SILENTWOOD: EFFICIENT PRIVATE INFERENCE OVER GRADIENT BOOSTING DECISION FORESTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Gradient boosting decision forests, used by XGBoost or AdaBoost, offer higher accuracy and lower training times than decision trees on large datasets. Private inference protocols for decision trees can preserve both input and tree privacy. However, naively extending them to decision forests by replication leads to impractical running times. In this paper, we propose an efficient private decision inference protocol using homomorphic encryption. We present several optimizations that identify and remove (approximate) duplication between trees, significantly reducing communication and computation costs over the naive approach. We present the private inference protocol for highly scalable gradient boosting decision forests. Our protocol (SilentWood) is faster than parallel RCC-PDTE by up to 42.5x, Zama's Concrete ML XGBoost by up to 27.8x, and SoK-GGG's two-party garbled circuit protocol by 2.94x.

1 INTRODUCTION

Machine Learning as a Service (MLaaS) allows large user groups to use machine learning models. A service provider trains a model and offers users inference using this model. Some of such examples include recommendation systems, spam filtering, and object detection used in e-commerce services, social networking services, and smart home. An advantage of such MLaaS is that the provider can control access to the model and easily update it when new training data is available. In contrast, its disadvantage is the privacy concern of the users revealing their test data to the provider which may be sensitive or legally restricted. Cryptography offers a compelling solution to this problem. For example, the provider can use fully homomorphic encryption Gentry (2009b); Fan & Vercauteren (2012b) to *blindly* process the user's input features in an encrypted form through the trained model and yield an encrypted prediction result, which only the user can decrypt. However, this solution usually has high computational costs. Alternatively, the user and the provider can engage in a two-party computation Yao (1982; 1986); Oded (2009) where the user's input is computed through the model in a *blinded* manner, during which the server's trained model parameters and the user's private data are not leaked to each other. However, a two-party computation usually incurs high communication costs.

In this paper, we develop a new protocol for private inference over gradient boosting decision forests, which are collections of decision trees whose results are combined into the final model prediction. Gradient boosting forests offer a better trade-off between higher accuracy and lower training time than training a single decision tree for large datasets. Moreover, they retain a critical advantage of decision trees over many other machine learning techniques; their predictions are explainable, helping to fulfill legal, social, and ethical requirements.

Many private inference protocols over decision trees exist (§5). However, extending these to gradient boosting decision forests comes with two challenges. First, the results of the decision trees need to be combined into a single prediction result, requiring further computation. This can particularly pose a problem for protocols using homomorphic encryption, since these computations increase the multiplication overhead, significantly increasing the runtime of the protocols. Second, even naively running several independent decision tree protocols without combining their results can exhaust the server's computing resources. Therefore, optimizations are necessary to reduce the cost of executing multiple decision tree evaluations on the same input.

To solve these problems, we propose SilentWood, a private inference protocol for gradient boosting that uses three novel techniques. First, to reduce the high computation overhead caused by frequent homomorphic computations in FHE-based PDTE (private decision tree evaluation), SilentWood applies

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

node clustering (i.e., grouping and weighted-averaging tree nodes having similar thresholds) to eliminate redundant computations (§3.1). Second, SilentWood proposes the Blind Code Conversion (BCC) protocol to combine SumPath's depth-independent multiplicative cost with MultiplyPath's arithmetic compatibility for score aggregation (§3.2). BCC is a lightweight two-party protocol without garbled circuits or secret sharing: the server pads and shuffles its intermediate ciphertext so its plaintext looks uniformly random to the client. The client's role is to decrypt it and convert the underlying plaintext values *blindly* to help the server's subsequent computation of score aggregation. This process ensures the information in the intermediate ciphertext to be inaccessible from both the client and server, while realizing arithmetic compatibility of the server's score aggregation. Lastly, to reduce communication overhead caused by repetitive data encoding in FHE-based PDTE, SilentWood's ciphertext compression carefully removes repetitive encodings before encryption and then homomorphically restores the original data format on the server side (i.e., homomorphic decompression), thus minimizing ciphertext size while preserving necessary SIMD comparisons (§3.3).

While RCC-PDTE Akhavan Mahdavi et al. (2023) (§B.3), the state-of-the-art PDTE protocol, is effective for computing a single decision tree, naively repeating this protocol for all decision trees (100 in our default setup) can take more than 5 minutes for a single inference, which is prohibitively long. The inference time of SilentWood is, on average, faster than this baseline by 42.5x, faster than Zama's Concrete ML Zama (2024) XGBoost by 34.0x, and faster than SoK-GGG (i.e., a state-of-the-art MPC protocol based on GC) by 2.94x (§4). Our paper's contributions are as follows:

- We propose three optimization techniques for efficient private inference of random forest models: node clustering, blind code conversion, and ciphertext compression.
- We implemented SilentWood, a prototype of our proposed techniques.
- We provide evaluation results of SilentWood and compare it to other state-of-the-art protocols.

2 PRELIMINARIES

2.1 GRADIENT BOOSTING MODELS

Ensemble decision trees Bishop (2006) combine many shallow trees (weak learners) to improve prediction, typically via either random forests Breiman (2001) or gradient boosting Friedman (2001). Random forests train trees independently and aggregate their outputs (e.g., voting/averaging), whereas gradient boosting trains trees sequentially to correct prior residual errors, yielding a strong additive model. We focus on private inference for AdaBoost Freund & Schapire (1995) and XGBoost Chen & Guestrin (2016). In AdaBoost, each tree has a weight and each leaf outputs a signed value ((+)/(-)); inference traverses every tree by threshold comparisons, retrieves the reached leaf value, multiplies it by the tree weight, sums across trees, and predicts by the sign of the final sum. XGBoost similarly sums leaf values of traversed trees: for binary classification it applies a sigmoid to the sum and thresholds at 0.5; for multi-class it computes a per-class sum, applies softmax, and outputs the argmax.

2.2 FULLY HOMOMORPHIC ENCRYPTION (FHE)

Fully Homomorphic Encryption (FHE) Gentry (2009a) enables addition and multiplication directly on ciphertexts, allowing computation on encrypted data without decryption. SilentWood adopts an RLWE-family scheme, BFV Fan & Vercauteren (2012a), primarily for efficient SIMD batching: an n-dimensional plaintext vector is encoded (Encode) into a $(n-1)$-degree (or lesser degree) polynomial $M(X)$, then encrypted (Encrypt) under a secret polynomial $S(X)$ to an RLWE ciphertext (with a large scaling factor Δ as part of the scheme). The scheme supports cipher-plaintext and cipher-cipher addition/multiplication (PAdd/CAdd/PMult/CMult) as well as slot rotations (Rotate) that cyclically permute packed plaintext elements without decryption. SIMD means these operations act element-wise over all packed slots at once; typical slot counts are $2^{13} \sim 2^{15}$, providing substantial parallelism. We will treat these FHE primitives as black boxes to implement private gradient-boosting inference. The supported operations of the BFV scheme are detailed in §A.

2.3 PRIVATE DECISION TREE EVALUATION

Private Decision Tree Evaluation (PDTE) is a protocol between a server holding a decision tree and a client holding a feature vector for one sample. At the end of PDTE, the server learns nothing

about the client's inputs, while the client learns nothing about the server's model beyond the prediction result and model hyper-parameters (e.g., feature/class types)[1]. The server typically returns class scores (probabilities), from which the client selects the top class (or applies a tie-breaking rule). PDTE generally consists of two steps: (i) value comparison at each node; and (ii) tree path evaluation.

Value Comparison at Each Node: Tree traversal requires *greater-than* comparisons between an input feature X and a node's threshold Y. Under FHE, only two arithmetic operations $(+, \cdot)$ are natively supported, so comparisons must be implemented as an arithmetic circuit over $(+, \cdot)$. A direct approach is to bitwise-decompose X, Y into encrypted bits and *homomorphically* evaluate a standard comparator circuit, where all logic gates are implemented as bit-wise $(+, \cdot)$ operations of encrypted bits. This approach computationally requires a cipher-cipher multiplication (CMult) chain as long as the bit length of X, Y. In FHE, cipher-cipher multiplication is computationally heavy, so shortening the multiplication chain is crucial. To this end, a more efficient method of value comparison is based on constant-weight (CW) encoding Akhavan Mahdavi et al. (2023), where numbers are encoded using a fixed Hamming weight, and the CMult chain's length for value comparison is equivalent to the Hamming weight. While SilentWood is compatible with both methods, our prototype adopts the CW-based one. We explain the details of these two comparison methods in §B.3.

Tree Path Evaluation: Given encrypted comparison results at all nodes (1 if > is satisfied; 0 otherwise), PDTE's next step is to determine the reached leaf by evaluating path conditions. Two common methods are MultiplyPath (path conjugation) Kiss et al. (2019) and SumPath Tueno et al. (2019). As illustrated in Figure 2 (§B.2), MultiplyPath assigns each edge a 0/1 indicator and multiplies indicators along a root-to-leaf path, so only the true path yields 1 (others 0). However, this requires a cipher-cipher multiplication (CMult) chain length proportional to tree depth. On the other hand, SumPath assigns 0 to satisfied edges and 1 to unsatisfied edges and sums along each path, so the true path yields 0 and every other path yields its number of unsatisfied edges ($1 \sim d$); each sum is then multiplied by a fresh random non-zero value to hide the count. This needs only ciphertext additions and one plaintext multiplication, which is substantially cheaper under FHE. After homomorphically computing the values for all tree paths, the server returns them to the client, who decrypts and selects the unique leaf whose SumPath equals 0 (or MultiplyPath equals 1) to obtain the prediction. We provide an example of tree path evaluation based on MultiplyPath and SumPath in Figure 5 (§B.4).

3 METHODOLOGY

This section explains the problems of applying existing PDTE methods to evaluate private gradient boosting models, and how SilentWood solves them through three novel techniques: node clustering, blind code conversion, and ciphertext compression.

Threat Model: Our threat model for privacy assumes a semi-honest client (i.e., follows the protocol) and a malicious server (i.e., may deviate from the protocol)[2]. They try to learn about each other's secrets from exchanged messages. The server's secret is its trained gradient boosting model; the client's secret is its input feature values. Their common knowledge is the feature names and feature bit-length. At the end of the protocol, the server learns nothing about the client's features; the client learns only the inference scores/probabilities for each class and the fact that the number of tree paths is capped at some multiple of the cyclotomic polynomial's degree (e.g., 2^{14}) as a hyperparameter. Besides that, the client learns nothing about the server's model (e.g., node/tree count, depth, or node thresholds).

3.1 NODE CLUSTERING

Problem of Many Tree Nodes and Paths. Existing PDTE protocols homomorphically compare each internal node's threshold Y with the client's encrypted feature value X by homomorphically evaluating the value comparison logic for $X > Y$ (see Figure 4), which entails expensive CMult, PMult, and Rotate operations. This cost becomes a substantial bottleneck for private inference over gradient-boosted forests, since such models typically comprise large ensembles (e.g., 100~1000 trees), involving orders of magnitude more nodes and paths than a single decision tree.

[1]More precisely, PDTE's server-side privacy goal is to reveal no more model information than plaintext inference.
[2]For assuming a malicious client and a malicious server, see §D

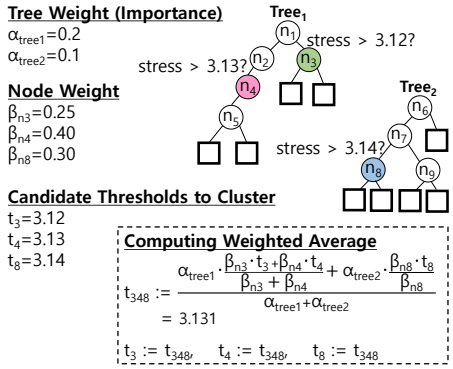

Figure 1: An example of node clustering

Algorithm 1 Accuracy-preserving merge

Input: $\mathcal{N}$, s ▷ nodes, clustering intensity
repeat
  $changed \leftarrow$ **false**
  for $n_i \in \mathcal{N}$ **do**
    $\mathcal{C} \leftarrow \left\{ \begin{array}{l} n \in \mathcal{N} \mid n.name = n_i.name \wedge \\ \quad \frac{|n.threshold - n_i.threshold|}{\textbf{MAX}(n_i.name) - \textbf{MIN}(n_i.name)} \leq s \end{array} \right\}$
    if $|\{n.threshold \mid n \in \mathcal{C}\}| > 1$ **then**
      $new_threshold \leftarrow$ WeightedAverage($\mathcal{C}$)
      for $n \in \mathcal{C}$ **do**
        $n.threshold \leftarrow new_threshold$
      end for
      $changed \leftarrow$ **true**
    end if
  end for
until $\neg changed$ ▷ stop when no more clustering occurs
Output: $\mathcal{N}$ ▷ clustered final tree nodes

SilentWood's insight is to homomorphically compute the comparison logic once per distinct ⟨*feature*,*threshold*⟩ value pair across all trees and reuse them for each group of nodes having the same value pair. This helps cut down the computation overhead for the nodes whose thresholds are exactly identical. To further reduce the overhead, SilentWood clusters nodes that share the same feature name and *similar* thresholds, then replaces the thresholds in the cluster with their *weighted* average: weighted by the tree weight α and the node weight β. In Figure 1's example, nodes n_3, n_4, n_8 (e.g., stress level) have close thresholds: 3.14, 3.12, and 3.13. SilentWood clusters these nodes together and updates their thresholds to 3.131 as their weighted average (dashed-box formula). Now their comparisons are identical, so we compute the comparison for these 3 nodes once instead of three times. Meanwhile, since changing thresholds can alter accuracy, SilentWood addresses this by using: (i) clustering intensity adjustment, (ii) weighted averaging, (iii) leaf value re-optimization; and (iv) accuracy-preserving merge.

Clustering Intensity: With learnable intensity s, we co-cluster node n_i and n_j if they share the same feature name and their values are close enough such that $\frac{|n_i.threshold - n_j.threshold|}{\textbf{MAX}(n_i.name) - \textbf{MIN}(n_i.name)} \leq s$, where $\textbf{MAX}(n_i.name)$ and $\textbf{MIN}(n_i.name)$ are the feature i's max/min value over the whole training data. The optimal s is dataset-dependent.

Weighted Threshold Average: For a given node cluster $\mathcal{C}$, we update the node thresholds in it to a weighted average by using tree weight α and node weight β. The high-level idea is that more influential nodes/trees should change their thresholds less to preserve accuracy. To compute tree i's weight α_i, we first compute w_i, the sum of hit absolute leaf values over all training samples on tree i, and normalize it over all T trees in the model as $\alpha_i = \frac{w_i}{\sum_{k=1}^{T} w_k}$. We compute node j's weight β_j as the fraction of information gain of node j among all nodes in $\mathcal{C}$ that belong to this tree. Using all α_i and β_j, we scale thresholds and get the weighted average as Equation (1).

$$t_{average} = \frac{\sum_{i=1}^{T} \alpha_i \cdot \frac{\sum\{\beta_j \cdot t_j \mid n_j \in Tree_i \wedge n_j \in \mathcal{C}\}}{\sum\{\beta_j \mid n_j \in Tree_i \wedge n_j \in \mathcal{C}\}}}{\sum_{i=1}^{T} \alpha_i} \tag{1}$$

Leaf Value Re-optimization: After updating thresholds for a cluster $\mathcal{C}$, we re-optimize the leaf values (scores) while keeping the tree topology fixed. Concretely, we re-route the training samples under the updated thresholds and, for each leaf ℓ, aggregate the first-order and second-order loss derivatives with respect to the raw prediction $\hat{y}(x)$: $G_\ell = \sum_{x \in \ell} \frac{\partial}{\partial \hat{y}(x)} \mathcal{L}\big(y(x), \hat{y}(x)\big)$, $H_\ell = \sum_{x \in \ell} \frac{\partial^2}{\partial \hat{y}(x)^2} \mathcal{L}\big(y(x), \hat{y}(x)\big)$. We then apply a regularized Newton update to the leaf value: $v_\ell \leftarrow v_\ell - \eta \cdot \frac{G_\ell}{H_\ell + \lambda}$, where λ is the ℓ_2 regularization coefficient on leaf values and η is the learning rate. This update recalibrates the existing leaf outputs under the modified decision boundaries induced by threshold averaging.

Accuracy-preserving Merge: Once the candidate cluster $\mathcal{C}$'s thresholds are weight-averaged and leaves are re-calibrated as described in Algorithm 1, we decide whether to commit/abort it. Basically, we evaluate the updated model on the validation set: if validation accuracy is unchanged or improved, we commit; otherwise we abort. We repeat this in a bottom-up manner until no more merge-able

Notations	
N	: The number of plaintext slots in an RLWE ciphertext
$\mathbb{Z}_p, \mathbb{Z}_p^*$	: The plaintext space $\{0, \cdots, p-1\}$ (p prime); $\mathbb{Z}_p$ without 0
$\mathcal{U}_p$	: A custom conversion table which defines the mapping of each plain value $\{0, \cdots, p-1\}$ to some $\{u_0, u_1, \cdots, u_{p-1}\}$
f	: The public zero/non-zero distribution of $\vec{w}$
f_{zero}	: The number of 0s in each SumPath list as hyperparameter
$\vec{v}^{(i)}$	: An array whose slot index 0 stores the i-th path cost c_i
$\vec{\rho}$	: A mask array uniformly drawn from $\mathbb{Z}_p^*$
$\vec{u}$	: An intermediate array that stores shuffled SumPath values
$\vec{w}$	: A shuffled array of f_{zero} 0s and $(N - f_{zero})$ uniform non-zero values
$\vec{c}$	: The client converts $\vec{w}$ into $\vec{c}$ according to table $\mathcal{U}_p$
$\langle\!\langle \vec{v}^{(i)} \rangle\!\rangle$	: A ciphertext encrypting $\vec{v}^{(i)}$ ▷ the same applies to: $\vec{u}, \vec{w}, \vec{c}$

Protocol	
Input	: $\{\langle\!\langle \vec{v}^{(i)} \rangle\!\rangle\}_{i=1}^{\ell}$ ▷ a list of ℓ ciphertexts storing all ℓ SumPaths
Server	: 1) Randomly rotate each $\{\langle\!\langle \vec{v}^{(i)} \rangle\!\rangle\}_{i=1}^{\ell}$ and merge as $\langle\!\langle \vec{u} \rangle\!\rangle$.
	2) $\langle\!\langle \vec{u} \rangle\!\rangle \leftarrow \mathsf{PMult}(\langle\!\langle \vec{u} \rangle\!\rangle, \mathsf{Encode}(\vec{\rho}))$, $\vec{\rho} \xleftarrow{\$} (\mathbb{Z}_p^*)^N$.
	3) Create a dummy array $\vec{d}$ (non-zero entries $\xleftarrow{\$} \mathbb{Z}_p^*$) such that the array $\vec{u} + \vec{d}$ has exactly f_{zero} zeros
	4) $\langle\!\langle \vec{w} \rangle\!\rangle \leftarrow \mathsf{PAdd}(\langle\!\langle \vec{u} \rangle\!\rangle, \mathsf{Encode}(\vec{d}))$.
S→C	: Send $\langle\!\langle \vec{w} \rangle\!\rangle$ and $\mathcal{U}_p$.
Client	: 1) Decrypt $\langle\!\langle \vec{w} \rangle\!\rangle$ to $\vec{w}$.
	2) Create $\vec{c}$ by converting each value in $\vec{w}$ according to $\mathcal{U}_p$.
	3) Encrypt $\vec{c}$ to $\langle\!\langle \vec{c} \rangle\!\rangle$.
C→S	: Send $\langle\!\langle \vec{c} \rangle\!\rangle$.
Output	: $\langle\!\langle \vec{c} \rangle\!\rangle$

Table 1: The blind code conversion (BCC) protocol

thresholds. Our evaluation (Table 4) shows that a high s does not reduce validation accuracy because accuracy-degrading merges are rejected.

Applying Node Clustering to MPC: Fewer distinct comparisons reduce cost in encrypted inference beyond FHE. In MPC, comparisons are often dominant; e.g., Table 15 (§F) shows GC-based XGBoost spends 84.1% time on comparisons. Thus, node clustering can directly lower MPC runtime and is compatible with both FHE and MPC.

3.2 Blind Code Conversion (BCC)

Problem of Final Score Aggregation. In a gradient boosting forest, the final prediction score is the sum of leaf values on the *true* paths of all trees. When it comes to private evaluation of this, the computation should be compatible with FHE arithmetic $(+, \cdot)$ and also efficient. Unfortunately, existing PDTE methods (§2.3) do not meet both. With SumPath Akhavan Mahdavi et al. (2023), a leaf's SumPath is 0 on the true path and a masked non-zero value $\rho_j c_j$ otherwise (§2.3), but there is no low-depth arithmetic expression over $(+, \cdot)$ to express the logic: $\mathsf{SumPath{==}0\ ?\ leafValue : 0}$. On the other hand, MultiplyPath Cong et al. (2022) can compatibly express this logic as $\mathsf{MultiplyPath} \cdot \mathsf{LeafValue}$, but it requires a cipher-cipher multiplication chain (CMult) as long as the maximum depth of all tree paths, which is computationally expensive.

SilentWood's novelty is resolving the dilemma of SumPath and MultiplyPath by introducing a light-weight two-party blind code conversion (BCC) protocol (consuming 1 RTT), as formally described in Table 1. This protocol essentially lets the client blindly convert SumPath values into MultiplyPath values for arithmetic compatibility. Initially (after path evaluation), the server packs the encrypted path costs into a list of intermediate ciphertexts ($\{\langle\!\langle \vec{v}^{(i)} \rangle\!\rangle\}_{i=1}^{\ell}$), homomorphically (blindly) shuffles their slots, merges them as a single ciphertext $\langle\!\langle \vec{u} \rangle\!\rangle$, applies the per-leaf masks $\vec{\rho}$ with a single PMult, and homomorphically pads it with dummy values to produce the final uniformly shuffled ciphertext $\langle\!\langle \vec{w} \rangle\!\rangle$. At this point, the plaintext of $\langle\!\langle \vec{w} \rangle\!\rangle$ has distribution f: a uniform mixture of zero : non-zero ratio of 1:1. Since p is prime and $1 \le c_j \le d < p$, every false-path value $\rho_j c_j$ is non-zero and uniform over $\mathbb{Z}_p^*$, so exactly the T true paths are zero (§D). Next, the server sends $\langle\!\langle \vec{w} \rangle\!\rangle$ with a conversion table $\mathcal{U}_p$ to the client, who in turn decrypts it, converts $0 \mapsto 1$ and non-zero $\mapsto 0$, re-encrypts them as $\langle\!\langle \vec{c} \rangle\!\rangle$, and returns it to the server. Finally, the server multiplies (PMult) the received ciphertext $\langle\!\langle \vec{c} \rangle\!\rangle$ with a plaintext leaf-score array, whose slot indices are aligned with the shuffled ones in $\langle\!\langle \vec{c} \rangle\!\rangle$, and aggregates per-class prediction score by homomorphic rotations/additions (Rotate, CAdd).

We provide an example of the BCC protocol in §C.1. BCC contributes to the inference speedup by 19.3x (Table 2, Table 3)

Blind Shuffling: Algorithm 2 describes how the server homomorphically pads and shuffles SumPaths according to a random permutation plan $\vec{\sigma}$ whose value distribution is f (line 3). Following the random $\vec{\sigma}$, the server accordingly rotates each real path cost (line 6~9), masks the merged ciphertext (line 10), adds randomly placed dummy SumPaths (line 11~14), and compactly merges all of them in the final ciphertext $\langle\!\langle \vec{w} \rangle\!\rangle$. Once the client receives $\langle\!\langle \vec{w} \rangle\!\rangle$ and returns its converted (i.e, $0 \mapsto 1$, non-zero $\mapsto 0$) version $\langle\!\langle \vec{c} \rangle\!\rangle$, the server homomorphically filters and aggregates the valid leaf scores in it as described in Algorithm 3: the server multiplies $\langle\!\langle \vec{c} \rangle\!\rangle$ with the slot-aligned plaintext leaf score array $\vec{x}$ to homomorphically zero out the leaf scores for false paths as well as dummy SumPaths (line 2), then sums the filtered scores and places them at slot 0 of $\langle\!\langle \vec{z} \rangle\!\rangle$ (line 3~9).

Algorithm 2 Blind Shuffling

Input: T, $\{\langle\!\langle \vec{v}^{(i)} \rangle\!\rangle\}_{i=1}^{\ell}$, $\{s_i\}_{i=1}^{\ell}$
▷ total number of trees, encrypted path costs, plain leaf scores
$\vec{\sigma} \leftarrow$ *genDist(f)* ▷ An offset list for random N-slot permutation
$\vec{x} \leftarrow 0$, $\vec{d} \leftarrow 0$ ▷ leaf-score array ▷ dummy-SumPath array
$\langle\!\langle \vec{u} \rangle\!\rangle \leftarrow \langle\!\langle [0, 0, \ldots, 0] \rangle\!\rangle$ ▷ a ciphertext of shuffled SumPaths
for i **in** $\{1, 2, \ldots, \ell\}$ ▷ permute each i-th SumPath **do**
  $\langle\!\langle \vec{u} \rangle\!\rangle \leftarrow \mathsf{CAdd}(\langle\!\langle \vec{u} \rangle\!\rangle, \mathsf{Rotate}(\langle\!\langle \vec{v}^{(i)} \rangle\!\rangle, \sigma_i))$
  $x_{\sigma_i} \leftarrow s_i$ ▷ also, store the slot-aligned leaf score
end for
$\vec{\rho} \xleftarrow{\$} (\mathbb{Z}_p^*)^N$
$\langle\!\langle \vec{u} \rangle\!\rangle \leftarrow \mathsf{PMult}(\langle\!\langle \vec{u} \rangle\!\rangle, \mathsf{Encode}(\vec{\rho}))$ ▷ $\rho_j c_j \neq 0$ iff false path
for i **in** $\{\ell + 1, \ldots, N - (f_{zero} - T)\}$ ▷ f_{zero}: freq. of 0s **do**
  $d_{\sigma_i} \xleftarrow{\$} \mathbb{Z}_p^*$ ▷ permute dummy non-zero SumPaths
end for
$\langle\!\langle \vec{w} \rangle\!\rangle \leftarrow \mathsf{PAdd}(\langle\!\langle \vec{u} \rangle\!\rangle, \mathsf{Encode}(\vec{d}))$ ▷ add dummy SumPaths
Output: $\langle\!\langle \vec{w} \rangle\!\rangle, \vec{x}$ ▷ shuffled SumPaths, slot-aligned leaf scores

Algorithm 3 Blind Score Aggregation

Input: $\langle\!\langle \vec{c} \rangle\!\rangle, \vec{x}$ ▷ client-converted SumPaths, aligned leaf scores
$\langle\!\langle \vec{z} \rangle\!\rangle \leftarrow \mathsf{PMult}(\langle\!\langle \vec{c} \rangle\!\rangle, \mathsf{Encode}(\vec{x}))$ ▷ zero out false-path scores
for n **in** $\{\log_2 N, (\log_2 N) - 1, \ldots, 2, 1\}$ **do**
  $\vec{m}^{(1)} \leftarrow [\overbrace{1, \ldots, 1}^{2^{n-1}\text{ slots}}, \overbrace{0, \ldots, 0}^{2^{n-1}\text{ slots}}, \overbrace{0, \ldots, 0}^{N-2^n\text{ slots}}]$ ▷ 1st 2^{n-1} slot filter
  $\vec{m}^{(2)} \leftarrow [\overbrace{0, \ldots, 0}^{2^{n-1}\text{ slots}}, \overbrace{1, \ldots, 1}^{2^{n-1}\text{ slots}}, \overbrace{0, \ldots, 0}^{N-2^n\text{ slots}}]$ ▷ 2nd 2^{n-1} slot filter
  $\langle\!\langle \vec{z}^{(1)} \rangle\!\rangle = \mathsf{PMult}(\langle\!\langle \vec{z} \rangle\!\rangle, \mathsf{Encode}(\vec{m}^{(1)}))$
  $\langle\!\langle \vec{z}^{(2)} \rangle\!\rangle = \mathsf{PMult}(\langle\!\langle \vec{z} \rangle\!\rangle, \mathsf{Encode}(\vec{m}^{(2)}))$
  $\langle\!\langle \vec{z} \rangle\!\rangle = \mathsf{CAdd}(\langle\!\langle \vec{z}^{(1)} \rangle\!\rangle, \mathsf{Rotate}(\langle\!\langle \vec{z}^{(2)} \rangle\!\rangle, 2^{n-1}))$ ▷ align, add
end for
Output: $\langle\!\langle \vec{z} \rangle\!\rangle$ ▷ slot 0 stores the aggregated prediction score

Algorithm 4 Ciphertext Compression

Input: $\{A_0, \ldots, A_{M-1}\}, \{start_j\}, \{gap_0, \ldots, gap_{R-1}\}, R$
▷ M data arrays (one per encoded bit), feature j's start slot, r-th repetition offset, repetitions per ciphertext
for $k \in \{0, \ldots, \lfloor M/R \rfloor - 1\}$ ▷ every R arrays share one ciphertext **do**
  $P \leftarrow [0, \ldots, 0]$ ▷ one compact plaintext array
  for $r \in \{0, \ldots, R-1\}$ ▷ r-th array of the group **do**
    $P[start_j + gap_r] \leftarrow A_{kR+r}[j]$ **for all** j ▷ the slot a repetition would have used now holds another array
  end for
  $C_k \leftarrow \mathsf{Encrypt}(\mathsf{Encode}(P), sk)$
end for
Output: $\{C_0, \ldots, C_{\lfloor M/R \rfloor - 1}\}$ ▷ M/R ciphertexts instead of M

BCC is computationally efficient because it requires no cipher-cipher multiplication (CMult) chain. Unlike FHE+GC hybrids, it needs no garbled circuits or secret sharing: one RTT with a few packed FHE ciphertexts suffices even for hundreds of trees, since extra paths only fill spare slots. For security, the server's masking, padding, and uniform shuffling make the client's view of SumPaths to be distribution f (Lemma D.1), while the server never sees their decryption. Complexity and formal analyses are in §C.2 and §D.

Extension to AdaBoost and Others: Private AdaBoost evaluation can be implemented by replacing each leaf score by $+\omega$ or $-\omega$ (class sign) where ω is the tree weight. Its computation time and complexity are unchanged. In general, BCC can be used for any FHE applications where the true frequency distribution of values for programmable bootstrapping in a target ciphertext is the server's private information to be hidden from the client (e.g., the number of zero SumPaths, since it equals/leaks the total number of trees in XGBoost).

3.3 Ciphertext Compression

Problem of Huge Communication Cost. In FHE-based PDTE, ciphertext size grows quickly with the number of nodes because inputs are encoded/encrypted bit-wise. For example, RCC-PDTE Akhavan Mahdavi et al. (2023) requires an 85.7 MB query ciphertext for 32-bit feature precision, where network transmission dominates inference time. The major bottleneck is *repetitive data encoding* for SIMD comparisons, which intentionally duplicates the same feature value across multiple slots for the server's computational efficiency, but instead sacrifices slot utility. For example, suppose that the server's tree model has three nodes on *age* (thresholds 21, 23, 45) and two on *sleep* (6, 8). To homomorphically compare against the encrypted client inputs *age*= 28 and *sleep*= 7 in SIMD, the client repeatedly encodes *age* and *sleep* three times each (capped by the maximum repetition of all features), wasting slots for redundancy.

To reduce such communication inefficiency, SilentWood introduces *ciphertext compression*: (1) the client removes such repetitions *before* encryption to form compact ciphertexts; (2) upon receiving them, the server *homomorphically* decompresses to restore the intended repetition pattern. Algorithm 4 generalizes the client's packing based on: (1) the M plaintext data arrays $A_0, \ldots, A_{M-1}$, one per CW-encoded bit position; (2) each feature j's start slot $start_j$; (3) the offsets $gap_0, \ldots, gap_{R-1}$ at which the R repetitions of a feature would have been encoded; and (4) the number R of arrays packed

into one ciphertext, so that the slot a repetition would have occupied now holds the bit of another array. The key requirement is that the offsets gap_r are the same for every feature, so the server can restore all repetitions homogeneously in SIMD (e.g., *age* and *sleep* are expanded simultaneously).

The server's homomorphic decompression is essentially a homomorphic reverse operation of Algorithm 4. Ciphertext (de-)compression does not leak the client's input data to the server. We provide an example of ciphertext compression and the ciphertext decompression algorithm in §E.

Benchmark	MPC		CPU-FHE					GPU-FHE				
	SoK-GGG		Zama	Baseline		SilentWood		Zama	Baseline		SilentWood	
	16-bit	32-bit	14-bit	16-bit	32-bit	16-bit	32-bit	14-bit	16-bit	32-bit	16-bit	32-bit
Spam	5.98s	5.95s	72.68s	153.68s	162.53s	3.82s	5.26s	47.42s	N/A	N/A	1.65s	2.27s
Steel	5.41s	5.56s	18.01s	37.24s	38.32s	1.81s	1.98s	10.57s	3.86s	5.62s	1.04s	1.21s
Breast	5.24s	5.69s	12.32s	20.62s	28.27s	1.57s	1.81s	9.08s	2.71s	4.85s	1.02s	1.08s
Heart	23.52s	23.26s	96.08s	174.19s	188.32s	3.76s	4.11s	75.65s	N/A	N/A	1.75s	1.95s
Defect	6.38s	6.39s	145.05s	336.04s	349.23s	6.92s	8.38s	83.66s	N/A	N/A	2.71s	3.33s
Bank	6.39s	6.45s	223.03s	436.02s	458.97s	9.39s	10.95s	151.14s	N/A	N/A	4.24s	4.16s
PenDigits	5.53s	5.47s	62.12s	71.76s	72.28s	2.17s	2.56s	49.68s	5.70s	7.53s	1.13s	1.31s
Ailerons	6.53s	6.56s	N/A	370.20s	392.15s	7.73s	10.08s	N/A	N/A	N/A	2.96s	3.97s
Credit	5.31s	5.20s	5.99s	17.51s	20.89s	1.52s	1.76s	5.55s	2.66s	4.52s	1.00s	1.08s
Satellite	4.03s	5.30s	14.49s	30.24s	36.08s	1.63s	1.91s	10.57s	3.50s	5.40s	1.05s	1.20s
Elevators	6.33s	6.24s	289.13s	427.75s	442.75s	8.39s	10.48s	149.87s	N/A	N/A	3.34s	4.07s
Telescope	4.85s	4.88s	181.88s	7.19s	9.20s	1.33s	1.57s	108.58s	1.94s	3.78s	0.89s	1.04s
MFeat	41.08s	41.57s	147.99s	301.05s	315.66s	6.81s	9.53s	169.62s	N/A	N/A	2.77s	3.86s
AVG.	9.73s	9.89s	105.73s	183.35s	193.43s	4.37s	5.41s	72.62s	3.40s	5.28s	1.97s	2.35s
SilentWood's Speedup	2.2x	1.8x	25.8x	, 41.9x	35.7x	–	–	38.6x	3.3x	4.6x	–	–

Table 2: Comparing inference time over various data bit-lengths (14 bits, 16 bits, and 32 bits) w/o GPU support. N/A denotes GPU (or RAM) failure due to large-memory-consuming FHE parameters.

4 EVALUATION

Setup: We implemented SilentWood in C++ (5518 lines) and Python (737 lines). We used the SEAL library Chen et al. (2017) for CPU-based FHE, and extended it with PhantomFHE Yang et al. (2024) for GPU-based FHE. Our experiments used an i9-14900K 24-core CPU, 168GB RAM, and an RTX 4090 GPU. To train and evaluate XGBoost, we used 13 UCI datasets Markelle Kelly et. al. (2024), whose numbers of features, classes, and sample sizes are summarized in Table 13 (§F.3). We trained the model based on the dataset ratio of training : validation : testing as 6:2:2. We compared the performance of the four systems: XGBoost based on Zama's Concrete ML (based on the TFHE-rs library) Zama (2024), SoK-GGG (an MPC protocol based on a 2-party garbled circuit) Kiss et al. (2019), Baseline (RCC-PDTE Akhavan Mahdavi et al. (2023) with MultiplyPath Cong et al. (2022)), and SilentWood. The default round-trip time (RTT) was 50ms, the default number of trees was 100, the default maximum tree depth was 7, and the default intensity s of node clustering was 0.2. We measured inference time, inference accuracy, and ciphertext size. We also conducted an ablation study for our proposed techniques over various tree counts (100, 200, 300), maximum tree depths (7, 11, 15), and RTTs (0 ms, 50ms, 100 ms, 200ms) by using the netem tool.

Summary of Results: For CPU-FHE, SilentWood's end-to-end inference time (50ms RTT) is faster than Baseline by 35.7x, faster than Zama by 25.8x, and faster than SoK-GGG by 2.2x. SilentWood is faster than SoK-GGG on average even when RTT is 0 ms, and the speedup grows to 2.94x when RTT is 200 ms. Compared to Baseline, SilentWood reduces the server-client communication size to roughly 1/5 on average. For GPU-FHE, SilentWood is faster than Baseline by 4.6x and faster than Zama by 38.6x.

End-to-end Inference Time: We evaluated end-to-end inference time with an RTT of 50ms (including the server and client's encoding/decoding, encryption/decryption, and homomorphic operations) for the cases where the feature and threshold bit sizes are 16 bits and 32 bits (quantized), each. Results are illustrated in Table 2. We tested only 14 bits for Zama because its framework supports a maximum 14 bit data length. We evaluated w/o GPU. Without a GPU, Baseline's average computation time is

Bench.	Base	Base+B	Base+B+K	Base+B+K+C (SilentWood)
Spam	162.53s	9.89s	6.59s	5.26s
Steel	38.32s	3.06s	2.26s	1.98s
Breast	28.27s	2.84s	2.21s	1.81s
Heart	188.32s	8.58s	4.06s	4.11s
Defect	349.23s	15.68s	9.43s	8.38s
Bank	458.97s	16.20s	12.20s	10.95s
Pendigits	72.28s	3.66s	2.86s	2.56s
Ailerons	392.15s	19.74s	12.24s	10.08s
Credit	20.89s	2.18s	1.79s	1.76s
Satellite	36.08s	2.52s	2.27s	1.91s
Elevators	442.75s	21.62s	11.37s	10.48s
Telescope	9.20s	1.92s	1.74s	1.57s
MFeat	315.66s	22.49s	13.29s	9.53s
AVG.	193.43s	10.03s	6.33s	5.41s
Speedup against Baseline		19.3×	30.6×	35.7×

Table 3: Ablation study.

	$s = 0$	0.01	0.05	0.1	0.2	0.4	0.6	1
Spam	0.94	0.93	0.93	0.93	0.93	0.92	0.92	0.92
Steel	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Breast	0.67	0.67	0.66	0.66	0.66	0.64	0.59	0.59
Heart	0.74	0.8	0.81	0.81	0.81	0.81	0.81	0.81
Defect	0.82	0.82	0.82	0.82	0.8	0.81	0.81	0.81
Bank	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
PenDigits	1.00	1.00	0.99	0.99	0.99	0.99	0.99	0.99
Ailerons	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.88
Credit	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Satellite	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
Elevators	0.8	0.8	0.81	0.82	0.8	0.81	0.81	0.81
Telescope	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
MFeat	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94
AVG.	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90

Table 4: Model accuracy at clustering intensities s

Benchmark	SoK-GGG				Baseline+B+K				Baseline+B+K+C			
	0ms	50ms	100ms	200ms	0ms	50ms	100ms	200ms	0ms	50ms	100ms	200ms
Spam	3.35s	5.95s	8.47s	13.44s	5.92s	6.59s	7.92s	10.11s	4.57s	5.26s	6.06s	7.70s
Steel	3.22s	5.56s	7.47s	11.91s	1.23s	2.26s	3.43s	5.57s	1.19s	1.98s	2.70s	4.31s
Breast	2.28s	5.69s	5.62s	9.16s	0.94s	2.21s	3.05s	5.30s	0.79s	1.81s	2.40s	3.94s
Heart	13.69s	23.26s	33.09s	52.88s	3.52s	4.06s	5.71s	7.92s	3.39s	4.11s	4.72s	6.34s
Defect	3.48s	6.39s	9.25s	15.06s	8.97s	9.43s	10.95s	13.27s	7.72s	8.38s	8.93s	10.48s
Bank	3.50s	6.45s	9.42s	15.96s	11.95s	12.20s	14.08s	16.25s	10.23s	10.95s	11.51s	13.00s
Pendigits	3.42s	5.47s	7.77s	12.49s	1.69s	2.86s	3.91s	6.08s	1.81s	2.56s	3.23s	4.74s
Ailerons	3.49s	6.56s	9.30s	16.15s	12.18s	12.24s	14.24s	16.36s	9.37s	10.08s	10.66s	12.17s
Credit	3.09s	5.20s	6.91s	11.84s	0.92s	1.79s	3.08s	5.30s	1.02s	1.76s	2.43s	3.93s
Satellite	3.04s	5.30s	7.54s	12.32s	1.05s	2.27s	3.29s	5.43s	1.11s	1.91s	2.61s	4.19s
Elevators	3.46s	6.24s	9.30s	15.21s	11.57s	11.37s	13.57s	15.64s	9.76s	10.48s	10.96s	12.38s
Telescope	3.16s	4.88s	6.53s	9.94s	0.58s	1.74s	2.81s	5.06s	0.83s	1.57s	2.25s	3.73s
MFeat	24.20s	41.57s	58.61s	95.23s	11.33s	13.29s	14.9s	15.76s	8.70s	9.53s	10.17s	12.22s
AVG.	5.64s	9.88s	13.79s	22.43s	5.53s	6.33s	7.76s	9.85s	4.65s	5.41s	6.05s	7.63s
SilentWood's Speedup	1.21x	1.83x	2.28x	2.94x	1.19x	1.17x	1.28x	1.29x	–	–	–	–

Table 5: End-to-end server-client inference time over various RTTs (32-bit feature size)

183.35 s (16 bits) and 193.43 s (32 bits), while Zama's average time is 105.73 s (14 bits). In contrast, SilentWood's time is 4.37 s (16 bits) and 5.41 s (32 bits). When using a GPU, Baseline's average computation time is 3.40s (16 bits) and 5.28s (32 bits), and Zama's average time is 72.62s (14 bits), whereas SilentWood's average time is 1.97s (16 bits) and 2.35s (32 bits). The FHE-based systems (Zama, Baseline, SilentWood) required more runtime for the 32-bit data size, whereas SoK-GGG (MPC) was less-sensitive to the data bit-length but more sensitive to tree depth. For some datasets, SoK-GGG's runtime is shorter for 32-bit than for 16-bit because the trained 32-bit model's tree depths were shallower.

Ablation Study: Table 3 illustrates the speedup contribution of SilentWood's each technique by evaluating the following versions: Baseline plus BCC (denoted as Baseline + B), plus computation clustering (denoted as Baseline + B + K), and plus ciphertext compression (denoted as Baseline + B + K + C). We used CPU-FHE and an RTT of 50ms for these evaluations. Regarding the speedup factors, BCC's contribution was the largest (19.3x); computation clustering was next (1.58x), and ciphertext compression was last (about 1.17x). Note that the speeedup by ciphertext compression grows as the RTT increases from 50ms to 100ms and 200ms (Table 5).

Accuracy of Node Clustering: Table 4 shows that the average model accuracy across all benchmarks (measured by the testing datasets) does not drop below 0.90 even with high clustering intensity s values, because after each round of node clustering, we validate (by using the validation datasets) the inference accuracy of the updated XGBoost model and abort the change if the model accuracy decreases. This way, we preserve model correctness even under aggressive clustering.

Inference Time over Various RTTs: Table 5 shows XGBoost's end-to-end inference time for various RTTs. In the case of 50ms RTT, the time difference between Baseline + B + K (6.33s) and Baseline

+ B + K + C (5.41s) implies the 1.19x speedup achieved by ciphertext compression. SilentWood is also faster than SoK-GGG by 1.83x, 2.28x, and 2.94x for RTT 50ms, 100ms, and 200ms respectively. SilentWood is faster than SoK-GGG by 1.21x on average even for RTT = 0ms, illustrating SilentWood's computational efficiency.

Additional Evaluations: §F provides the following experiment results: performance comparison across different tree counts and depths (§F.1), the overhead of network, CPU, and memory across different modes (§F.2), varying the number of node clusters, the validation set size, and the applicability of node clustering to MPC (§F.3), and inference time across different network bandwidths (§F.4).

5 Related Work

FHE vs 2PC for PDTE: RCC-PDTE Akhavan Mahdavi et al. (2023) uses CW-encoding-based comparison to improve this limitation, yet its SumPath aggregation is incompatible with standard forest inference and becomes costly with many trees. Brickell et al. (2007) and Kiss et al. (2019) design constant-round 2PC protocols for PDTE from oblivious transfer and garbled circuits (GC), but their communication grows exponentially with tree depth for evaluating all paths; oblivious RAM (Tueno et al. (2019)) or pseudo-random functions (Bai et al. (2022)) help only modestly. Gazelle-style FHE+GC hybrids Juvekar et al. (2018) inherit this cost for tree traversal (as SoK-GGG does), whereas BCC's only interactive step exchanges packed FHE ciphertexts.

Scaling PDTE to Random Forests: HBDT (Shin et al. (2024)) uses FHE with one-hot encoded features, but one-hot inflates ciphertext count (e.g., a 32-bit feature needs 655 ciphertexts). BPDTE-CW (Cong et al. (2024)) uses one non-zero digit per node to batch comparisons, yet communication still grows (e.g., +17.7% when precision increases from 11 to 16 bits). Moreover, since summing path scores (SumPath) dominates cost, faster comparisons alone yield limited speedup. Both HBDT and BPDTE-CW aggregate results via MultiplyPath, multiplying along decision paths and incurring heavy ciphertext multiplications. In contrast, SilentWood uses summation with BCC to avoid multiplications. Kangaroo (Xu et al. (2026)) combines packed HE and secret sharing for amortized private inference over large tree ensembles in WANs. Compared with SilentWood, it assumes a weaker threat model (semi-honest server and client), reveals an obfuscated tree topology that constrains the original tree structure, and requires 4 RTTs. SilentWood supports malicious-server privacy, hides tree topology from the client, and uses 2 RTTs.

Other Efficiency Protocols: SortingHat (Cong et al. (2022)) and Onion-ORAM (Chen et al. (2019)) are based on TFHE, much slower than SilentWood because they lack SIMD. SortingHat's transciphering accounts for 35% of inference time and grows with tree depth; Onion-ORAM's ciphertext expansion adds 33–66% overhead (see Table 2 in their paper). SilentWood 's decompression overhead is only 4.5% (Table 11), independent of depth, making it a more efficient compression method for BFV/BGV.

Tree Compression: Various pruning techniques reduce inference cost by trimming nodes or trees, e.g., Liu & Mazumder (2023), Devos et al. (2025), (fip (2025). In contrast, our node clustering preserves the full model structure and merges similar thresholds per feature to reduce distinct comparisons while preserving logic structure. This approach is beneficial when preserving the tree structure is important for interpretability or certification workflows. SilentWood 's node clustering complements pruning methods by unifying similar threshold values even after pruning a model.

6 Conclusion

SilentWood's private inference over gradient-boosting forests significantly enhances performance compared to existing methods. SilentWood's novel techniques include node clustering to eliminate duplicate homomorphic computations, blind code conversion to preserve arithmetic compatibility for efficient tree score aggregation, and ciphertext compression to reduce the encrypted data size. SilentWood achieves an average private XGBoost inference time of 1.8x ~ 41.9x faster than state-of-the-art FHE or MPC-based protocols (i.e., Zama's concrete ML, RCC-PDTE, and SoK-GGG).

AI USE STATEMENT

In this work, we used generative AI tools to formulate mathematical claims, assist in writing proofs, implement methods, and review the manuscript for numerical, notational, and textual inconsistencies. We did not use generative AI tools to develop theoretical models or conceptual frameworks; provide critical ingredients for proving mathematical claims; generate synthetic datasets; propose or refine hypotheses; design or provide feedback on research methodology or experiments; assist with translation; clean or reformat datasets; support qualitative or thematic data analysis; or interpret results. We reviewed all AI-assisted work and take responsibility for the final content.

ETHICS STATEMENT

Our work studies privacy-preserving inference for gradient boosting decision forests using only public UCI datasets Markelle Kelly et. al. (2024), which contain no personally identifying information. No attacks were conducted against real systems, and no harm resulted from this study. As with any cryptographic technique, misuse is possible: the confidentiality that protects legitimate models could also conceal discriminatory or harmful ones, which we discourage.

REPRODUCIBILITY STATEMENT

We will release the artifact to reproduce the experimental results illustrated in §4.

REFERENCES

Free lunch in the forest: Functionally-identical pruning of boosted tree ensembles. 39:16488–16495, Apr. 2025. doi: 10.1609/aaai.v39i16.33811. URL https://ojs.aaai.org/index.php/AAAI/article/view/33811.

Adi Akavia, Craig Gentry, Shai Halevi, and Margarita Vald. Achievable CCA2 relaxation for homomorphic encryption. *J. Cryptol.*, 38(1):5, 2025.

Rasoul Akhavan Mahdavi, Haoyan Ni, Dimitry Linkov, and Florian 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*, CCS '23, pp. 2945–2958, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3623095. URL https://doi.org/10.1145/3576915.3623095.

Jianli Bai, Xiangfu Song, Shujie Cui, Ee-Chien Chang, and Giovanni Russello. Scalable private decision tree evaluation with sublinear communication. In *Proceedings of the ACM Asia Conference on Computer and Communications*, pp. 843–857, 2022.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

Leo Breiman. Random forests. *Machine Learning*, pp. 5, 2001.

Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security*, pp. 498–507, 2007.

Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - seal v2.1. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (eds.), *Financial Cryptography and Data Security*, pp. 3–18, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70278-0.

Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring oram: Efficient constant bandwidth oblivious ram from (leveled) tfhe. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pp. 345–360, New York, NY, USA, 2019. Association for

Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354226. URL https://doi.org/10.1145/3319535.3354226.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pp. 785–794, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira. Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In *Proceedings of the 2022 ACM Conference on Computer and Communications Security*, pp. 563–577, 2022.

Kelong Cong, Jiayi Kang, Georgio Nicolas, and Jeongeun Park. Faster private decision tree evaluation for batched input from homomorphic encryption. In Clemente Galdi and Duong Hieu Phan (eds.), *Security and Cryptography for Networks*, pp. 3–23, Cham, 2024. Springer Nature Switzerland.

Laurens Devos, Timo Martens, Deniz Can Oruc, Wannes Meert, Hendrik Blockeel, and Jesse Davis. Compressing tree ensembles through level-wise optimization and pruning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=9Klg7ce8D7.

Léo Ducas and Damien Stehlé. Sanitization of FHE ciphertexts. In *EUROCRYPT (1)*, volume 9665 of *Lecture Notes in Computer Science*, pp. 294–310. Springer, 2016.

Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012a.

Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, pp. 144, 2012b.

Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, EuroCOLT '95, pp. 23–37, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3540591192.

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001. doi: 10.1214/aos/1013203451. URL https://doi.org/10.1214/aos/1013203451.

Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford, CA, USA, 2009a. AAI3382729.

Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing*, pp. 169–178, 2009b.

Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1651–1669, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar.

Ágnes Kiss, Masoud Naderpour, Jian Liu, N. Asokan, and Thomas Schneider. Sok: Modular and efficient private decision tree evaluation. *Proceedings on Privacy Enhancing Technologies*, 2019(2): 187–208, 2019.

Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pp. 277–346. Springer International Publishing, 2017.

Brian Liu and Rahul Mazumder. Forestprune: Compact depth-pruned tree ensembles. In Francisco Ruiz, Jennifer Dy, and Jan-Willem van de Meent (eds.), *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pp. 9417–9428. PMLR, 25–27 Apr 2023. URL https://proceedings.mlr.press/v206/liu23h.html.

Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 1723–1740, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/mahdavi.

Markelle Kelly et. al. The UCI Machine Learning Repository, 2024. https://archive.ics.uci.edu.

Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, USA, 1st edition, 2009. ISBN 052111991X.

Hojune Shin, Jina Choi, Dain Lee, Kyoungok Kim, and Younho 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*, pp. 217–237, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-70895-4. doi: 10.1007/978-3-031-70896-1_11. URL https://doi.org/10.1007/978-3-031-70896-1_11.

Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. Private evaluation of decision trees using sublinear cost. *Proceedings on Privacy Enhancing Technologies*, 2019(1):266–286, 2019.

Wei Xu, Hui Zhu, Yandong Zheng, Song Bian, Ning Sun, Hao Yuan, Dengguo Feng, and Hui Li. Kangaroo: A private and amortized inference framework over wan for large-scale decision tree evaluation. In *33rd Annual Network and Distributed System Security Symposium, NDSS 2026, San Diego, California, USA, February 23 - 27, 2026*. The Internet Society, 2026. URL https://www.ndss-symposium.org/ndss-paper/kangaroo-a-private-and-amortized-inference-framework-over-wan-for-large-scale-deci

Hao Yang, Shiyu Shen, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. Phantom: A cuda-accelerated word-wise homomorphic encryption library. *IEEE Transactions on Dependable and Secure Computing*, pp. 1–12, 2024. doi: 10.1109/TDSC.2024.3363900.

Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *Proceedings of the 23rd Symposium on Foundations of Computer Science*, pp. 160–164, 1982.

Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pp. 162–167, 1986.

Zama. Concrete ML, 2024. https://docs.zama.ai/concrete-ml.

A OPERATIONS OF THE BFV SCHEME

The inputs and outputs of the BFV scheme's operations are defined as follows:

1. Encode$([v_0, v_1, \cdots, v_{n-1}])$ ▷ each $v_i \in \mathbb{Z}_t$ for plaintext modulus t
   $\rightarrow M(X)$ ▷ $M(X) \in \mathbb{Z}_t[X]/(X^n + 1)$
2. Decode$(M(X)) \rightarrow [v_0, v_1, \cdots, v_{n-1}]$
3. Encrypt$(M(X), pk) \rightarrow ct$ ▷ a ciphertext that encrypts $\Delta \cdot M(X)$ under $sk = S(X) \in \{-1, 0, 1\}^n \subset \mathbb{Z}[X]/(X^n + 1)$
4. Decrypt$(ct, sk) \rightarrow M(X)$
5. PAdd$(ct_1, M_2(X)) \rightarrow ct$ ▷ encrypts $\Delta \cdot (M_1(X) + M_2(X))$, which decodes to Decode$(M_1(X))$ + Decode$(M_2(X))$
6. CAdd$(ct_1, ct_2) \rightarrow ct$ ▷ encrypts $\Delta \cdot (M_1(X) + M_2(X))$, which decodes to Decode$(M_1(X))$ + Decode$(M_2(X))$
7. PMult$(ct_1, M_2(X)) \rightarrow ct$ ▷ encrypts $\Delta \cdot (M_1(X) \cdot M_2(X))$, which decodes to Decode$(M_1(X))$ $\odot$ Decode$(M_2(X))$ (where $\odot$ is element-wise vector multiplication)
8. CMult$(ct_1, ct_2) \rightarrow ct$ ▷ encrypts $\Delta \cdot (M_1(X) \cdot M_2(X))$, which decodes to Decode$(M_1(X))$ $\odot$ Decode$(M_2(X))$
   (assuming CMult includes modulus switching to rescale $\Delta^2 \rightarrow \Delta$)
9. Rotate$(ct, r) \rightarrow ct'$ ▷ if Decode$(M(X)) = [v_0, \ldots, v_{n-1}]$, then Decode$(M'(X)) = [v_r, \ldots, v_{n-1}, v_0, \ldots, v_{r-1}]$

B Detailed Composition of Private Decision Tree Evaluation (PDTE)

B.1 Value Comparison: Homomorphic Logic Gates

As explained in §2.2, the only arithmetic operations directly supported by FHE are addition and multiplication $(+, \cdot)$. Therefore, to homomorphically compare two numbers, one needs to design an arithmetic logic circuit comprising only $(+, \cdot)$ operators such that the gate returns 1 if $\alpha > \beta$, and returns false if $\alpha \leq \beta$.

Given the input feature X node and threshold Y are n-bits long, one way to build a value comparison circuit is to encrypt each binary bit of X and Y as $(x_1, x_2, \cdots, x_n)$ and $(y_1, y_2, \cdots, y_n)$ and design the value comparator logic as follows:

- $\textbf{EQBit}(a, b) = 1 - a - b \pmod 2$ ▷ bit a is equal to bit b
- $\textbf{GTBit}(a, b) = a \cdot (1 - b) \pmod 2$ ▷ bit $a > b$
- $\textbf{EQ}_{i \to j}(X, Y) = \prod_{j=i}^{k} \mathsf{EQBit}(x_j, y_j)$ ▷ Are X, Y's bits $i \sim k$ equal?
- $\textbf{GT}(X, Y) = \sum_{i=0}^{n-1} \left(\mathsf{EQ}_{i+1 \to n-1}(X, Y) \cdot \mathsf{GTBit}(x_i, y_i)\right)$ ▷ $X > Y$?

Given the above setup, $\mathsf{GT}(X, Y)$ returns 1 if $X > Y$; 0 otherwise. Since the above formulas comprise only addition and multiplication operations, they can be executed by using FHE. However, using a bit-wise comparator requires the later tree path evaluation operation to be performed also as a verbose binary circuit, which is not an optimal design for FHE. This is because binary circuits usually involve a deep multiplication chain and FHE requires an increasingly more computation time for a greater multiplication depth.

B.2 Edge Formulas for SumPath and MultiplyPath

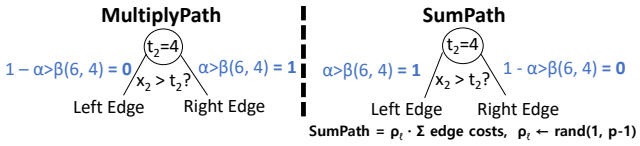

Figure 2: Edge formulas for SumPath and MultiplyPath

Figure 2 shows the per-edge formulas. Let $b_i = \alpha{>}\beta(x_i, t_i) \in \{0, 1\}$ be the encrypted comparison result at node i. MultiplyPath assigns the indicator b_i to the edge taken when the comparison holds and $1 - b_i$ to the other edge, and multiplies the indicators along each root-to-leaf path. SumPath assigns the complementary *cost* $1 - b_i$ to the edge taken when the comparison holds and b_i to the other edge, and sums the costs along each path. The cost of leaf j is therefore

$$c_j = \sum_{i \in path(j)} cost_i \in \{0, 1, \ldots, d_j\}, \tag{2}$$

which is the number of unsatisfied edges on the path, where $d_j \leq d$ is the depth of leaf j. $c_j = 0$ iff j is the true leaf of its tree, and otherwise $c_j \geq 1$ because at least the edge where the path departs from the true path is unsatisfied. Computing c_j requires only CAdd (and a plaintext negation for $1 - b_i$). To hide c_j from the client, the server multiplies each leaf cost by a fresh mask $\rho_j \xleftarrow{\$} \mathbb{Z}_p^*$, so the SumPath value of leaf j is $\rho_j c_j$. In SilentWood, this mask is applied once to the merged ciphertext (Algorithm 2, line 10), so path evaluation costs one plaintext multiplication per ciphertext rather than one per node.

Two properties follow because p is prime (a requirement of BFV batching anyway) and $d < p$. First, no false path can vanish: $c_j \in \{1, \ldots, d\} \subseteq \mathbb{Z}_p^*$ and $\rho_j \in \mathbb{Z}_p^*$, and a field has no zero divisors, so $\rho_j c_j \neq 0$. Second, $\rho_j c_j$ is *uniform over* $\mathbb{Z}_p^*$ and independent across leaves, since $\rho \mapsto \rho c_j$ is a bijection on $\mathbb{Z}_p^*$ and the ρ_j values are independent.

B.3 VALUE COMPARISON: CW-ENCODING

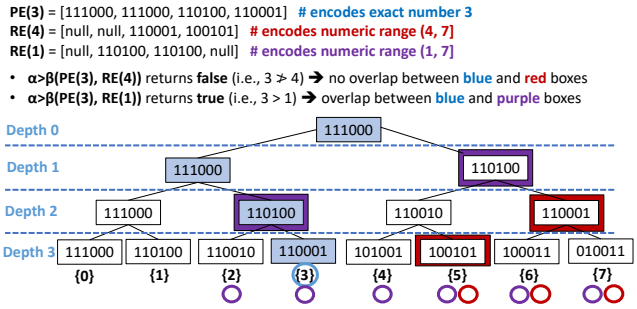

Figure 3: RCC-PDTE Akhavan Mahdavi et al. (2023)'s examples of value comparison ($3 > 4$? $3 > 1$?), where each tree node's label is CW-encoded.

We run this function at each tree node, where **α** = feature, **β** = threshold

```
bool α>β(PE(α), RE(β)): # Given PE(α) and RE(β), determine if α > β
  int accumulator = 0;
    for (int i = 0; i < PE(α).length; i++)
      accumulator += is_equal(PE(α)[i], RE(β)[i])
    return accumulator;              # returns 1 if α > β; 0 otherwise
```

```
bool is_equal(x, y)                  # returns 1 if x == y; 0 otherwise
  int h = 0;
  for (int i = 0; i < L; i++)        # L: bit length of the encoded values
    h += x[i] * y[i]
  m = 1;
  for (int i = 0; i < H; i++)        # H: Hamming weight (number of 1s)
    m *= h - i
  int H_FACTORIAL = H*(H-1)*(H-2)*...3*2*1  # a pre-computed constant
  return (H_FACTORIAL)^-1 * m  # where m is either H_FACTORIAL or 0
```

Figure 4: An arithmetic circuit of comparing two input numbers (α, β) encoded by CW-encoding (PE(α), RE(β)).

The state-of-the-art value comparison algorithm based on FHE is the one proposed by RCC-PDTE Akhavan Mahdavi et al. (2023). Figure 3 illustrates a high-level idea of how this works: two numbers to be compared are encoded as a vector of node labels that form a complete binary tree (CBT, which is different from a decision tree). CBT's each leaf node corresponds to the target number to encode. The CBT has 8 leaves and can encode any numbers between 0 and 7. Based on this CBT, RCC-PDTE proposes two number encoding schemes: PE (point encoding) and RE (range cover encoding). PE encodes an exact number corresponding to a particular leaf node in a CBT as a vector of node labels. The elements of this vector tell the path from the root to a target leaf node. In Figure 3, we PE-encode 3 as PE vector $[111000, 111000, 110100, 110001]$ (visualized as blue nodes and a blue circle). The CBT's each node label is encoded as constant-weight encoding Mahdavi & Kerschbaum (2022) (CW-encoding), which is essentially a Hamming-weight encoding whose base-2 representation always contains the same frequency of 1s for any target value to encode. We will later explain why we use CW-encoding for labeling.

Unlike PE encoding, RE encodes a range of values as a vector such that each of its elements corresponds to a specific node in each tree depth (or can be NULL if no node is chosen at that depth), and the encoded range of values is the set of all descendant leaves of the nodes in the vector. In Figure 3, we RE-encode range $(4, 7]$ as RE vector $RE(4) = [null, null, 110001, 100101]$ (visualized as red nodes), whose each element's descendant leaves collectively cover value range $\{5, 6, 7\}$ (red circle). As another example, we RE-encode range $(1, 7]$ as $RE(1) = [null, 110100, 110100, null]$ (purple nodes), whose descendant leaves collectively cover value range $\{2, 3, 4, 5, 6, 7\}$ (purple circles). Notably, the upper bound of the range encoded by RE is always the right-most leaf value in CBT (e.g., 7 in case of Figure 3).

Based on this CBT setup, to evaluate the condition $\alpha > \beta$, we determine whether two vectors $PE(\alpha)$ and $RE(\beta)$ share any common node (i.e., same node label at the same depth of CBT). If yes, then $\alpha > \beta$ holds; otherwise $\alpha \leq \beta$. For example, $PE(3)$ and $RE(4)$ have no common node at any depth, which visually implies that the blue path of $PE(3)$ has no overlap with the red boundary of $RE(4)$

(i.e., $3 \leq 4$). On the other hand, $PE(3)$ and $RE(1)$ have a common node at depth 2 (110100), which visually implies that the blue path of $PE(3)$ has an overlap with the purple boundary of $RE(1)$ (i.e., $3 > 1$). This value comparison logic is implemented in the upper dashed box of Figure 4. Within this function, the for-loop statement can be executed with FHE as a series of $(+, \cdot)$ operations, and we further detail the is_equal function's internal logic in the lower dashed box, which again comprises only $(+, \cdot)$ operations. Therefore, the entire value comparison operation can be performed by FHE with the target numbers being encrypted ciphertexts. In particular, is_equal assumes that the CBT's each node label is encoded as CW-encoding. Note that all for-loops in these functions have a deterministic number of looping, because the conditional variables for looping (.length, L, and H) are statically known values. The benefit of using CW encoding in this comparison algorithm is that during the FHE computation, we can effectively fix the number of multiplicative depth (fixed to $\lceil \log H \rceil + 2$) required for the is_equal operation regardless of the bit-length of input data (x, y).

B.4 Tree Path Evaluation: MultiplyPath and SumPath

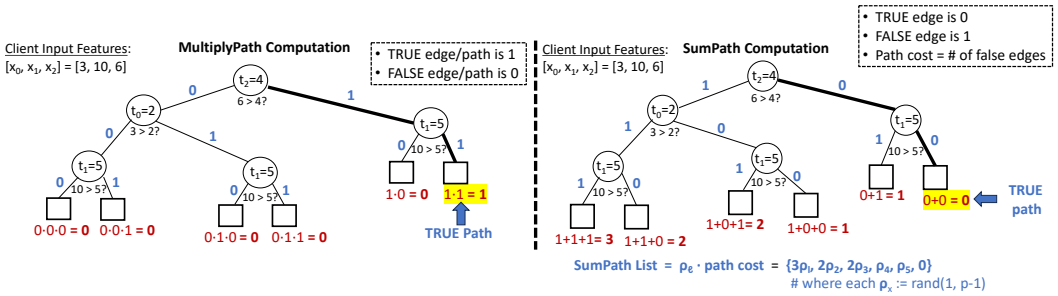

Figure 5: A tree path evaluation example of MultiplyPath and SumPath

In Figure 5, the client's input is $[x_0, x_1, x_2] = [3, 10, 6]$, so every comparison holds ($6 > 4, 3 > 2$, $10 > 5$) and, at every node, the left edge (taken when the comparison fails) has cost 1 while the right edge has cost 0. For MultiplyPath the indicators are complementary, and only the rightmost leaf has product 1. For SumPath, the six leaves (left to right) obtain the costs $3, 2, 2, 1, 1, 0$; only the rightmost leaf, the true path, is 0. After masking with per-leaf ρ_ℓ, the SumPath values are $3\rho_1, 2\rho_2, 2\rho_3, \rho_4, \rho_5, 0$: all non-zero except the true path, and each non-zero value is uniform over $\mathbb{Z}_p^*$.

C An Example and Overhead Analysis of the Blind Code Conversion Protocol

C.1 An example of the Blind Code Conversion (BCC) Protocol

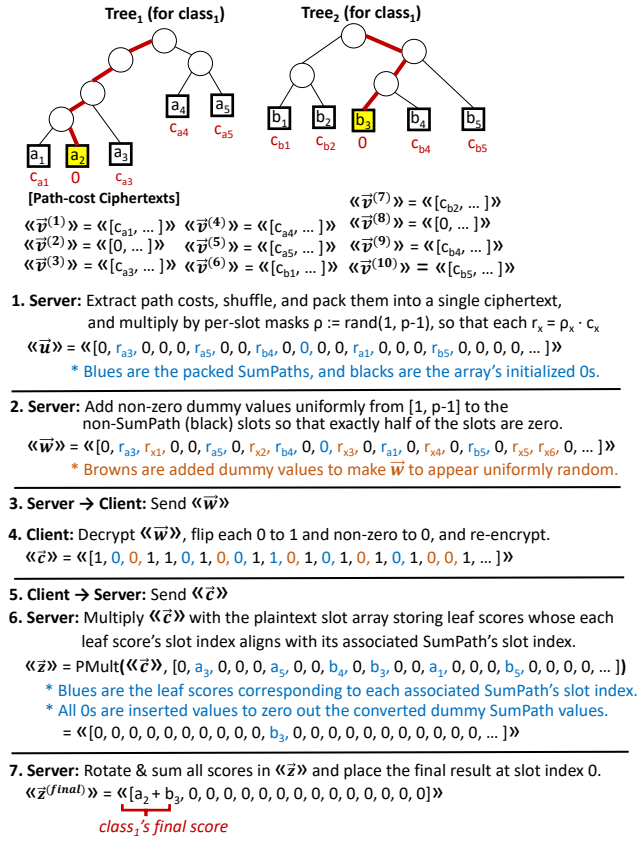

Figure 6: An example of BCC for XGBoost scoring.

Figure 6 shows an example of XGBoost score computation based on the BCC protocol. Initially, the server has a list of path-cost ciphertexts, each of which contains a distinct (unmasked) path cost c_i (Equation 2) in slot index 0 and zero values in other slots. In step 1, the server extracts only the encrypted path costs, randomly rotates each of them, merges them into a new array $\vec{u}$, and multiplies the merged ciphertext by a per-slot mask $\vec{\rho}$ whose entries are uniform over $\mathbb{Z}_p^*$ (one PMult). The resulting slot values $r_x = \rho_x c_x$ are the SumPaths shown in the figure. The output of step 1 is $\langle\!\langle \vec{u} \rangle\!\rangle$ (where $\langle\!\langle\rangle\!\rangle$ denotes encryption). In step 2, the server homomorphically adds dummy values to $\vec{u}$ (non-zero dummies drawn uniformly from $\mathbb{Z}_p^*$) to make the ratio between the zeros and non-zeros uniform (1:1 in Figure 6's example) and randomly ordered. This ratio is a fixed hyperparameter of the protocol. In step 3, the server sends the ciphertext to the client. In step 4, the client decrypts it, flips 0s to 1s and non-zeros to 0s, re-encrypts them, and sends them back to the server. During this process, the client learns nothing about the real SumPath values: since p is prime and $1 \le c_j \le d < p$, every false-path value $\rho_j c_j$ is non-zero and uniform over $\mathbb{Z}_p^*$, so the decrypted slot array is distributed exactly as f_{zero} zeros at random positions and $N - f_{zero}$ i.i.d. $\mathbb{Z}_p^*$-uniform values, whatever the model and input (Lemma D.1). In particular, exactly T of the zeros are true paths, and no false path is ever converted to 1. The server also learns nothing about the SumPath values because they always remain encrypted when handled by the server. In step 5, the server receives the ciphertext containing client-converted values. In step 6, the server homomorphically multiplies the ciphertext by the server's plaintext array that encodes the leaf values (scores), where each array slot's leaf value aligns with its associated SumPath's slot index. Thus, when we multiply this array with the client-returned ciphertext, the leaf values for false SumPaths and dummy values are masked to 0, while those for true SumPaths remain. In step 7, the

server sums all per-tree scores according to the class by rotating and adding the slot values to compute the final score for each class. Finally, the server compactly packs all final class scores into $\langle\!\langle \vec{z}^{(final)} \rangle\!\rangle$ and sends them to the client. In the case of binary classification, a single final score is returned to the client, and whether this score is positive or negative indicates the binary classification result. In the case of multi-class classification, the highest class score is regarded as the inferred class.

In Figure 6, steps $1 \sim 5$ comprise the generic BCC protocol (which we formally describe in Table 1), whereas steps $6 \sim 7$ involve application-specific score computation for XGBoost.

C.2 Analysis of the Score Aggregation Overhead

	BCC	**MultiplyPath**
CMult	0	$O(\ell)$
PMult	$O(\log N) + 1 \approx O(1)$	$O(\ell)$
Rotate	$O(\ell + \log N) \approx O(\ell)$	0
CAdd	$O(\ell + \log N) \approx O(\ell)$	$O(\ell)$
RTTs	1	0

Table 6: Comparing the runtime overhead of leaf score aggregation based on BCC *vs.* MultiplyPath.

	BCC	**MultiplyPath**
CMult	0	$O(d)$
PMult	$O(\log N) + 1 \approx O(1)$	$O(1)$
Rotate	$O(\log N) \approx O(1)$	0
CAdd	$O(\log \ell + \log N)$	$O(\log \ell)$

Table 7: Circuit depth (i.e., critical path) of each type of operation.

N is the number of plaintext slots in a ciphertext, ℓ is the total number of tree paths, and d is the tree depth (i.e., path length). In this time complexity analysis, we assume a complete binary tree. Therefore, there are a total of $2\ell - 1$ nodes. In practical FHE, $\log N$ is fixed to be between 13 and 15; thus, $O(\log N) \approx O(1)$.

Table 6 shows the time complexity of score aggregation based on BCC and MultiplyPath. As illustrated in Algorithm 2, score aggregation based on BCC requires ℓ rotations and $\ell - 1$ additions to randomly permute and aggregate ℓ real path costs (lines 6~9), 1 plaintext multiplication to mask them (line 10), and 1 addition to merge them with randomly permuted dummy SumPaths (line 14). The overhead of $O(\log N)$ rotations and $O(\log N)$ plaintext-ciphertext multiplications occur during score aggregation (line 6, 7, and 8 in Algorithm 3), but this overhead is fixed and small in practice because $\log N$ is fixed to be between 13 and 15. Additionally, BCC involves one round-trip communication between the server and the client. During score aggregation in Algorithm 3, BCC requires $O(\log N) \approx O(1)$ ciphertext-to-plaintext multiplications to aggregate the final score (line 8). The masking PMult does not add to the cost of path evaluation either: the original SumPath formulation multiplies every node's comparison result by a random value (one PMult per node) before summation, whereas SilentWood sums the $0/1$ edge costs and applies a single PMult per merged ciphertext afterwards. In both cases the multiplicative depth after the comparison circuit is one plaintext multiplication with plaintext magnitude bounded by p, so the noise budget is unchanged.

In contrast, for a full binary tree with ℓ leaves, there are $\ell - 1$ internal nodes and $2\ell - 2$ edges. With shared path prefixes, MultiplyPath computes all leaf indicators using $O(\ell)$ CMults. Weighting the ℓ leaf indicators requires ℓ PMults, followed by $\ell - 1$ CAdds to aggregate the scores. Since all these operations are performed locally by the server, no round-trip communication with the client is needed. However, the dominant cost arises from the $O(\ell)$ CMult operations, which is the most computationally expensive and consumes the largest noise budget among all (i.e., it is generally 10 times slower than PMult or Rotate).

Table 7 shows the circuit depth (i.e., the critical path) of score aggregation based on BCC and MultiplyPath.

The ℓ rotations are mutually independent. Combining their outputs requires $\ell - 1$ ciphertext additions, with depth $\lceil \log_2 \ell \rceil$ when implemented as a balanced reduction. The ℓ rotations are mutually

independent. Combining their outputs requires $\ell - 1$ ciphertext additions, with depth $\lceil \log_2 \ell \rceil$ when implemented as a balanced reduction. The subsequent score aggregation contributes another $\log_2 N$ sequential levels.

With multi-core processors or GPUs, this for-loop can also run efficiently in parallel. On the other hand, the critical path of MultiplyPath is d CMult operations during path evaluation. This is a significant computational bottleneck because, as the tree depth grows linearly, the computational overhead grows polynomially.

D SECURITY ANALYSIS: BLIND CODE CONVERSION

Our protocol achieves privacy against both semi-honest clients and malicious servers.[3] In particular, the client only learns the final class scores and some hyperparameters of the server's model, while the server learns essentially nothing about the client's features. Below we formally prove security with respect to each type of corruption when the protocol is instantiated with a sanitized FHE scheme.

The idealized functionality $\mathcal{F}_{\text{BCC}}$ of the protocol is defined as follows:

- The client chooses its input for the classification.
- The server chooses its decision forest model, including some hyperparameters such as the frequency distribution f and the upper bound on the number of SumPaths ℓ_{max}.
- The client obtains the final class scores computed from the decision forest and the hyperparameters f and ℓ_{max}.

Correctness: Let p be the (prime) plaintext modulus and $d < p$ the maximum tree depth. For every leaf j, the path cost satisfies $c_j \in \{0, \ldots, d\}$ with $c_j = 0$ iff j is the true leaf of its tree (Equation 2). Since $\mathbb{Z}_p$ is a field and $\rho_j \in \mathbb{Z}_p^*$, the masked value $\rho_j c_j$ is 0 iff $c_j = 0$. Hence the conversion table $\mathcal{U}_p$ maps exactly the T true-path slots of $\vec{w}$ to 1 (all dummy slots being either 0 or $\mathbb{Z}_p^*$-valued), Algorithm 3 sums exactly the true-path leaf scores, and the number of zeros in $\vec{w}$ is exactly f_{zero}. The protocol therefore has no probabilistic decoding error, provided that homomorphic evaluation decrypts correctly. Assigning independent random values to individual edges and summing them instead would let a false path with $k \geq 2$ unsatisfied edges vanish with probability $\approx 1/p$, i.e., $\approx \ell/p$ per query.

Lemma D.1 (Distribution of the client's intermediate view)**.** *Let p be prime and $d < p$; let $\vec{\rho} \xleftarrow{\$} (\mathbb{Z}_p^*)^N$ and the non-zero dummies be sampled independently and uniformly from $\mathbb{Z}_p^*$; and let $\vec{\sigma}$ be a uniformly random permutation with f_{zero} zero slots. We assume $T \leq f_{zero}$ and $\ell - T \leq N - f_{zero}$. Then, for every model y and every input x, the plaintext $m_1 = \vec{w}$ underlying c_1 is distributed as follows: the positions of the f_{zero} zero slots form a uniformly random subset of $[N]$, and the remaining $N - f_{zero}$ slots are i.i.d. uniform over $\mathbb{Z}_p^*$.*

Proof. Each true path contributes a 0, and the $f_{zero} - T$ dummy zeros contribute the rest, so there are exactly f_{zero} zeros. For a false path j, c_j is a fixed element of $\{1, \ldots, d\} \subseteq \mathbb{Z}_p^*$, and $\rho \mapsto \rho c_j$ is a bijection on $\mathbb{Z}_p^*$, so $\rho_j c_j$ is uniform over $\mathbb{Z}_p^*$; independence across paths follows from the independence of the ρ_j. The non-zero dummies are i.i.d. uniform over $\mathbb{Z}_p^*$ by construction. Finally, $\vec{\sigma}$ places all values at uniformly random positions. None of these distributions depends on y or x. □

Security Against Corrupted Client: We demonstrate how the client's view may be simulated using only its own input, the hyperparameters and the final class scores. The client's view consists of its own input, its random coins, and the transcript of the protocol Lindell (2017). Formally, we have

$$\mathsf{view}_{\mathsf{Clnt}}(x, y, \lambda) = (x, r, c_1, c_2),$$

where x holds the client's private features, y is the server's private model, λ is the security parameter, r is the collection of the client's random coins, c_1 is the ciphertext holding the padded and shuffled SumPath values, and c_2 is the ciphertext holding the final class scores.

We require a sanitization algorithm, which transforms a ciphertext in a manner that preserves the message, while removing all information about the evaluated circuit. In other words, when sanitizing two ciphertexts that decrypt to the same message, the outputs are statistically close (hence indistinguishable). We refer to Ducas and Stehlé Ducas & Stehlé (2016) for a formal treatment of sanitization. Assuming that c_1 and c_2 are sanitized, we can build a simulator for the client's view as follows:

- The simulator copies the value of x from its input.
- The simulator generates r uniformly at random.

[3] Privacy against malicious clients can be achieved by attaching a proof of well-formedness to all keys and ciphertexts.

- By Lemma D.1, the distribution of the plaintext m_1 underlying c_1 depends only on the public f and ℓ_{max}, not on x, y, or the other entries of the client's view (it is masked, padded, and shuffled using the server's random coins). Therefore, the simulator samples m_1 exactly from this distribution (f_{zero} zeros at uniformly random positions, i.i.d. $\mathbb{Z}_p^*$-uniform values elsewhere), so the simulated m_1 is identically distributed to the real one. It then generates the encryption key from the client's random coins, and it encrypts and sanitizes m_1. Due to the sanitization property, the resulting ciphertext is indistinguishable from the actual c_1.
- The simulator encrypts and sanitizes the final class scores. Due to the sanitization property, the resulting ciphertext is indistinguishable from the actual c_2.

Security Against Corrupted Server: SilentWood can be modeled as a client-aided outsourcing protocol. Akavia et al. Akavia et al. (2025) show that such a protocol achieves privacy if at least one of the following conditions is satisfied:

- It is cleartext computable (all plaintexts/cleartexts are functions of the client's input) and it is instantiated with a CPA-secure FHE scheme.
- It is instantiated with a funcCPA-secure FHE scheme.

The first condition is not satisfied in SilentWood, because the plaintexts depend on the server's model and its padding & random shuffling. However, Akavia et al. Akavia et al. (2025) show that sanitized CPA-secure schemes are funcCPA-secure. Therefore, security can be proven in one of these situations: (i) BFV is funcCPA-secure; or (ii) BFV is CPA-secure and all ciphertexts are sanitized before sending them to the server.

E HOMOMORPHIC CIPHERTEXT DECOMPRESSION WITH EXAMPLE

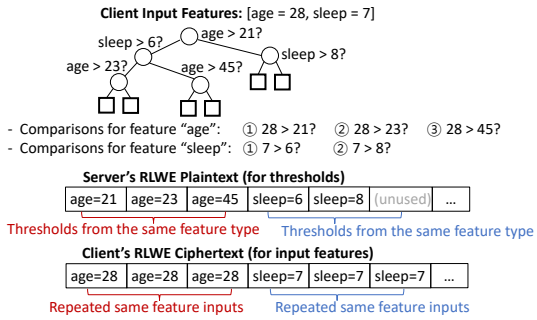

Figure 7: An example of repetitive data encoding

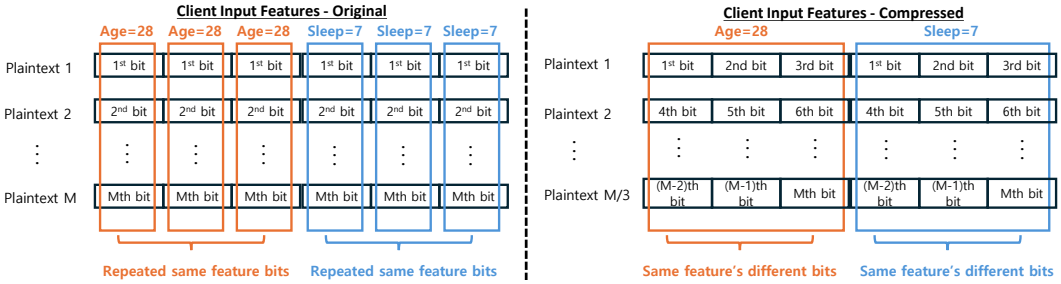

Figure 8: An example of ciphertext compression

Figure 7 is an example showing how PDTE generally uses repetitive data encoding: the server's tree has 3 nodes conditioned on the *age* feature (each threshold is 21, 23, and 45) and 2 nodes conditioned on the *sleep* feature (each threshold is 6 and 8). The server homomorphically compares these threshold values against the client's encrypted *age=28* and *sleep=7* input feature values in a SIMD manner. Since the server has 3 distinct threshold values for *age* and 2 distinct threshold values for *sleep*, the client encodes its *age=28* and *sleep=7* values repetitively 3 times (capping to the server's maximum number of comparisons for any same feature name). While such repetitive data encoding enables efficient SIMD comparisons, this repetition wastes ciphertext slot utility.

Figure 8 shows the resulting example of SilentWood's ciphertext compression. In FHE-based private protocols, when a client CW-encodes feature data, s/he has to create as many ciphertexts as the number of feature data bits (M in Figure 8's example), where each ciphertext repetitively encodes each feature's same bit 3 times (to help the server-side SIMD operation). On the other hand, using SilentWood's ciphertext compression, the client removes these repetitions of data before encrypting them, which results in 2 empty slots for each feature bit storage in the ciphertext. Therefore, the client can store the bits of other digits of data in this available extra space. Applying this compact data encoding, the total required number of ciphertexts reduces from M to $\frac{M}{3}$. Once the server receives these compressed ciphertexts, it homomorphically performs a reverse operation of what the client did, effectively decompressing and restoring the originally intended data encoding (i.e., 3 repetitions of feature bits).

Algorithm 5 Ciphertext Decompression

Input: $\{C_0, \ldots, C_{K-1}\}, \{start_j\}, \{gap_0, \ldots, gap_{R-1}\}, R$
▷ $K = \lfloor M/R \rfloor$ compressed ciphertexts; same $start_j$, gap_r, R as in Algorithm 4
for $k \in \{0, \ldots, K-1\}$ **do**
  for $r \in \{0, \ldots, R-1\}$ ▷ recover A_{kR+r} **do**
    $\vec{m} \leftarrow [0, \ldots, 0]$; $m[start_j + gap_r] \leftarrow 1$ **for all** j ▷ mask selecting the r-th repetition slots
    $E \leftarrow \mathsf{PMult}(C_k, \mathsf{Encode}(\vec{m}))$ ▷ only A_{kR+r}'s bits remain, at slots $start_j + gap_r$
    $D_{kR+r} \leftarrow \sum_{r'=0}^{R-1} \mathsf{Rotate}(E,\ gap_{r'} - gap_r)$ ▷ copy them to all R repetition slots (CAdd; the $r'{=}r$ term is E itself)
  end for
end for
Output: $\{D_0, \ldots, D_{KR-1}\}$ ▷ $D_i = A_i$ with every bit repeated R times, exactly as repetitive encoding

Algorithm 5 reverses Algorithm 4 on the server side: for each compressed ciphertext C_k and each r, it masks the slots of the r-th repetition (one PMult) and copies them into the other $R-1$ repetition slots with $R-1$ rotations and additions, so that D_{kR+r} encrypts exactly what the client would have sent under repetitive encoding of A_{kR+r}.

F ADDITIONAL EVALUATIONS

F.1 SILENTWOOD'S PERFORMANCE ACROSS VARIOUS MODEL SIZES

Bench.	Inference Time			Model Accuracy			Inference Time			Model Accuracy		
	100T	200T	300T	100T	200T	300T	7D	11D	15D	7D	11D	15D
Spam	3.82s	5.49s	6.68s	0.94	0.94	0.94	3.82s	4.42s	4.92s	0.94	0.94	0.94
Steel	1.81s	2.33s	2.69s	1.00	1.00	1.00	1.81s	1.79s	1.73s	1.00	1.00	1.00
Breast	1.57s	1.69s	1.82s	0.67	0.67	0.67	1.57s	1.45s	1.46s	0.67	0.67	0.67
Heart	3.76s	4.97s	6.67s	0.74	0.74	0.74	3.76s	3.18s	3.86s	0.74	0.74	0.74
Defect	6.92s	12.73s	14.39s	0.79	0.81	0.80	6.92s	11.23s	16.38s	0.79	0.81	0.80
Bank	9.39s	10.66s	12.42s	0.99	0.99	0.99	9.39s	12.47s	12.51s	0.99	0.99	0.99
Pendigits	2.17s	2.33s	2.40s	0.99	1.00	1.00	2.17s	2.22s	2.01s	0.99	1.00	1.00
Ailerons	7.73s	10.59s	12.34s	0.87	0.88	0.88	7.73s	11.99s	15.12s	0.87	0.87	0.87
Credit	1.51s	1.59s	1.58s	1.00	1.00	1.00	1.51s	1.45s	1.46s	1.00	1.00	1.00
Satellite	1.63s	1.69s	1.87s	0.99	0.99	0.99	1.63s	1.62s	1.57s	0.99	0.99	0.99
Elevators	8.39s	10.82s	14.49s	0.80	0.80	0.80	8.39s	15.38s	18.02s	0.80	0.80	0.80
Telescope	1.33s	1.41s	1.54s	1.00	1.00	1.00	1.33s	1.27s	1.24s	1.00	1.00	1.00
MFeat	6.81s	6.68s	7.55s	0.93	0.93	0.93	6.81s	6.13s	6.19s	0.93	0.93	0.93
AVG.	4.37s	5.61s	6.65s	0.90	0.90	0.90	4.37s	5.74s	6.65s	0.90	0.90	0.90

Table 8: Inference time over various numbers of trees (100T~ 300T) and maximum tree depths (7D~15D).

Inference Time: Table 8 shows the average inference time across various numbers of trees (100 ~ 300). The average computation time roughly increased linearly with the number of trees, as their number of rotations to handle SumPath also increased linearly. In Table 8, we also measured the change in inference time across various maximum tree depths (7, 11, 15). The average inference time increased with depth (not due to BCC, but due to SumPath computation). The increase between 11 and 15 is relatively bigger than that between 7 and 11.

Default Model Accuracy: Table 8 also illustrates the accuracy of each trained XGBoost model (measured using the testing datasets) across various numbers of trees and maximum tree depths. Note that these measured model accuracies are affected only by the training parameters, not by the optimization techniques of SilentWood. The average model accuracy of all benchmarks was 0.9 for all numbers of trees (100~300) and all maximum tree depths (7~15). This implies that 100 trees with a maximum depth of 7 are generally a sufficient setup to achieve the best accuracy for private XGBoost inference.

Table 9: Comparison of inference time across various numbers of trees.

	Baseline				SilentWood			
Dataset	**10T**	**50T**	**100T**	**200T**	**10T**	**50T**	**100T**	**200T**
spam	78.50s	198.29s	309.42s	523.18s	2.22s	4.49s	8.05s	10.39s
steel	9.08s	28.05s	46.68s	74.32s	1.48s	1.90s	2.21s	2.91s
cancer	9.09s	20.69s	24.82s	43.12s	1.50s	1.75s	1.85s	2.16s
heart	94.93s	230.22s	306.98s	506.65s	2.36s	5.76s	6.90s	9.56s
defect	118.81s	449.49s	833.95s	1488.26s	2.75s	7.60s	11.79s	19.28s
bank	163.85s	606.74s	1183.46s	1729.59s	3.40s	10.77s	17.76s	23.67s
pendigits	29.35s	90.93s	117.17s	170.98s	1.73s	2.45s	2.71s	3.11s
ailerons	153.87s	532.60s	1070.04s	1537.48s	3.35s	8.23s	14.68s	21.71s
creditcard	5.58s	7.81s	15.78s	27.29s	1.45s	1.59s	1.75s	2.11s
satellite	11.74s	20.43s	39.61s	45.20s	1.54s	1.74s	2.05s	2.09s
elevators	152.44s	630.26s	1185.83s	1683.95s	2.97s	8.95s	15.01s	29.89s
telescope	5.69s	8.48s	13.24s	21.70s	1.39s	1.53s	1.70s	2.04s
mfeat	215.13s	435.30s	586.70s	963.95s	5.37s	10.88s	12.41s	15.43s
AVG	80.62s	250.72s	441.05s	678.13s	2.42s	5.20s	7.61s	11.10s
Speedup	–	–	–	–	**33.26×**	**48.18×**	**57.99×**	**61.08×**

Inference Time Comparison between Baseline and SilentWood: Table 9 reports the inference time (in seconds) of Baseline and SilentWood at RTT=50ms for 10, 50, 100, 200 trees (these supplementary runs were performed on an Intel Ultra 7 255H, 16 Cores). Even when the number of trees is small, SilentWood significantly outperforms the Baseline. This is because Baseline's MultiplyPath requires large cryptographic parameters to support long multiplicative chains of homomorphic operations. In contrast, SilentWood's path evaluation requires only additive chains, consuming a negligible amount of noise budget. Regardless of the number of trees, the maximum depth of the trees is set as a training hyperparameter, capped at 7 in this evaluation. As the tree count grows,SilentWood's speedup increases because the benefits of our node clustering and BCC become more pronounced.

F.2 Overhead of Ciphertext Compression and Blind Code Conversion

Bench.	SoK-GGG		Baseline + B + K		Baseline + B + K + C	
	16-bit	32-bit	16-bit	32-bit	16-bit	32-bit
Spam	181.2MB	182.4MB	26.7MB	85.7MB	5.8MB	22.8MB
Steel	35.0MB	36.3MB	26.7MB	85.7MB	4.5MB	16.3MB
Breast	18.3MB	9.8MB	26.7MB	85.7MB	4.5MB	12.3MB
Heart	219.4MB	208.8MB	26.7MB	85.7MB	3.2MB	7.1MB
Defect	324.9MB	322.5MB	26.7MB	85.7MB	3.4MB	10.0MB
Bank	401.6MB	381.2MB	26.7MB	85.7MB	4.7MB	17.5MB
Pendigits	64.8MB	66.6MB	26.7MB	85.7MB	3.2MB	8.4MB
Ailerons	359.4MB	358.5MB	26.7MB	85.7MB	4.7MB	16.5MB
Credit	17.6MB	13.5MB	26.7MB	85.7MB	4.5MB	12.3MB
Satellite	27.5MB	30.6MB	46.4MB	85.7MB	4.5MB	16.3MB
Elevators	401.9MB	390.8MB	26.7MB	85.7MB	3.4MB	8.7MB
Telescope	12.6MB	12.1MB	26.7MB	85.7MB	3.2MB	5.8MB
MFeat	346.5MB	341.5MB	26.7MB	85.7MB	7.1MB	43.8MB
AVG.	185.5MB	180.8MB	28.22MB	85.75MB	4.34MB	15.20MB

Table 10: Communication cost (i.e., exchanged data) between the server and client over various feature bit-lengths

Network Overhead of Ciphertext Compression: Table 10 shows the total size of data exchanged between the server and client during XGBoost inference, where Baseline + B + K + C and Baseline + B + K are SilentWood w/o ciphertext compression. Generally, SilentWood's ciphertext sizes were much smaller than those of SoK-GGG's. For SilentWood w/o compression, the exchanged ciphertext size increased with the feature data size. Without the compression technique, the ciphertext sizes remain constant across all datasets. This is mainly because the client's number of ciphertexts for the initial query matched the number of CW-encoded data bits. For BCC and the final inference response, only a single ciphertext was needed, as all intermediate data fit within 16,384 slots of a single ciphertext. By contract, SilentWood's ciphertext compression could reduce the number of required ciphertexts by a factor of repetitive data encoding, where the repetition was calculated by dividing the total number of slots in a ciphertext by: the total number of feature types multiplied by the CW-encoded bit-length. On average, using ciphertext compression reduced the ciphertext size from 85.75MB to 15.20MB in the case of 32-bit feature data bit-length. Table 11 shows the computation overheads for the server's ciphertext decompression (Algorithm 5 in §E); the average times are 142 ms and 241 ms for 16-bit and 32-bit data bit lengths, respectively.

Computation Overhead of Blind Code Conversion: Table 11 also shows the computation overheads BCC (2 and 3). A majority of the overhead came from the rotation operations to permute real SumPaths. The average times are 2282ms and 2362ms for 16-bit and 32-bit data bit lengths.

Peak Memory Usage: Table 12 measures the peak memory usage of the server and client in MB. SilentWood's average peak memory usage is about 4 6 times less than Baseline. The server used the majority of memory for FHE computations. Both Baseline and SilentWood actively run rotations to compute either MultiplyPath or SumPath, as well as to aggregate tree scores, which require significantly more rotations than decompression. Their Galois key sizes are 784.06MB (Baseline) and 195.03MB (SilentWood), each.SilentWood's is smaller due to its smaller cryptographic parameters.

Bench.	Cipher Decompression		BCC Computation	
	16-bit	32-bit	16-bit	32-bit
Spam	80ms	102ms	1608ms	1629ms
Steel	150ms	189ms	717ms	711ms
Breast	126ms	207ms	453ms	429ms
Heart	205ms	310ms	2314ms	2358ms
Defect	149ms	250ms	4248ms	4454ms
Bank	145ms	174ms	4713ms	4717ms
Pendigits	156ms	351ms	833ms	832ms
Ailerons	131ms	180ms	4223ms	4495ms
Credit	111ms	194ms	379ms	380ms
Satellite	112ms	178ms	498ms	518ms
Elevators	176ms	361ms	5916ms	6081ms
Telescope	209ms	412ms	335ms	367ms
MFeat	90ms	224ms	3433ms	3733ms
AVG.	142ms	241ms	2282ms	2362ms

Table 11: The computation overhead of ciphertext decompression and BCC.

Table 12: Server and client's peak memory usage of Baseline and SilentWood.

	Baseline		SilentWood	
Dataset	**Server**	**Client**	**Server**	**Client**
spam	15438.8MB	413.6MB	3359.8MB	63.7MB
steel	6542.3MB	413.5MB	1357.3MB	58.2MB
cancer	4543.5MB	413.5MB	1009.6MB	58.6MB
heart	23856.0MB	413.6MB	4563.3MB	63.2MB
defect	28849.3MB	413.3MB	6231.1MB	55.6MB
bank	29034.2MB	412.8MB	7832.8MB	59.2MB
pendigits	8547.7MB	413.2MB	1669.4MB	53.7MB
ailerons	29029.4MB	412.7MB	7594.2MB	59.8MB
creditcard	4173.5MB	413.4MB	962.3MB	58.4MB
satellite	5604.0MB	413.3MB	1156.8MB	58.8MB
elevators	28985.5MB	412.6MB	8464.4MB	55.8MB
telescope	4072.9MB	413.5MB	935.8MB	53.6MB
mfeat	29082.8MB	412.9MB	6887.6MB	85.8MB
AVG	**16750.8**	**413.2**	**4001.9**	**60.3**

F.3 Further Analysis of Node Clustering

	Spam	Steel	Breast	Heart	Defect	Bank	PenDigits	Ailerons	Credit	Satellite	Elevators	Telescope	MFeat
Features	57	33	30	13	32	21	16	40	30	36	18	11	64
Classes	2	2	2	5	2	2	2	2	2	2	2	2	10
Samples	4,601	1,941	569	294	8,191	10,885	10,992	14,750	14,143	5,100	16,599	19,020	2,000

Table 13: Specification of benchmarks

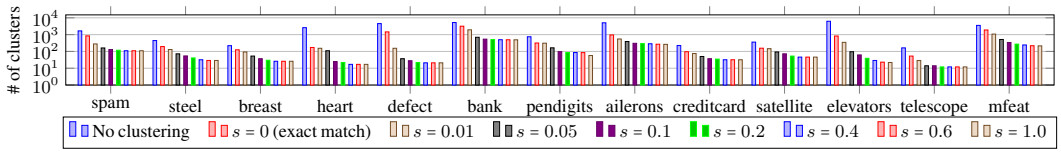

Figure 9: The number of node clusters across various node clustering intensities.

Number of Node Clusters: Table 13 summarizes the properties (i.e., features, classes, and sample sizes) of the 13 UCI datasets Markelle Kelly et. al. (2024) used for SilentWood's evaluation. Figure 9 shows the number of node clusters across various clustering intensities s (0.0 ~ 1.0), where two nodes n_i and n_j are co-clustered if $\frac{|n_i.val - n_j.val|}{\textbf{MAX}(n_i.name) - \textbf{MIN}(n_i.name)} < s$. The success of clustering gradually decreased as we increased the intensity s. For example, when intensity $s = 0.2$, on average, 2,388

Val. Set	Metric	Benchmark												
		Spam	Steel	Breast	Heart	Defect	Bank	PenDigits	Ailerons	Credit	Satellite	Elevators	Telescope	MFeat
5%	Node Reduction	71%	91%	84%	98%	74%	54%	79%	88%	66%	81%	92%	94%	76%
	Test Accuracy	91%	100%	63%	72%	61%	90%	98%	83%	100%	100%	74%	100%	91%
	Abortion	30%	0%	9%	14%	70%	74%	38%	52%	0%	10%	57%	0%	27%
10%	Node Reduction	69%	91%	84%	94%	79%	62%	77%	89%	66%	79%	90%	94%	71%
	Test Accuracy	93%	100%	61%	72%	62%	91%	97%	84%	100%	100%	73%	100%	91%
	Abortion	40%	0%	9%	29%	60%	61%	50%	56%	0%	13%	57%	0%	30%
20%	Node Reduction	72%	91%	78%	96%	76%	68%	71%	89%	66%	75%	89%	94%	65%
	Test Accuracy	95%	100%	60%	81%	82%	99%	99%	88%	100%	99%	80%	100%	94%
	Abortion	35%	0%	23%	14%	60%	55%	56%	52%	0%	26%	64%	0%	42%
30%	Node Reduction	74%	91%	72%	93%	81%	62%	69%	89%	62.5%	75%	90%	94%	66%
	Test Accuracy	98%	100%	60%	81%	82%	93%	97%	85%	100%	100%	84%	100%	89%
	Abortion	40%	0%	23%	43%	50%	68%	69%	52%	9%	26%	79%	0%	45%
35%	Node Reduction	74%	91%	70%	93%	82%	62%	69%	89%	62.5%	72%	90%	94%	65%
	Test Accuracy	95%	100%	80%	81%	85%	93%	95%	85%	100%	100%	76%	100%	91%
	Abortion	40%	0%	27%	43%	50%	65%	69%	56%	9%	35%	79%	0%	52%

Table 14: Node clustering with various validation set sizes (5%~35%).

nodes in XGBoost were clustered into 46 node clusters (i.e., $\frac{2388}{46}$ = 51.9x node reduction rate). While the greater clustering intensity provided a higher opportunity for node clustering, the effectiveness of clustering (i.e., node reduction rate) did not increase further when s was greater than 0.4.

Various Validation Set Sizes: Table 14 shows model accuracy, clustering efficiency, and merge abortion rate across various validation set sizes (5%~35%). A small validation set (5%) leads to unstable accuracy, which improves and stabilizes as its size increases to 20%. At 35%, the test-set size drops too much, causing instability in test accuracy. As the validation-set size increases, clustering efficiency declines, and the abortion rate increases, which is natural. Our default (20% validation set, 20% test set) balances these factors effectively.

	Comparison Time	Total Inference Time	Fraction of Comparison Time
Spam	5.26s	6.14s	85.6%
Steel	4.46s	5.39s	82.7%
Breast	4.49s	5.32s	84.4%
Heart	20.37s	24.00s	84.9%
Defect	5.55s	6.41s	86.6%
Bank	5.67s	6.58s	86.2%
PenDigits	4.83s	5.82s	83.0%
Ailerons	5.62s	6.56s	85.7%
Credit	4.42s	5.24s	84.3%
Satellite	4.68s	5.49s	85.2%
Elevators	5.56s	6.49s	85.7%
Telescope	4.03s	5.00s	80.1%
MFeat	37.51s	45.37s	82.7%
AVG.	8.65s	10.29s	84.1%

Table 15: SoK-GGG's Comparison Times and Total Inference Times

Applicability of Node Clustering to SoK-GGG: As we showed in Table 5, SoK-GGG's computation time (without the communication cost) for private inference is large: 5.64 seconds on average. This cost primarily includes the time for garbled circuit operations in the comparison phase: OT extension for input values, circuit garbling (server-side), and circuit evaluation (client-side). Table 15 additionally shows SoK-GGG's inference time and the fraction of time spent on comparisons at tree nodes. If we apply node clustering to SoK-GGG, the reduction ratio of nodes linearly reduces the amount of computation spent on the OT extension. Let α be the ratio of comparisons before node clustering to comparisons after node clustering, and let β be the time spent on comparisons relative to the total time spent in SoK-GGG. Let t be the time spent in SoK-GGG's garbled circuit for comparisons. Then, SoK-GGG's computation time for comparison with node clustering is expected to run in $\frac{(\beta \cdot t)}{\alpha} + (1 - \beta) \cdot t$.

Table 16: Communication size and end-to-end latency under different bandwidths.

	Baseline			SilentWood (without compression)			SilentWood		
Dataset	**Comm. Size**	**B = 1** Gb/s	**100** Mb/s	**Comm. Size**	**1** Gb/s	**100** Mb/s	**Comm. Size**	**1** Gb/s	**100** Mb/s
spam	84.8MB	312.0s	314.1s	85.7MB	10.3s	12.4s	22.8MB	8.2s	8.9s
steel	84.8MB	49.4s	51.2s	85.7MB	4.9s	6.9s	16.3MB	4.0s	4.3s
cancer	84.8MB	27.5s	29.3s	85.7MB	4.6s	6.5s	12.3MB	3.5s	3.6s
heart	84.8MB	309.6s	311.8s	85.7MB	9.3s	11.4s	7.1MB	8.1s	8.2s
defect	84.8MB	836.5s	838.5s	85.7MB	14.3s	14.3s	10.0MB	12.7s	13.1s
bank	84.8MB	1186.1s	1187.9s	85.7MB	19.9s	22.6s	17.5MB	16.1s	18.3s
pendigits	84.8MB	119.9s	121.9s	85.7MB	6.4s	7.5s	8.4MB	4.4s	4.9s
ailerons	84.8MB	1072.7s	1074.6s	85.7MB	18.6s	20.7s	16.5MB	15.4s	16.1s
creditcard	84.8MB	18.4s	20.5s	85.7MB	4.3s	6.2s	12.3MB	3.4s	4.3s
satellite	84.8MB	31.0s	33.0s	85.7MB	4.5s	6.4s	16.3MB	3.6s	3.8s
elevators	84.8MB	1188.4s	1190.6s	85.7MB	17.4s	19.6s	8.7MB	14.9s	16.1s
telescope	84.8MB	15.8s	17.8s	85.7MB	4.2s	6.1s	5.8MB	3.2s	3.6s
mfeat	84.8MB	589.3s	591.1s	85.7MB	20.4s	21.6s	43.8MB	17.2s	18.3s
SilentWood's AVG	**84.8MB**	**442.8s**	**444.8s**	**85.7MB**	**10.7s**	**12.5s**	**15.2MB**	**8.8s**	**9.5s**
Speedup	n/a	50.3×	46.8×	n/a	1.2×	1.3×	n/a	–	–

F.4 End-to-end Inference Time across Different Bandwidths

Table 16 is an evaluation for end-to-end inference time and communication data size (C-Size) at latency=200ms and bandwidth B=1Gb/s and 100Mb/s (for these measurements we used Intel Ultra 7 255H, 16 Cores). SilentWood's (with ciphertext compression) average communication size is smaller than Baseline (15.2MB < 84.8MB), and its average inference speed is 46.8x faster even at the biggest RTT (200ms) and the smaller bandwidth (100Mb/s). Without ciphertext compression, SilentWood's average inference speed is still 35.6x faster than Baseline (444.8s > 12.5s) at the network condition. Without ciphertext compression, SilentWood's total communication data size is only marginally bigger than Baseline (85.7MB > 84.8MB), because the BCC protocol's exchanged intermediate path results all compactly fit in 1~2 ciphertexts.