest leaves of $\tilde{\mathcal{F}}$ are indistinguishable from a random set of leaves.

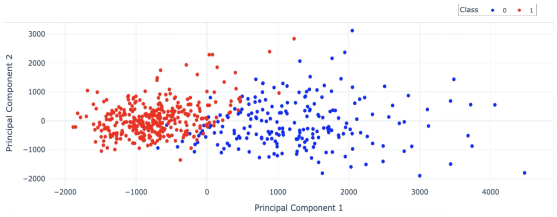
□

Appendix E
Operations Running Time

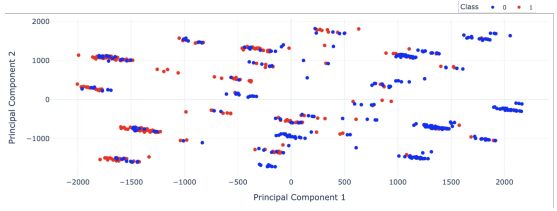
We show in Table V the average time in seconds for the main operations in our protocol for the three datasets used in the experiments. This shows the impact of the dataset characteristics (number of features, number of classes, etc.) on the protocol runtime.

Appendix F
PCA of the Datasets

The Figure 8 displays 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.



(a) Cancer dataset



(b) Japanese Credit Screening dataset

Fig. 8. Principal Component Analysis (PCA) in 2D of the Cancer and Credit datasets.

Table V: Average running 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 $\mathcal{F}$.

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