LCPDTE: Low-Complexity Private Decision Tree Evaluation over Homomorphic Encryption

Dongjin Park
Chung-Ang University
Seoul, Republic of Korea
thrudgelmir@cau.ac.kr

Gyeongwon Cha
Chung-Ang University
Seoul, Republic of Korea
dbfldk20@cau.ac.kr

Joon-Woo Lee*
Chung-Ang University
Seoul, Republic of Korea
jwlee2815@cau.ac.kr

Abstract

As machine-learning-as-a-service (MLaaS) becomes ubiquitous, protecting model queries via private inference is increasingly critical. Existing homomorphic encryption (HE)-based protocols for Private Decision Tree Evaluation (PDTE) have server complexity that scales at least as $O(2^D)$ in the tree depth D, so the cost of evaluating each tree grows exponentially with depth; in gradient boosted decision tree (GBDT) ensembles, where predictions aggregate the outputs of many trees, this per-tree cost is directly amplified.

In this paper, we present a non-interactive HE-based PDTE protocol built on the CKKS scheme with an end-to-end complexity of $O(p\sqrt{2^D})$, where p is the input bit-length. To the best of our knowledge, this is the first HE-based PDTE scheme that asymptotically improves over the $O(2^D)$ dependence on D while remaining non-interactive. We address two depth-driven sources of $O(2^D)$ dependence in existing protocols: we use the One-Branch-Only (OBO) paradigm from PROBONITE for comparisons, and we design the Baby-Step Giant-Step based Branch Selection algorithm for traversal. To further exploit the structure of GBDT ensembles, we deploy the batched bootstrapping technique by applying level-major tree evaluation.

Our experimental results show that, at depth $D = 12$, our protocol reduces communication by 8.38× and runtime by 7.74× compared to FASTER, which is the fastest prior HE-based non-interactive PDTE baseline in our amortized setting, and the advantage increases as D grows. These results suggest that our design provides a practical path toward depth-scalable HE-based PDTE for large boosted ensembles.

CCS Concepts

• **Security and privacy → Privacy-preserving protocols; Cryptography**.

Keywords

Decision Trees, Private Decision Tree Evaluation, Non-Interactive Protocols, Homomorphic Encryption, CKKS

1 Introduction

As artificial intelligence services continue to expand, many companies are increasingly offering cloud-based machine learning services. In this setting, decision-tree-based models are widely adopted across various industries, including healthcare and finance, due to their simple structure, interpretability, and ease of training. However, in order to utilize such services, users are required to entrust sensitive data to remote servers, which raises persistent concerns regarding data privacy and potential information leakage. Conversely, disclosing the model structure or parameters to users may expose a company's core intellectual property, and is therefore generally avoided from the service provider's perspective.

To address these conflicting requirements, private decision tree evaluation (PDTE) protocols have been proposed. Existing PDTE approaches can be broadly divided into interactive and non-interactive protocols. Interactive PDTE protocols typically rely on cryptographic primitives such as garbled circuits or secure multi-party computation [5, 14, 19, 21–23] . These approaches enable the client and the server to jointly evaluate the decision tree through multiple rounds of interaction, while providing strong security guarantees. However, the need for frequent interaction incurs substantial communication overhead and latency, which limits their practicality in high-latency networks or large-scale deployment scenarios. Accordingly, this work focuses on non-interactive PDTE protocols.

Non-interactive PDTE protocols are primarily based on homomorphic encryption (HE). In these protocols, the client encrypts its attribute vector using HE and transmits the resulting ciphertexts to the server. The server then performs a homomorphic evaluation of the decision tree and returns a ciphertext encoding the inference result. As a consequence, the efficiency of HE-based non-interactive PDTE is largely determined by the cost of homomorphic computation performed on the server side. This observation has motivated prior research to concentrate on optimizing the homomorphic operations that constitute PDTE evaluation.

Despite extensive efforts toward optimizing non-interactive PDTE, existing protocols still exhibit an inherent $O(2^D)$ dependence on the tree depth D. On the comparison side, the one-branch-only (OBO) method introduced in PROBONITE [3] is the only approach that reduces the number of comparisons to D; the remaining protocols do not eliminate the $O(2^D)$ factor in the comparison stage and instead focus on reducing the runtime constants or the communication cost of the underlying comparison procedure. On the traversal side, SumPath method [2, 16, 17, 20] is the only method that avoids the exponential dependence in path evaluation, while other works primarily reduce the multiplicative depth or achieve constant-factor improvements. However, OBO and SumPath cannot be combined, and thus existing non-interactive HE-based PDTE protocols ultimately cannot avoid an $O(2^D)$ cost in the tree depth D.

1.1 Our Contributions

We propose LCPDTE[1], a CKKS-based non-interactive PDTE protocol with end-to-end server complexity $O(p \cdot \sqrt{2^D})$ (for p-bit inputs).

*Corresponding author.

[1] Source code is available at: https://github.com/thrudgelmir/LCPDTE.

	BPDTE_CW	BPDTE_RCC	RDCMP_ASM	Ours
Time(ms)	326.5083	439.1848	905.1788	42.21
Speedup	7.74×	10.4×	21.45×	1×

Table 1: Amortized runtime and speedup over our method.

This reduces the dominant exponential term in prior HE-based non-interactive PDTE from $O(2^D)$ to $O(2^{D/2})$. Our experiments show that LCPDTE improves over existing protocols by up to 21.45× and achieves the highest amortized throughput for sufficiently deep decision trees (see Table 1).

OBO Paradigm over CKKS. Recent non-interactive PDTE protocols [2, 8, 16] typically achieve efficiency by leveraging SIMD capabilities over leveled schemes. In contrast, we adopt the OBO paradigm to limit the number of comparisons to the tree depth D, avoiding the exponential $O(2^D)$ scaling of existing protocols. However, OBO introduces strong sequential dependence, leading to a large multiplicative depth and thus conflicting with the shallow circuit constraints of leveled schemes. We therefore instantiate OBO over CKKS with bootstrapping. This design trades heavier HE primitives and bootstrapping for an algorithmic reduction in comparison complexity from $O(2^D)$ to D.

Efficient Square-Root Traversal. The conventional OBO evaluation relies on MulPath traversal, which demands $O(2^D)$ multiplications and a multiplicative depth of D. We exploit a key structural property of OBO—that it yields exactly one comparison outcome per level—to redesign the traversal without incurring additional overhead. We present two complementary techniques: GenOH reduces traversal depth to $\log D$, and BsgsBBS reduces the traversal cost from $O(2^D)$ to $O(p\sqrt{2^D})$. Together, these algorithms yield logarithmic depth and a square-root reduction in the dominant 2^D term, resulting in an end-to-end server complexity strictly below $O(2^D)$ in D.

High-Precision Comparison. To support standard 32-bit inference and the ciphertext–ciphertext comparisons required by OBO, we develop a fast comparison algorithm based on bit decomposition. For an p-bit comparison, it reduces the number of homomorphic multiplications from $3p-2$ to $2p-\log p$, improving over RDCMP [16].

Extension to GBDT. In practical deployments, decision tree models are often used as ensembles, and gradient boosted decision tree (GBDT) is a representative example. We extend LCPDTE to GBDT using a level-major evaluation strategy and apply batched-bit bootstrapping across trees, reducing the bootstrapping cost by about 3.6×.

1.2 Technical Overview

The proposed LCPDTE protocol is a non-interactive, cryptographic HE-based PDTE protocol that is scalable to deep decision trees. For a decision tree of depth D and precision p, the protocol requires D homomorphic comparisons and achieves a traversal complexity of $O(p \cdot \sqrt{2^D})$ while consuming only $O(\log D)$ multiplicative depth. By reducing both the number of key-switching operations and the required multiplicative depth, the protocol allows for smaller cryptographic parameters, leading to a non-negligible performance improvement in practice. The core components of our protocol consist of homomorphic comparison, one-hot vector generation, and a baby-step giant-step–based branch selection procedure. A brief technical overview is provided below, with detailed descriptions given in Sections 3 and 4. For readers unfamiliar with the OBO method, a detailed explanation is provided in Section 2.4.

High-Precision Homomorphic Comparison. Our comparison algorithm uses a bit-decomposed representation for comparing high-precision integers. Accordingly, a p-bit integer is decomposed into bits and stored across p ciphertexts, where the bits corresponding to the same integer are aligned at the same slot index. The algorithm compares two integers in a manner analogous to carry propagation, which is equivalent to evaluating the following expression:

$$\sum_{0 \le i < p} (x_i - y_i) \cdot \prod_{p-i \le j < p} \mathbb{I}[x_j = y_j].$$

This expression is evaluated using a divide-and-conquer approach, and by combining the base case with the immediately following combine phase, the algorithm achieves a lower computational cost than existing folklore-based comparison algorithms.

One-Hot Vector Generation with $\log D$ Multiplicative Depth. To reduce the multiplicative depth required for one-hot vector generation, we adopt a different approach from the conventional method. Specifically, when constructing each element of a d-level one-hot vector (Eq. (1)), instead of directly multiplying the basis factors S_i and $1-S_i$, we begin by constructing a linear basis of the one-hot vector. This linear basis consists of all product combinations of the $\{S_i\}_{0 \le i < d}$, and by carefully reusing the product combinations computed at previous levels, it can be constructed using only $\log d$ multiplicative depth. Our one-hot vector generation algorithm is described in Section 4.2.

Blind Branch Selection via the BSGS optimization. Independently of the reduction in multiplicative depth, the generation of one-hot vectors still requires 2^d key-switching operations to construct the underlying basis. We observe that each entry of the one-hot vector is expressed as a product of the factors $S_i, 1-S_i$, and that these entries are ultimately used in inner product operations with the nodes at depth d of the decision tree.

Based on this observation, our BSGS optimization partitions the 2^d entries of the one-hot vector into $2^{d/2}$ blocks of length $2^{d/2}$ such that all entries within each block share a common factor. In this setting, it suffices to generate only two one-hot vectors of length $2^{d/2}$. The decision tree nodes at depth d are partitioned in the same manner according to their indices, and the next node is selected via two inner product operations with lazy relinearization. Since the attributes and thresholds of each decision node are decomposed into p bits, each inner product is invoked p times. As a result, the total number of key-switching operations is bounded by $O(p \cdot 2^{D/2})$. Further details are provided in Section 4.3.

1.3 Related Work

In this section, we review prior HE-based protocols for non-interactive PDTE. Prior work differs mainly in (i) how it computes comparisons and (ii) how it traverses the tree from those outcomes. To make

State	Choice	[17]	[20] PDT-BIN	[20] PDT-BIN(v)	[20] PDT-INT	[3]	[7]	[18]	[11]	[2]	[8]	[16] ASM	[16] ESM	Ours
Comparison	All-node	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	
	OBO					✓								✓
Traversal	SumPath (client)	✓			✓					✓			✓ shuf	
	SumPath (server)							✓ poly	✓ eq			✓ poly		
	MulPath		✓ DAG	✓ DAG		✓	✓							✓ BSGS
Scheme		BGV	BGV	TFHE	BGV	TFHE	TFHE	TFHE	TFHE	FV	FV	FV	FV	CKKS
Goal		Latency	Amort.	Latency	Amort.	Latency	Latency	Latency	Latency	Latency	Amort.	Trade-off.	Trade-off.	Amort.

Table 2: Design-space summary of non-interactive HE-based PDTE protocols. Each column corresponds to a method; when a single paper presents multiple variants, we list them in separate columns. Each work marks a primary choice (✓) for comparison and traversal; inline tags indicate key variants (shuf, poly/eq, DAG, BSGS).

Stage	Variant	#Mult	#Level
Comparison	All-node	$O(2^D \cdot \text{cmp})$	cmp_level
	OBO	$D \cdot$ cmp	$D \cdot$ cmp_level
Traversal (SumPath)	Client-side	0	0
	Client-side (+ shuffle)	0	$\log B$
	Server-side (poly)	$O(2^D \cdot \sqrt{D})$	$\log D$
	Server-side (eq)	$O(2^D \cdot \text{cmp})$	cmp_level
Traversal (MulPath)	Plain	$O(2^D)$	D
	DAG	$O(D \cdot 2^D)$	$\log D$
	BSGS	$O(p \cdot \sqrt{2^D})$	$\log D$

Table 3: Stage-wise complexity summary. D is the tree depth and B is the batch size. cmp **and** cmp_level **denote the multiplication cost and level consumption of one comparison, respectively. Boldface entries correspond to our method.**

these differences explicit, we decompose PDTE into the comparison state, which produces node-wise comparison outcomes, and the traversal state, which uses those outcomes to select the correct path and leaf. Table 2 summarizes the main design choices of representative methods, and Table 3 summarizes their stage-wise cost scaling.

Comparison state. The comparison state can be categorized into all-node methods and OBO methods [3]. All-node methods evaluate comparisons for all internal nodes; since the number of internal nodes is $O(2^D)$, their comparison cost scales with 2^D, and most works in this class focus on improving the comparison primitive and packing rather than reducing the node count itself. In contrast, OBO method performs only one comparison per level, reducing the number of comparisons from $O(2^D)$ to D (See Sec. 2.4).

Level-Up [2] is a notable special case among all-node methods. It still evaluates all-node comparisons, but it targets latency by packing comparisons into the SIMD slot space so that the comparison step can be completed with a single invocation (up to packing constraints). This packing introduces substantial data-movement overhead: to make the packed outcomes usable for traversal, the evaluator must reorganize values across slots, which requires $O(2^D)$ rotations. In short, Level-Up reduces the number of comparison invocations but pays an $O(2^D)$ rotation overhead.

Traversal state. The traversal state is commonly implemented by SumPath or MulPath. SumPath avoids multiplications when forming path aggregates, but it requires a conversion step that leaves only the correct path in a form suitable for selecting the output [2, 11, 16–18, 20]. Depending on the protocol, this conversion is done on the server side (e.g., via polynomial evaluation or equality checks) or on the client side (optionally with shuffling).

SilentWood [15] is another notable special case. Although it is interactive, it largely follows the same scope as HE-based non-interactive PDTE schemes. It completes the post-processing step in SumPath via a single round of communication, while adopting a Level-Up-style comparison state that requires $O(2^D)$ rotations.

MulPath multiplies comparison outcomes along each path, which directly yields a selector for the correct leaf without an additional conversion step[3, 7, 20]. However, the plain MulPath requires $O(2^D)$ multiplications and consumes D levels of multiplicative depth. A DAG-based MulPath variant reduces multiplicative depth to $\log D$, but increases the number of multiplications, yielding a clear trade-off [20].

Our approach. Existing protocols include components that avoid the full 2^D scaling in one stage: OBO reduces the number of comparisons to D, while SumPath avoids multiplications during path formation. However, these components do not combine well in prior non-interactive designs. OBO relies on per-level branch selection, while SumPath is built around producing a final aggregate, and this mismatch prevents their direct combination. As a result, prior protocols still incur an exponential 2^D factor once comparison and traversal are considered together. Our protocol breaks this barrier by enabling the OBO method under CKKS and by implementing per-level branch selection using a BSGS-based construction, reducing the end-to-end dependence on depth from $O(2^D)$ to $O(p \cdot \sqrt{2^D})$.

For a fair comparison, we focus on recent Ring LWE-based non-interactive homomorphic PDTE protocols [8, 16]. We exclude [2], since [8, 16] have already demonstrated superior amortized performance, and [15], as it is an interactive protocol.

2 Preliminaries

Let N be a power-of-two integer and let $Q > 0$ be an integer, which are parameters related to the efficiency of RLWE–based schemes. Vectors are denoted by bold lowercase letters (e.g., $\boldsymbol{v}, \boldsymbol{w}$). For a nonnegative integer $b \in \mathbb{Z}_{\geq 0}$ with binary length n, its binary

representation is denoted by $(b_{n-1} \cdots b_0)_2$, such that $b = \sum_{i=0}^{n-1} b_i 2^i$ with $b_i \in \{0, 1\}$. We denote by $\mathbb{I}[\cdot]$ the indicator function, which evaluates to 1 if the condition inside the brackets holds, and to 0 otherwise.

2.1 The CKKS Schemes

We present only the components of CKKS that are essential for understanding the proposed protocol. Further details of the CKKS scheme are provided in the appendix A.

CKKS is a homomorphic encryption scheme that supports approximate arithmetic over encrypted complex vectors. A single ciphertext can pack and encrypt up to $N/2$ complex values, and the scheme supports the following representative operations.

- Vectorized arithmetic operation : element-wise addition, subtraction, and multiplication over encrypted vectors in $\mathbb{C}^{N/2}$, denoted by Add, Sub, and Mult, respectively. When one of the operands is a plaintext, the corresponding operations are denoted by PAdd, PSub, and PMult.
- Key switching: a procedure that transforms a ciphertext encrypted under a secret key s' into an equivalent ciphertext under a secret key s while preserving the underlying message, denoted by $\mathsf{KS}_{s',s}$.
- Bootstrapping: a procedure that takes as input a ciphertext and outputs a fresh ciphertext encrypting the same message with its available multiplicative depth restored, denoted by Btp.

Among arithmetic operations, ciphertext–ciphertext multiplication Mult uniquely requires key switching and is therefore the most expensive operation. Since key switching dominates the cost of Mult, we adopt the *lazy relinearization* technique to reduce the number of key-switching calls.

Specifically, homomorphic multiplication first applies a tensor product to two ciphertexts, producing a ciphertext encrypted under s^2. This ciphertext is normally converted back to one encrypted under s via $\mathsf{KS}_{s^2,s}$. Lazy relinearization postpones this conversion by performing subsequent operations directly on ciphertexts encrypted under s^2, thereby reducing the total number of key-switching operations.

2.1.1 Discrete CKKS. Unlike conventional CKKS schemes, our protocol performs homomorphic operations on discrete data. For discrete data processing in CKKS, the message and noise must be explicitly separated, and the accumulated noise must be removed during intermediate computations via a *cleaning* procedure. In our setting, the cleaning procedure is integrated into the bootstrapping operation. When cleaning is performed as part of bootstrapping, we denote the procedure simply by Btp. Otherwise, we explicitly denote it by Clean.

We next formally define the noise-control capability of the cleaning procedure.

Definition 2.1 (Cleaning). Let $A \subset \mathbb{C}$ be a finite set. The CKKS Clean procedure over A satisfies the following property: for any input ciphertext with noise e such that $\|e\|_2 \le B$, the output ciphertext has noise e' satisfying

$$\|e'\|_2 \le \|e\|_2,$$

where $\|\cdot\|_2$ denotes the ℓ_2-norm over $\mathbb{C}$.

2.2 Decision Tree

In this section, we formally define decision trees. Our notation and definitions closely follow the SortingHat work by Cong et al. [7].

We define a decision tree model as a tuple of functions $\mathcal{M} = (\mathcal{T}, \alpha, \tau, \Lambda)$.

The function $\mathcal{T} : \mathbb{Z}_{>0}^F \to \mathcal{L}$ is an evaluation function that takes an attribute vector as input and outputs the leaf value. The evaluation proceeds by traversing a binary tree composed of decision nodes and leaf nodes. At each decision node, a comparison is performed between the attribute value associated with the node and a threshold, and the result of this comparison determines the next node to visit. Once a leaf node is reached, the leaf value associated with that node is returned as the output.

The functions α, τ, and Λ respectively specify the attribute index, the threshold value associated with each decision node, and the leaf value associated with each leaf node. We index all nodes in the tree in breadth-first search order, assigning index 0 to the root node. For a tree consisting of m decision nodes and $M - m$ leaf nodes, the functions are defined as $\alpha : \{0, \ldots, m-1\} \to \{0, \ldots, F-1\}$, $\tau : \{0, \ldots, m-1\} \to \mathbb{Z}_{>0}$, $\Lambda : \{m, \ldots, M-1\} \to \mathcal{L}$

For notational convenience, we denote by I_d the index set of nodes at level d of the tree. That is, for a tree of depth D, the indices of all decision and leaf nodes can be partitioned by level as $\{0, \ldots, M-1\} = I_0 \| \cdots \| I_D$ where the concatenation operator $\|$ preserves the ordering of indices.

2.3 Gradient-Boosted Decision Tree (GBDT)

Gradient boosting builds a strong predictor by adding many simple models step by step, each one correcting the remaining errors of the previous ones [12]. When the simple models are decision trees, the resulting model is a gradient-boosted decision tree. A single decision tree is simple, but its accuracy is often limited, so in many real deployments GBDT are used instead.

Widely used GBDT systems include, for example, XGBoost and LightGBM [6, 13]. While their training details may differ, inference follows the same high-level pattern: each tree is evaluated on the input independently, and the outputs are then combined to form the final prediction. Even though the trees are trained sequentially, inference has no order dependency across trees, so they can be evaluated in parallel. The remaining differences mainly come from how results are combined and how the final decision is made. For example, binary classification may map the combined score to a probability and apply a threshold, while multi-class classification combines scores per class and selects the largest one.

2.4 One Branch Only Traversal Algorithm

Azogagh et al. proposed the *One Branch Only* (OBO) algorithm, a tree traversal algorithm that designed over the TFHE scheme [3]. Unlike conventional traversal algorithms, which perform comparisons at all decision nodes, the OBO algorithm performs a comparison at only a single decision node at each level of the tree.

OBO Traversal Algorithm.

(1) First, a comparison is performed at the root node. Let the result of this comparison be denoted by S_1.
(2) Blind Branch Selection : The OBO traversal proceeds through the decision tree in a level-by-level manner. At each level, the blind branch selection procedure retains only a single decision node to be visited. Specifically, when traversing level d, a one-hot vector is generated from the decision values $\{S_i\}$ obtained at the previous levels as

$$\bigotimes_{0\le i<d} (S_i, 1-S_i) \tag{1}$$

The index of 1 corresponds to the node to be visited at the next level. Using this one-hot vector, the attribute value and threshold of the node to be visited at level d are extracted. We treat this selection process as a black box; depending on the implementation, it can be realized via inner-product operations. Since all computations are performed over ciphertexts, the extracted threshold remains encrypted even if the original threshold is given in plaintext. Subsequently, a ciphertext–ciphertext comparison is performed on the selected attribute value and threshold to compute the decision value S_d, which is then used to proceed to the traversal of the next level.
(3) Upon reaching the last level, i.e., the leaf nodes, the algorithm computes a plaintext–ciphertext inner product between the final one-hot vector and the vector of inference values associated with the leaves, and outputs the result.

3 Homomorphic Comparison Algorithm

In this section, we describe the homomorphic comparison algorithm used in the PDTE protocol. Most decision tree models perform comparisons on data with 32-bit or 64-bit precision. Supporting such bit precision with conventional CKKS comparison algorithms based on polynomial approximation requires large cryptographic parameters and substantial multiplicative depth, which leads to non-negligible computational overhead. This limitation also arises in PDTE protocols based on exact homomorphic encryption schemes, and prior works have addressed it by reducing the model bit precision or by employing various decomposition-based encoding strategies.

In this work, we propose a high-precision comparison algorithm based on bit-decomposition encoding. Section 3.1 presents a detailed description of the proposed algorithm, while Section 3.2 provides an efficiency analysis. In particular, the last part of Section 3.2 compares the proposed comparison algorithm with those used in previous PDTE protocols.

3.1 Algorithm Description and Optimization

We begin with a folklore algorithm based on bit decomposition. The algorithm computes the following expression, which outputs the indicator function $\mathbb{I}[x > y]$:

$$\sum_{0\le i<p} \mathbb{I}[x_i > y_i] \cdot \prod_{p-i\le j<p} \mathbb{I}[x_j = y_j].$$

Our observation is that, by replacing each factor $\mathbb{I}[x_i > y_i]$ with $(x_i - y_i)$, the above expression evaluates to 1 if $x > y$, 0 if $x = y$, and -1 otherwise.

Claim 1. *The following equality holds.*

$$f_p(x,y) = \sum_{0\le i<p} (x_i - y_i) \cdot \prod_{p-i\le j<p} \mathbb{I}[x_j = y_j] = \begin{cases} 1, & \text{if } x > y, \\ 0, & \text{if } x = y, \\ -1, & \text{if } x < y. \end{cases} \tag{2}$$

Proof. We prove the claim by mathematical induction on p. The base case $p = 1$ is immediate.

For the inductive step, assume that the statement holds for p bits. Then, for $p+1$ bits, the following relation holds:

$$f_{p+1}(x,y) = (x_p - y_p) + \mathbb{I}[x_p = y_p] \cdot f_n([x]_{2^p}, [y]_{2^p}).$$

If $x_p = y_p$, the comparison result is determined by the lower p bits. Otherwise, when $x_p \neq y_p$, the result is uniquely determined by the value of the p-th bit.

Therefore, the function f_p outputs one of the values 1, 0, or -1, depending on whether $x > y$, $x = y$, or $x < y$, respectively. □

Algorithm 1 GEQ

Procedure: Merge($p, \{\mathsf{ct}_i\}_{0\le i<p}, \{\mathsf{ct''}_i\}_{0\le i<p}, \mathsf{flag}$)
Input: A length p, two sequences of ciphertexts $\{\mathsf{ct}_i\}_{0\le i<p}$ and $\{\mathsf{ct}'_i\}_{0\le i<p}$, and a Boolean value flag.
Output: If flag is True, then output ciphertexts L, R, otherwise output R.
if p=2 **then**
  $\Delta_0 \leftarrow \mathsf{Sub}(\mathsf{ct}_0, \mathsf{ct}'_0)$
  $\Delta_1 \leftarrow \mathsf{Sub}(\mathsf{ct}_1, \mathsf{ct}'_1)$
  tmp $\leftarrow \mathsf{Mult}(\Delta_0, \Delta_1)$
  L $\leftarrow$ Compute (2) using tmp.
  if Tail=True **then**
    R $\leftarrow$ Compute (3) using tmp.
    return L, R
  else
    return L
  end if
end if
$\mathsf{L}_0, \mathsf{R}_0 \leftarrow \mathsf{Merge}(p/2, \{\mathsf{ct}_i\}_{0\le i<p/2}, \{\mathsf{ct}'_i\}_{0\le i<p/2}, \text{True})$.
$\mathsf{L}_1, \mathsf{R}_1 \leftarrow \mathsf{Merge}(p/2, \{\mathsf{ct}_i\}_{p/2\le i<p}, \{\mathsf{ct}'_i\}_{p/2\le i<p}, \mathsf{flag})$.
L $\leftarrow \mathsf{Add}(\mathsf{L}_0, \mathsf{Mult}(\mathsf{L}_1, \mathsf{R}_0))$
if flag=True **then**
  return L, $\mathsf{Mult}(\mathsf{R}_0, \mathsf{R}_1)$
else
  return L
end if
Procedure: GEQ($\{\mathsf{ct}_i\}_{0\le i<p}, \{\mathsf{ct}'_i\}_{0\le i<p}$)
Setting: A bit precision $p = 2^k$.
Input: A bit precision p, $\{\mathsf{ct}_i\}_{0\le i<p}$, $\{\mathsf{ct}'_i\}_{0\le i<p}$, where $\mathsf{ct}_i = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{x}_i)$, $\mathsf{ct}'_i = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{y}_i)$, and $\boldsymbol{x}_i, \boldsymbol{y}_i \in \{0,1\}^{N/2}$.
Output: $\mathsf{ct}_{\mathsf{out}} = \mathsf{End} \circ \mathsf{Ecd}(\mathbb{I}[\boldsymbol{x} \ge \boldsymbol{y}])$.
ct $\leftarrow \mathsf{Merge}(p, \{\mathsf{ct}_i\}_{0\le i<p}, \{\mathsf{ct}'_i\}_{0\le i<p}, \mathit{False})$
$\mathsf{ct}_{\mathsf{out}} \leftarrow$ Evaluate ϕ on ct
return $\mathsf{ct}_{\mathsf{out}}$

The comparison algorithm takes as input the bit-decomposed ciphertexts of two p-bit integers. Specifically, each of the p ciphertexts encodes a single bit, and the binary representation of a p-bit integer is aligned across ciphertexts at the same slot index (see Section 4.2 for details on the encoding procedure). The algorithm then recursively evaluates Equation 2 by invoking the procedure Merge. It then evaluates a look-up table polynomial defined according to the chosen comparison operator. In this work, we consider the $\geq$ operator, and the corresponding look-up table polynomial ϕ is defined as

$$\phi(x) = 1 - \frac{x}{2}(x-1).$$

The overall comparison algorithm is named GEQ (greater than or equal) to reflect the use of the $\geq$ operator, and its pseudocode is presented in Algorithm 1.

The algorithm evaluates Equation (2) using a divide-and-conquer manner. To reduce the number of key-switching operations, we merge the base case with the immediately following combine phase (Steps 1–12 of Algorithm 1). When $p = 2$, the algorithm first computes the following two intermediate results from ciphertexts encrypting x_0, y_0, x_1, y_1:

$$(x_1 - y_1) + \mathbb{I}[x_1 = y_1] \cdot (x_0 - y_0), \tag{3}$$

$$\mathbb{I}[x_1 = y_1] \cdot \mathbb{I}[x_0 = y_0]. \tag{4}$$

To compute Equations (3) and (4), we rewrite them using the identity

$$\mathbb{I}[a = b] = 1 - (a-b)^2 \qquad \text{for } a, b \in \{0, 1\}.$$

For convenience, we define the following notations:

$$\Delta_0 := x_0 - y_0, \qquad \Delta_1 := x_1 - y_1.$$

With these definitions, Equations (3) and (4) can be rewritten as

$$(3) \;= \Delta_1 + \Delta_0 - \Delta_0 \cdot \Delta_1^2,$$

$$\begin{aligned}(4) \;&= (1-\Delta_0)(1+\Delta_1)(1+\Delta_0)(1-\Delta_1) \\ &= \big(1 - \Delta_0 - \Delta_1 - \Delta_0 \cdot \Delta_1\big)\big(1 + \Delta_0 - \Delta_1 - \Delta_0 \cdot \Delta_1\big).\end{aligned}$$

Therefore, by computing the common term $\Delta_0 \cdot \Delta_1$ once and reusing it, the algorithm can evaluate Equations (3) and (4) using only three homomorphic multiplications in total.

Remark. Algorithm 1 can be naturally generalized to the case where p is not a power of two. Since the goal of the algorithm is simply to evaluate (2) recursively, it suffices to add an additional branch in the Merge procedure that outputs $x_0 - y_0$ when $p = 1$.

3.1.1 Further Optimization. We further reduce the number of key-switching operations in the algorithm by applying the lazy relinearization technique. To this end, we focus on Step 15 of the algorithm, which corresponds to the combine phase. If L_0 is assumed to remain unrelinearized at this stage, the multiplication $\mathsf{Mult}(L_1, R_0)$ can be performed together with its relinearization.

In addition, we reduce the multiplicative depth consumed by the algorithm by one using a simple optimization. Specifically, we perform the scaling by $1/2$ required in the evaluation of the look-up table polynomial ϕ in advance within Equation (3). While Equation (3) itself still requires a multiplicative depth of two, the evaluation of ϕ on the scaled input can be carried out with a multiplicative depth of one by using the following form:

$$\phi(y) = 1 - 2y \cdot \left(y - \frac{1}{2}\right),$$

where $y = x/2$. Here, multiplication by an integer constant does not consume multiplicative depth, and therefore only a single depth-consuming multiplication remains in the evaluation of ϕ.

3.2 Analysis

To simplify the analysis of the key-switching complexity of Algorithm 1, we assume that all multiplications in the Merge procedure, except for Step 4, are performed without immediate relinearization. Under this assumption, the base case requires only a single key-switching operation, resulting in a total of $p/2$ key-switching operations.

In the subsequent combine phase, relinearization is applied to L_1, R_0, and R_1. Thus, each combine phase requires three key-switching operations. However, for $\log(p/2)$ of these phases, relinearization of R_1 is not required. As a result, the total number of key-switching operations used in the combine phase is $3 \cdot (p/2 - 1) - \log(p/2)$.

Finally, including the key-switching operations required for relinearization and the evaluation of ϕ at Step 12, Algorithm 1 requires a total of

$$2p - \log(p)$$

key-switching operations.

We now analyze the multiplicative depth of Algorithm 1. Algorithm 1 consumes a multiplicative depth of 2 in the base case, $\log(p/2)$ in the combine phase, and 1 for the evaluation of the look-up table polynomial ϕ. In total, this results in a multiplicative depth of $\log(p) + 2$.

Finally, the noise analysis of the comparison operation is provided in Appendix B.1,

3.2.1 Comparison with Prior Work. Recent non-interactive PDTE works [8, 16] report fast comparison primitives in the amortized setting. [8] proposes Batched_CW and Batched_RCC, while [16] proposes TECMP, RDCMP, and CDCMP. Among these, only RDCMP and CDCMP support ciphertext–ciphertext comparison, and they exhibit a latency–throughput trade-off; in particular, RDCMP is typically the most favorable for throughput.

We compare costs for processing the same number of queries N. Our comparator and RDCMP process N queries with a single packed evaluation, whereas CDCMP effectively operates on a smaller batch of size N/p and thus requires p evaluations to process N queries. Therefore, we normalize CDCMP's cost by multiplying its per-evaluation cost by p. Concretely, CDCMP requires $p \cdot (p(1+2\log p)+1)$ ciphertext multiplications and $p \cdot p(1+2\log p)$ rotations to process N queries, while RDCMP requires $3p - 2$ ciphertext multiplications. In contrast, our comparator performs ciphertext–ciphertext comparison using only $2p - \log p$ ciphertext multiplications, reducing the total operation count by more than one-third compared to RDCMP.

4 Optimized OBO Algorithm over CKKS

In this section, we describe the proposed LCPDTE protocol, a CKKS-based implementation of the OBO paradigm that requires only D comparison operations for a decision tree of depth D. As in existing

Algorithm 2 OboTraversal

Input: Ciphertexts of attribute vectors $\{\boldsymbol{x}_{i,j}\}$ with $\boldsymbol{x}_{i,j} \in \{0,1\}^{N/2}$ for $0 \le i < F,\ 0 \le j < p$, and a decision tree $(\mathcal{T}, \alpha, \tau, \Lambda)$.
Output: A ciphertext of the leaf value $\mathrm{ct}_{\mathrm{out}}$.

1: $\mathsf{OneHot}_0 \leftarrow \{1\}$
2: **for** $d = 0$ to $D-1$ **do**
3: $\quad \mathsf{OH}_d \leftarrow \mathsf{GenOH}(S_0, \ldots, S_{d-1})$
4: $\quad \hat{\boldsymbol{x}}_0, \ldots, \hat{\boldsymbol{x}}_{p-1} \leftarrow \{\mathsf{InnerProduct}(\boldsymbol{x}_j^*, \mathsf{OH}_d)\}_{0 \le j < p}$ where $\boldsymbol{x}_j = \{\boldsymbol{x}_{\alpha(i),j}\}_{i \in I_d}$.
5: $\quad \hat{\boldsymbol{t}}_0, \ldots, \hat{\boldsymbol{t}}_{p-1} \leftarrow \{\mathsf{InnerProduct}(\boldsymbol{t}_j^*, \mathsf{OH}_d)\}_{0 \le j < p}$ where $\boldsymbol{t}_j^* = \{\tau(i)_j\}_{i \in I_d}$.
6: $\quad S_d \leftarrow \mathsf{GEQ}(\{\hat{\boldsymbol{x}}_i\}_{0 \le i < p}, \{\hat{\boldsymbol{t}}_i\}_{0 \le i < p})$
7: $\quad S_d \leftarrow \mathsf{Btp}(S_d)$
8: **end for**
9: $\mathsf{OH}_D \leftarrow \mathsf{GenOH}(S_0, \ldots, S_{D-1})$
10: $\mathrm{ct}_{\mathrm{out}} \leftarrow \mathsf{InnerProduct}(\{\Lambda(i)\}_{i \in I_D}, \mathsf{OH}_D,)$
11: **return** $\mathrm{ct}_{\mathrm{out}}$

OBO-based approaches, the traversal method still incurs a complexity of 2^D, since it is inherently equivalent to the mul-path method. The main contribution of this paper, and the focus of this section, is to reduce the multiplicative depth in the traversal stage while simultaneously lowering the overall complexity to $O(p \cdot 2^{D/2})$. This reduction is particularly beneficial for RLWE-based schemes, as it decreases the number of bootstrapping invocations and allows for smaller cryptographic parameters, leading to a significant performance improvement. These optimizations are discussed in detail in Sections 4.2 and 4.3.

At the beginning of this section, we provide a high-level description of the overall algorithm and describe the encoding and packing methods performed by the client. In addition, Section 4.4 analyzes the efficiency and noise growth of the proposed algorithms; in particular, the noise analysis is combined with the comparison noise analysis presented in Section 3.2 to discuss the dominant noise terms of the overall protocol.

4.1 High-Level Description

Algorithm 2 provides a formal description of our protocol. The main subroutines of the algorithm are blind branch selection (Steps 3–5) and homomorphic comparison (Step 6). In the blind branch selection stage, the algorithm executes GenOH to homomorphically evaluate Eq. (1) and generate a one-hot vector for selecting the decision node to be visited at the current level. Assuming a complete binary tree without loss of generality, the procedure outputs 2^d ciphertexts, each encrypting one entry of the resulting one-hot vector.

Subsequently, the algorithm computes ciphertext–ciphertext inner products between the one-hot vector and the encrypted attribute values of all decision nodes, thereby retaining only the encrypted attribute value corresponding to the selected node. In a similar manner, plaintext–ciphertext inner products are performed with the threshold values associated with the decision nodes, which allows the encrypted threshold value of the selected node to be extracted. We denote by $\tau(i)_j$ the j-th bit of the threshold value associated with the i-th node.

We denote the inner product operation as

$$\mathsf{InnerProduct} : \sum_{0 \le i < 2^d} \mathsf{Mult}(\mathrm{ct}_i, \mathrm{ct}_i').$$

Ciphertext–ciphertext inner products are computed using the lazy relinearization technique and require at most a single key-switching operation. In contrast, inner products involving threshold values are evaluated as plaintext–ciphertext operations and therefore require no key switching.

Encoding & Packing. We describe the packing and encoding methods performed by the client. These methods are designed to maximize the amortized performance of the CKKS scheme. To this end, the client sets the number of attribute vectors to match the number of slots provided by CKKS. Accordingly, the decision tree evaluation can be performed simultaneously for up to $N/2$ inputs within a single ciphertext.

Let p denote the bit-length required to represent each attribute value by the decision tree model. To execute the homomorphic comparison procedure in the protocol, the server requires each attribute value to be decomposed into p bits.

In summary, let F denote the length of the attribute vector. At the beginning of the protocol, the client encodes, encrypts, and transmits $N/2$ p-bit attribute vectors

$$(\boldsymbol{x}_i)_{0 \le i < F} \in \mathbb{Z}^{N/2 \times F}$$

to the server as follows:

(1) Bit Decomposition: $\boldsymbol{x}_i \mapsto (\boldsymbol{x}_{i,0}, \ldots, \boldsymbol{x}_{i,p-1})$ for $0 \le i < F$.
(2) Encryption: $\mathsf{Enc}(\boldsymbol{x}_{i,j})$ for $0 \le i < F$ and $0 \le j < p$.
(3) Transmission: The client sends $\{\mathsf{Enc}(\boldsymbol{x}_{i,j}) : 0 \le i < F,\ 0 \le j < p\}$ to the server.

4.2 Blind Branch Selection

In this section, we introduce the one-hot vector generation algorithm, denoted by GenOH. We define the one-hot vector representing the path taken during tree traversal as follows:

$$\mathsf{OH}_d\big[b = (b_{d-1} \cdots b_0)_2; \{S_i\}_{0 \le i < d}\big] := \prod_{0 \le i < d} S_i^{b_i} \cdot (1 - S_i)^{1 - b_i},$$

where S_i denotes the decision-node value at the i-th tree level. For notational simplicity, we omit the explicit dependence on $\{S_i\}_{0 \le i < d}$ in the subsequent notation.

Strictly speaking, the above indexing is not consistent with the ordering of decision nodes at level d. We nevertheless adopt this convention for expository convenience: the traversal proceeds to the right child when $S_i = 1$, and the decison value at the root node is placed at S_d. The resulting index mismatch is corrected immediately before output by applying a bit-flip operation followed by a bit-reversal operation to align the indices with the node ordering.

We now describe the algorithm for constructing the one-hot vector. Each element of the one-hot vector can be expressed as a product of S_i and $(1 - S_i)$. The algorithm first computes the collection of all products over $\{S_i\}_{0 \le i < d}$, and then constructs the elements of the one-hot vector by applying linear combinations of the elements in this collection.

Algorithm 3 GenOH

Input: The product collection Prod_{d-1} from the previous level and the d-th decision value S_{d-1}.
Output: The updated product collection Prod_d and the encrypted one-hot vector OH_d.

$\mathsf{Prod}_d \leftarrow [0]^{2^d}$
for $i = 0$ to 2^d **do**
  if $i < 2^{d-1}$ **then**
    $\mathsf{Prod}_d[i] \leftarrow \mathsf{Prod}_{d-1}[i]$
  else
    $\mathsf{idx}_1, \mathsf{idx}_2 \leftarrow \mathsf{SplitHW}(i)$ (See App. 6)
    $\mathsf{Prod}_d[i] \leftarrow \mathsf{Mult}(\mathsf{Prod}_d[\mathsf{idx}_1], \mathsf{Prod}_d[\mathsf{idx}_2])$
  end if
end for
$\mathsf{OH}_d \leftarrow \mathsf{Prod}_d$
for $i = 0$ to d **do**
  $s \leftarrow 2^i$
  for $j = 0$ to $d/2s$ **do**
    for $k = 0$ to s **do**
      $\mathsf{OH}_d[2js + k] \leftarrow \mathsf{Sub}(\mathsf{OH}_d[2js + k], \mathsf{OH}_d[2js + k + s])$
    end for
  end for
end for
$\mathsf{OH}_d \leftarrow$ apply bit-flip and bit-reversal
return $\mathsf{Prod}_d, \mathsf{OH}_d$

We define the collection of all product combinations over $\{S_i\}_{0 \le i < d}$, denoted by Prod_d, as follows:

$$\mathsf{Prod}_d[b = (b_{n-1} \cdots b_0)_2] := \prod_{0 \le i < d} S_i^{b_i}.$$

where S_i denotes the decision-node value at the i-th tree level. For notational simplicity, we omit the explicit dependence on $\{S_i\}_{0 \le i < d}$ in the subsequent notation.

The elements of Prod_d can be paired by flipping only the k-th bit b_k for any $0 \le k < d$. For each such pair, subtracting the element with $b_k = 1$ from the element with $b_k = 0$ yields a product involving $(1 - S_k)$, as shown below:

$$\prod_{\substack{0 \le i < d \\ i \ne k}} S_i^{b_i} - S_k \cdot \prod_{\substack{0 \le i < d \\ i \ne k}} S_i^{b_i} = (1 - S_k) \cdot \prod_{\substack{0 \le i < d \\ i \ne k}} S_i^{b_i}.$$

By repeating this process for each $0 \le k < d$, we can compute all elements of the one-hot vector, and this process does not consume any multiplicative depth.

The remaining task is to compute Prod_d. The algorithm takes as input Prod_{d-1} from the previous level and the new decision value S_{d-1}. The first half of Prod_d (i.e., the elements with $b_{d-1} = 0$) is identical to Prod_{d-1} and therefore does not require additional computation. As a result, only the second half (i.e., the elements with $b_{d-1} = 1$) needs to be computed.

To compute an element $\mathsf{Prod}_d[b]$ in the second half, we split the index b into two indices according to its Hamming weight. Concretely, the first index is obtained by retaining the bits corresponding to the upper half of the Hamming weight and setting the remaining bits to zero, while the second index is obtained by retaining the bits corresponding to the lower half of the Hamming weight and setting the remaining bits to zero. For example, when $b = 11011_2$, splitting it into the upper three bits and the lower two bits according to the Hamming weight yields 11000_2 and 00011_2.

Algorithm 4 BsgsBBS

Input: Encryption of attribute vectors $\{\boldsymbol{x}_i\}_{i=0}^{N/2}$ where $\boldsymbol{x}_i \in \mathbb{Z}^F$ and tree $(\mathcal{T}, \alpha, \tau, \Lambda)$.
Output: 1 ciphertext $\mathsf{ct}_{\mathsf{out}}$

if $d \bmod 2 = 0$ **then**
  $\mathsf{Prod}_{d,E}, \mathsf{OH}_{d,E} \leftarrow \mathsf{GenOH}(\mathsf{Prod}_{d,E}, S_d)$
else
  $\mathsf{Prod}_{d,O}, \mathsf{OH}_{d,O} \leftarrow \mathsf{GenOH}(\mathsf{Prod}_{d,O}, S_d)$
end if
for $j = 0$ to p **do**
  for $c = 0$ to $2^{\lfloor d/2 \rfloor}$ **do**
    $\mathsf{bx}_c \leftarrow \mathsf{InnerProduct}(\{\boldsymbol{x}_{\alpha(i),j}\}_{i \in G_{d,c}}, \mathsf{OH}_{d,E})$
    $\mathsf{bt}_c \leftarrow \mathsf{InnerProduct}(\{\tau(i)_j\}_{i \in G_{d,c}}, \mathsf{OH}_{d,E})$
  end for
  $\hat{\boldsymbol{x}}_j \leftarrow \mathsf{InnerProduct}(\{\mathsf{bx}_c\}_{0 \le c < 2^{\lfloor d/2 \rfloor}}, \mathsf{OH}_{d,O})$
  $\hat{\boldsymbol{t}}_j \leftarrow \mathsf{InnerProduct}(\{\mathsf{bt}_c\}_{0 \le c < 2^{\lfloor d/2 \rfloor}}, \mathsf{OH}_{d,O})$
end for
return $\{\hat{\boldsymbol{x}}_i\}_{0 \le i < p}, \{\hat{\boldsymbol{t}}_i\}_{0 \le i < p}$

4.3 Baby-step Giant-step Branch Selection

We now introduce our BSGS-based branch selection algorithm, denoted by BsgsBSS, which reduces the traversal complexity to $O(p \cdot \sqrt{2^D})$. The key intuition behind this algorithm is that each entry of the one-hot vector is a product of factors of the form S_i or $1 - S_i$, and that these entries are ultimately used in inner product operations with attribute values and thresholds.

Based on this observation, we optimize the algorithm using the baby-step giant-step paradigm, and a detailed description of the resulting procedure is provided in Algorithm 4. More precisely, we generate a one-hot vector of length $2^{\lfloor d/2 \rfloor + 1}$ from the even-indexed decision values $\{S_{2i}\}$, denoted by $\mathsf{OH}_{d,E}$, and another one-hot vector of length $2^{\lfloor d/2 \rfloor}$ from the odd-indexed decision values $\{S_{2i+1}\}$, denoted by $\mathsf{OH}_{d,O}$. We then partition the decision nodes at depth d into $2^{\lfloor d/2 \rfloor}$ index sets as follows:

$$G_{d,c} := \left\{ (b_{\lfloor d/2 \rfloor} c_{\lfloor d/2 \rfloor - 1} \cdots b_0 c_0)_2 \;\middle|\; 0 \le b < 2^{\lfloor d/2 \rfloor + 1} \right\},$$

for $0 \le c < 2^{\lfloor d/2 \rfloor}$.

Subsequently, for each index set, we compute an inner product with $\mathsf{OH}_{d,E}$ to obtain $2^{\lfloor d/2 \rfloor}$ intermediate results, and then take an inner product between these intermediate results and $\mathsf{OH}_{d,O}$ to produce the final output.

Correctness. To establish the correctness of the algorithm, it suffices to show that, for any vector

$$\boldsymbol{x} = (x_i)_{0 \le i < 2^d},$$

and for the partition induced by the groups $\{G_{d,c}\}$, where

$$\boldsymbol{x}_c := (x_i)_{i \in G_{d,c}}, \quad 0 \le c < 2^{\lfloor d/2 \rfloor},$$

the following equality holds:

$$\langle \boldsymbol{x}, \mathsf{OH}_d \rangle = \sum_{0 \le c < 2^{\lfloor d/2 \rfloor}} \langle \boldsymbol{x}_c, \mathsf{OH}_{d,E} \rangle \cdot \mathsf{OH}_{d,O}[c].$$

This identity follows directly from the fact that each entry of the one-hot vector OH_d is uniquely determined by the binary representation of its index. In particular, for any index belonging to the c-th group $G_{d,c}$, the corresponding one-hot entry admits a common multiplicative factor that depends only on c.

More precisely, for any

$$(b_k)_{0 \le k \le \lfloor d/2 \rfloor} \in \{0,1\}^{\lfloor d/2 \rfloor + 1},$$

the following holds:

$$\mathsf{OH}_d\big[(b_{\lfloor d/2 \rfloor} c_{\lfloor d/2 \rfloor - 1} \cdots b_0 c_0)_2\big] = \prod_{0 \le i \le \lfloor d/2 \rfloor} S_{2i}^{b_i} (1 - S_{2i})^{1-b_i} \cdot \left(\prod_{0 \le j < \lfloor d/2 \rfloor} S_{2j+1}^{c_j} (1 - S_{2j+1})^{1-c_j} \right).$$

In the above expression, the second product depends solely on c and therefore acts as a common factor for all indices within the same group $G_{d,c}$. As a result, the global inner product $\langle \boldsymbol{x}, \mathsf{OH}_d \rangle$ can be decomposed into a sum of group-wise inner products with $\mathsf{OH}_{d,E}$, followed by a weighted aggregation using $\mathsf{OH}_{d,O}$.

4.4 Analysis

We first analyze the key-switching complexity of GenOH (Algorithm 3) and BsgsBBS (Algorithm 4).

Suppose that blind branch selection is performed for the nodes at depth d. BsgsBBS invokes the GenOH procedure to generate an updated one-hot vector that incorporates the newly added decision value S_d. Since GenOH requires key switching only at Step 7, this step incurs $2^{d/2}$ key-switching operations.

Next, in Steps 7–12 of BsgsBBS, inner product operations are performed. Due to the use of the lazy relinearization technique, each ciphertext–ciphertext inner product requires only a single key-switching operation. Therefore, this stage incurs $p \cdot 2^{d/2} + 2p$ key-switching operations.

Consequently, the number of key-switching operations required at depth d is bounded by $O(p \cdot 2^{d/2})$. Summing over all depths up to a decision tree of depth D, the overall key-switching complexity of the protocol is

$$O(p \cdot 2^{D/2}).$$

We now discuss the multiplicative depth consumed by the algorithm. Recall that in the GenOH procedure, the selection of factors used to construct Prod_d is performed by partitioning indices according to their Hamming weights. The number of ones in an index of Prod_d corresponds to the number of decision values involved in the product represented by that entry.

Partitioning the indices by halving the Hamming weight induces a multiplication structure analogous to a butterfly network; as a result, the construction of Prod_d requires at most $\log d$ multiplicative depth. On the other hand, the BsgsBBS procedure generates $d/2$ product combinations, which requires a multiplicative depth of $\log(d/2)$. Together with two subsequent inner product operations, the blind branch selection stage requires a multiplicative depth of $\log d + 1$.

Algorithm 5 LevelMajorEvaluation

Input: Ciphertexts of attribute vectors $\{\boldsymbol{x}_{i,j}\}$ with $\boldsymbol{x}_{i,j} \in \{0,1\}^{N/2}$ for $0 \le i < F$ and $0 \le j < p$, and K decision trees $\{(\mathcal{T}^{(k)}, \alpha^{(k)}, \tau^{(k)}, \Lambda^{(k)})\}_{0 \le k < K}$.
Output: K ciphertexts of the leaf values $\mathsf{ct}_{\mathsf{out}}^{(k)}$.

1: **for** $d = 0$ to $D-1$ **do**
2: **for** $k = 0$ to K **do**
3: $\{\hat{\boldsymbol{x}}_i^{(k)}\}_{0 \le i < p}, \{\hat{\boldsymbol{t}}_i^{(k)}\}_{0 \le i < p} \leftarrow \mathsf{BsgsBBS}(S_0^{(k)}, \ldots, S_{d-1}^{(k)})$
4: $S_d^{(k)} \leftarrow \mathsf{GEQ}(\{\hat{\boldsymbol{x}}_i^{(k)}\}_{0 \le i < p}, \{\hat{\boldsymbol{t}}_i^{(k)}\}_{0 \le i < p})$
5: **end for**
6: $S_d^{(0)}, \ldots, S_d^{(K-1)} \leftarrow \mathsf{BatchBtp}(S_d)$
7: **end for**
8: **for** $k = 0$ to K **do**
9: $\mathsf{OH}_D \leftarrow \mathsf{GenOH}(S_0^{(k)}, \ldots, S_{D-1}^{(k)})$
10: $\mathsf{ct}_{\mathsf{out}}^{(k)} \leftarrow \mathsf{InnerProduct}(\{\Lambda^{(k)}(i)\}_{i \in I_D}, \mathsf{OH}_D^{(k)})$
11: **end for**
12: **return** $\{\mathsf{ct}_{\mathsf{out}}^{(k)}\}$

Finally, the noise analysis of the traversal algorithm is presented in Appendix B.2.

5 Level-Major Tree Evaluation

In this section, we describe an evaluation strategy that mitigates the overhead arising from evaluating multiple trees in GBDT models.

The tree evaluation algorithm introduced earlier is designed for evaluating a single decision tree. However, a GBDT model requires evaluating multiple trees in parallel and aggregating their outputs. As a result, when extending the PDTE protocol from a single-tree setting to GBDT, the overall runtime increases linearly with respect to the number of trees.

To alleviate this overhead, we propose a *level-major evaluation* strategy. The parallel tree evaluation procedure based on this strategy is described in Algorithm 5. The key idea is to exploit the fact that the encrypted data takes values in $\{0,1\}$, and to evaluate multiple trees in a level-wise manner rather than independently. This allows us to significantly reduce the number of bootstrapping operations.

More concretely, after traversing K trees level by level, we update the decision values using the batch bit bootstrapping technique proposed in [4], which enables bootstrapping K decision bits simultaneously. According to the analysis in [4], performing batch bit bootstrapping on K bit ciphertexts requires approximately one bootstrapping and $O(K \cdot 2^{K/2})$ key-switching operations. When K is chosen appropriately, the additional key-switching cost remains negligible compared to the overall bootstrapping complexity, and the performance of batch bit bootstrapping is therefore comparable to that of a single bootstrapping operation.

Consequently, when evaluating K decision trees of depth D, the proposed level-major evaluation strategy reduces the total number of bootstrapping operations from $K \cdot D$ to D.

6 Experiments

We ran all experiments on a single workstation with an AMD Ryzen 9 7900X CPU and 128 GB of RAM. Table 4 lists the CKKS parameters

$\log N$	$\log QP$	$(h, \tilde{h})$
16	1328–1508	(192,32)

Base	StC	Circuit	LUT	EvalExp	CtS	P
45	39×3	$45 \times 8 + 45 \times \log(D)$	$45 \times (K+1)$	45×8	42×3	46×5

Table 4: The CKKS scheme parameter set used for the experiments. Here, D denotes the depth of the decision tree, and K denotes the number of trees referenced in Algorithm 5.

Method	D=2	D=5	D=8	D=12
Ours	1.1209	3.9627	**8.9725**	**42.2055**
BPDTE-CW	0.1757	1.9633	16.9559	326.5083
BPDTE-RCC	**0.1583**	**1.7586**	15.3342	439.1848
RDCMP-ASM	0.6301	6.8249	56.3396	905.1788
RDCMP-ESM	1.2249	12.4789	101.6719	

Table 5: Amortized runtime (ms) at representative depths. RDCMP-ESM fails to run at D=12 under the same setting; we leave the entry blank.

used throughout the paper. We implemented our protocol and all baselines using the Lattigo library [1]. For the baselines from [8, 16], we evaluated all four variants at $\log N = 15$ and used the smallest ring modulus that guarantees decryption correctness while matching the batch size of our CKKS setting and using the same CKKS parameterization. We use complex packing, as CKKS natively supports complex numbers; details are provided in Appendix C.

Except for the real-model validation in Section 6.4, we use randomly generated perfect binary trees of depth D, each with 2^D nodes. We fix the attribute vector length to 1 in all experiments. The attribute dimension does not affect the model runtime, while the input query size grows linearly with the attribute dimension. Thus, all reported input query sizes scale linearly with the attribute vector length.

6.1 Threat Model

We assume the standard semi-honest threat model used in prior non-interactive PDTE works: the client holds the private input, the server holds the model, and the server follows the protocol but may try to infer information from the transcript. The client sends encrypted inputs and receives an encrypted output under CKKS with 128-bit security parameters.

The protocol reveals only the agreed public input format for deployment (feature count and ordering) and the final output, but no additional model-dependent information during evaluation.

6.2 Baseline comparison

Figure 1 presents a comparison with prior baselines. The left plot reports the amortized runtime, while the right plot shows the amortized query size. In addition, we summarize the amortized runtimes at representative tree depths in Table 5. The baselines are taken from [8, 16], which represent the most recent non-interactive PDTE protocols and achieve the best amortized performance among prior work. Specifically, BPDTE_CW and BPDTE_RCC are methods proposed in [8], where CW and RCC denote the two comparison algorithms introduced by the authors. Similarly, RDCMP_ASM and RDCMP_ESM are methods proposed in [16], where ASM and ESM correspond to two traversal algorithms introduced in that work. For [16], we restrict our comparison to RDCMP, as it achieves both the lowest amortized runtime and the smallest amortized query size among the three comparison functions proposed in the paper (TECMP, CDCMP, and RDCMP).

For runtime, while the other protocols exhibit an almost linear trend on the log-scale as D increases, our method maintains a smaller slope. Although our method is not the fastest at small depths, it becomes faster than all baselines from $D = 7$. At $D = 12$, our method is 7.74× faster than BPDTE_CW, 10.4× faster than BPDTE_RCC, and 21.45× faster than RDCMP_ASM. For RDCMP_ESM, the evaluation is feasible only up to $D = 11$ due to the 128 GB memory limit, and within this range our method achieves up to a 32.67× speedup.

For query size, we report the sum of the input query sent by the client and the output query returned after evaluation, and plot it on a linear scale. BPDTE_CW and BPDTE_RCC achieve the best runtime at small depths and are consistently faster than the RDCMP-based protocols. This behavior stems from reducing the complexity of comparisons via decomposition, at the cost of increased input-query size, leading to an explicit trade-off between communication and runtime. Concretely, compared to RDCMP_ASM (4.13 KB), BPDTE_CW and BPDTE_RCC incur substantially larger query sizes of 71.26 KB and 882.8 KB, respectively. RDCMP_ESM uses SumPath-style traversal, which causes the output query size to grow exponentially with D. While its traversal computation is small, it yields the worst runtime in our experiments due to the heavier parameterization required by the additional levels consumed for shuffling. Our method uses an input query of 8.5 KB and transmits the same number of ciphertexts as RDCMP_ASM; however, due to differences in scheme complexity, it incurs a slightly higher memory footprint.

Overall, our method achieves substantial improvements in both runtime and communication compared to the BPDTE variants and RDCMP_ESM. Compared to RDCMP_ASM, our method increases communication by 4.37 KB, while providing a 21.45× speedup.

6.3 Performance Breakdown

Figure 2 presents an end-to-end performance breakdown of our protocol. We observe that the bootstrapping and comparison components grow approximately linearly with the depth, while the traversal component grows much more rapidly. This behavior is consistent with the $O(p \cdot \sqrt{2^D})$ complexity of our traversal procedure. At $D = 12$, traversal accounts for 81.7% of the total runtime.

6.4 Ablation / Additional experiments

Ablation Study for OBO paradigm. Figure 3 reports an ablation study for the OBO paradigm. Since OBO is a core component of our overall framework and is tightly coupled with our traversal procedure, it is not straightforward to disable OBO in isolation. Instead, we compare the end-to-end runtime of our full framework against the comparison-time of an all-node strategy that evaluates

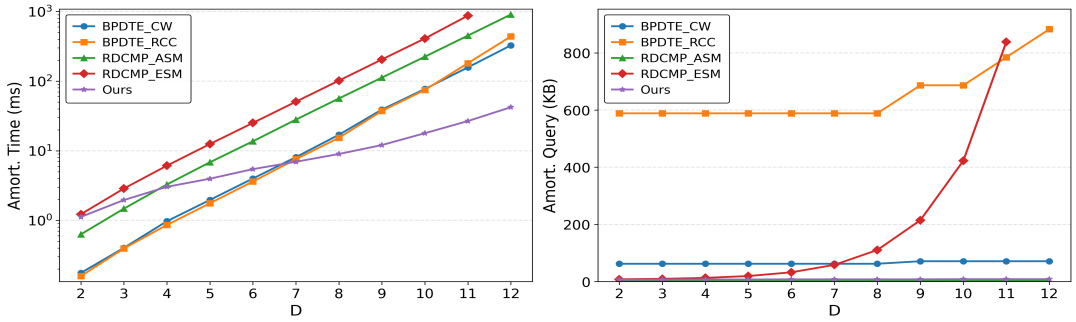

Figure 1: A baseline comparison against four recent non-interactive PDTE variants, where the left plot shows amortized runtime versus depth D (log scale) and the right plot shows amortized query size (input+output; linear scale); RDCMP_ESM is infeasible beyond $D = 11$ due to the 128 GB memory limit.

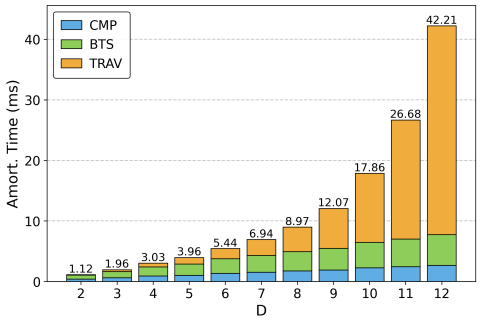

Figure 2: End-to-end performance breakdown of our protocol versus depth D (linear scale). We report the time spent in comparison (CMP), bootstrapping (BTS), and traversal (TRAV).

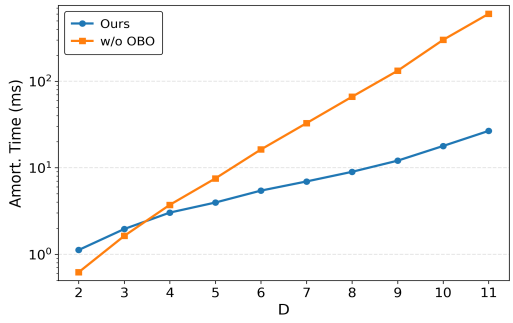

Figure 3: An ablation study of the OBO paradigm on a log scale, comparing the end-to-end runtime of our framework with the comparison time of an all-node $O(2^D)$ strategy.

comparisons for all internal nodes, incurring $O(2^D)$ comparisons. The all-node comparison cost exceeds the end-to-end runtime of our full framework starting from $D = 4$, and becomes infeasible at $D = 12$ due to the 128 GB memory limit. At $D = 11$, the all-node comparison cost is 22.54× larger. Notably, this gap holds even though we compare only the comparison-time of the all-node strategy against the full end-to-end runtime of our framework.

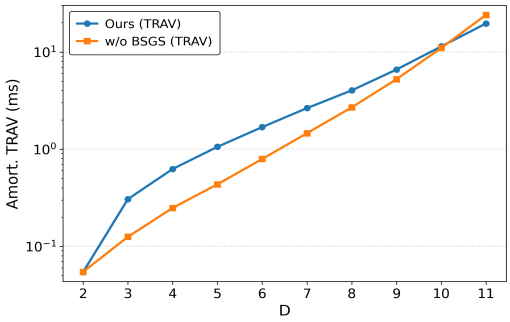

Figure 4: An ablation study of BSGS-based branch selection on a log scale, comparing the end-to-end runtime of our method (with GenOH fixed) against a naive $O(2^D)$ traversal and highlighting the resulting $O(p\sqrt{2^D})$ scaling.

Ablation Study for BSGS traversal. Figure 4 presents an ablation study of BSGS-based branch selection. With the GenOH procedure (Algorithm 3) in place, we compare the runtime of branch selection before applying BSGS, which has $O(2^D)$ complexity, against the runtime after applying BSGS, which has $O(p \cdot \sqrt{2^D})$ complexity.

At small depths, the $O(2^D)$ approach yields lower runtime. This is because we use $p = 32$, and the resulting bit-length overhead dominates at small D. Starting from $D = 11$, however, the crossover occurs and the BSGS-based algorithm becomes more efficient. Moreover, due to the difference in growth rates, the gap is expected to widen as D increases. From $D = 12$, without BSGS, traversal becomes infeasible because storing a length-2^D one-hot vector exceeds our 128 GB memory limit.

Batched bootstrapping. Figure 4 shows the effect of applying batched bootstrapping. The results presented so far evaluate our method in the single-tree setting. When we extend the protocol to GBDT and adopt a level-major evaluation order, we can apply batched bootstrapping (see Algorithm 5). In this case, the bootstrapping time is reduced by about 3.6×, independent of D. However, as shown in Figure 2, since the bootstrapping time grows linearly with D, its fraction of the end-to-end runtime decreases as D increases.

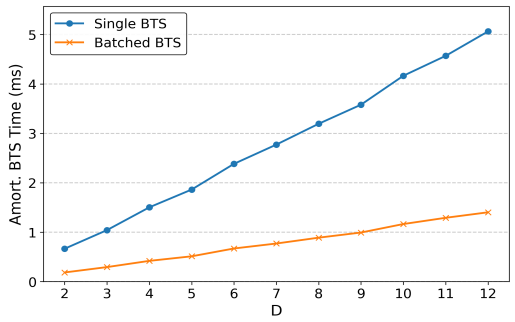

Figure 5: The effect of batched bootstrapping on GBDT evaluation.

(D, K)	Time(ms)	MaxNoise (bits)	MatchRate
$(8, 8)$	48.38	−20.09	100.00%
$(10, 8)$	79.99	−20.22	99.95%
$(12, 8)$	162.684	−17.47	100.00%

Table 6: End-to-end amortized evaluation time on $N = 32768$ test queries, where (D, K) denotes tree depth and the number of trees. We report amortized time per query, the maximum noise as $\log_2(\text{MaxAbs})$ (in bits), and MatchRate in the margin domain.

Consequently, the overall speedup from batched bootstrapping diminishes from a 1.57× reduction in total runtime at $D = 2$ to a 1.10× reduction at $D = 12$.

Real-Model Validation. Table 6 reports an end-to-end validation on XGBoost models trained on the Credit Card Fraud dataset.[2] This dataset is widely used in the fraud-detection literature [9, 10]. We evaluate ensembles with $K = 8$ trees at depths $D \in \{8, 10, 12\}$ on $N = 32768$ test queries. The table reports the amortized evaluation time per query, MaxNoise in bits (reported on a log scale), and the logit-domain MatchRate between plaintext and encrypted evaluation. Across all settings, our full end-to-end pipeline—including level-major GBDT evaluation and batched-bit bootstrapping—exhibits low noise and near-perfect agreement (MatchRate ≥ 99.95%), providing an end-to-end sanity check of the proposed framework on trained models.

7 Discussion

7.1 Memory Efficiency

Our reduced complexity yields benefits beyond runtime. In prior HE-based non-interactive PDTE protocols with $O(2^D)$-type evaluation, the peak number of ciphertexts materialized during evaluation can also grow as $O(2^D)$, leading to prohibitive memory footprints at larger depths. In our ablation study, both (i) executing $O(2^D)$ comparisons and (ii) performing an $O(2^D)$-cost traversal exceeded 128 GB of memory at $D = 12$. Moreover, even though BFV ciphertexts are typically smaller than CKKS ciphertexts, the BFV-based RDCMP_ESM baseline also exceeded 128 GB at $D = 12$, since its shuffling procedure introduces an additional constant-factor ciphertext overhead on top of the $O(2^D)$ growth.

These results indicate that the limitations of prior approaches at larger D are not confined to latency: memory becomes a hard bottleneck as well. In contrast, our method keeps the peak number of ciphertexts to $O(\sqrt{2^D})$, enabling evaluation at substantially larger depths within a fixed memory budget. Empirically, at $D = 12$ our method uses 33.97GB of memory, while the $O(2^D)$ variants exceed the 128 GB budget.

7.2 Bit Casting vs. Quantization

Most machine-learning models operate under `float32` precision. However, existing HE-based PDTE baselines typically quantize inputs by truncating them to an ad-hoc low-bit integer format to reduce evaluation cost. Such quantization lies in a large uncertainty space: it introduces approximation error and can have unintended effects on model fidelity, which is why it is generally considered a careful design choice in machine learning.

We instead use bit casting. Rather than reducing precision, bit casting reinterprets each `float32` input in a same-bit integer space (with $p = 32$), preserving the exact information required by our 32-bit comparison procedure. Although this representation is not intended for general real-valued arithmetic, PDTE for tree models does not require such arithmetic: once an attribute and a split threshold are provided, inference proceeds only via comparisons. Accordingly, we perform comparisons via bit casting and keep $p = 32$ throughout, enabling deterministic and accurate comparisons without relying on quantization.

8 Conclusion

In this paper, we presented LCPDTE, a CKKS-based non-interactive protocol for private decision-tree evaluation (PDTE). By combining the One-Branch-Only (OBO) paradigm with our Baby-Step Giant-Step (BSGS)-based branch selection, LCPDTE reduces the dominant evaluation cost from $O(2^D)$ to $O(p \cdot \sqrt{2^D})$, with a memory footprint of $O(\sqrt{2^D})$. In our experiments, LCPDTE achieves up to a 21.45× speedup at depth $D = 12$. While the dependence on tree depth remains exponential, this square-root asymptotic improvement substantially reduces the computational and memory burden at larger depths. Overall, LCPDTE improves scalability without compromising precision, strengthening the practicality of HE-based inference and serving as a building block toward large-scale encrypted deployment, including ensemble models such as GBDT.

Acknowledgments

This work was supported by the Institute of Information Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00399401, Development of post-quantum security infrastructure transition and comprehensive quantum security verification technology, 50% & No.RS-2024-00398360, Development of a user-friendly and efficiency-optimized real-time homomorphic statistical analysis processing platform, 50%).

[2] https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

References

[1] 2024. Lattigo v6. Online. https://github.com/tuneinsight/lattigo EPFL-LDS, Tune Insight SA.

[2] Rasoul Akhavan Mahdavi, Haoyan Ni, Dimitry Linkov, and Florian Kerschbaum. 2023. 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*. 2945–2958. doi:10.1145/3576915.3623095

[3] Sofiane Azogagh, Victor Delfour, Sébastien Gambs, and Marc-Olivier Killijian. 2022. PROBONITE: PRivate One-Branch-Only Non-Interactive decision Tree Evaluation. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 23–33. doi:10.1145/3560827.3563377

[4] Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. 2024. Bootstrapping Small Integers With CKKS. In *Advances in Cryptology – ASIACRYPT 2024*. 330–360. doi:10.1007/978-981-96-0875-1_11

[5] Raphaël Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine Learning Classification over Encrypted Data. In *Network and Distributed System Security Symposium (NDSS) 2015*. Internet Society. doi:10.14722/ndss.2015.23241

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. doi:10.1145/2939672.2939785

[7] Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira. 2022. SortingHat: Efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 563–577. doi:10.1145/3548606.3560702

[8] Kelong Cong, Jiayi Kang, Georgio Nicolas, and Jeongeun Park. 2024. Faster Private Decision Tree Evaluation for Batched Input from Homomorphic Encryption. In *Security and Cryptography for Networks*. Springer-Verlag, 3–23. doi:10.1007/978-3-031-71073-5_1

[9] Andrea Dal Pozzolo, Giacomo Boracchi, Olivier Caelen, Cesare Alippi, and Gianluca Bontempi. 2017. Credit Card Fraud Detection: A Realistic Modeling and a Novel Learning Strategy. *IEEE Transactions on Neural Networks and Learning Systems* 29, 8 (2017), 3784–3797. doi:10.1109/TNNLS.2017.2736643

[10] Andrea Dal Pozzolo, Olivier Caelen, Yann-Aël Le Borgne, Serge Waterschoot, and Gianluca Bontempi. 2014. Learned lessons in credit card fraud detection from a practitioner perspective. *Expert Systems with Applications* 41, 10 (2014), 4915–4928. doi:10.1016/j.eswa.2014.02.026

[11] Jordan Frery, Andrei Stoian, Roman Bredehoft, Luis Montero, Celia Kherfallah, Benoit Chevallier-Mames, and Arthur Meyre. 2023. Privacy-Preserving Tree-Based Inference with TFHE. In *Mobile, Secure, and Programmable Networking (MSPN 2023), Revised Selected Papers*. Springer-Verlag, 139–156. doi:10.1007/978-3-031-52426-4_10

[12] Jerome H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics* 29, 5 (2001), 1189–1232. doi:10.1214/aos/1013203451

[13] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 3149–3157.

[14] Ágnes Kiss, Masoud Naderpour, Jian Liu, N. Asokan, and Thomas Schneider. 2019. SoK: Modular and Efficient Private Decision Tree Evaluation. *Proceedings on Privacy Enhancing Technologies* 2019, 2 (2019), 187–208. doi:10.2478/popets-2019-0026

[15] Ronny Ko, Rasoul Akhavan Mahdavi, Byoungwoo Yoon, Makoto Onizuka, and Florian Kerschbaum. 2024. SilentWood: Private Inference Over Gradient-Boosting Decision Forests. arXiv preprint arXiv:2411.15494. arXiv:2411.15494 [cs.CR] doi:10.48550/arXiv.2411.15494

[16] Huiqiang Liang, Haining Lu, Yifeng Guo, Geng Wang, Haining Yu, Hongli Zhang, Baoyu An, Jinyu Li, and Li Su. 2024. BPDTE: Batch Private Decision Tree Evaluation via Amortized Efficient Private Comparison. Cryptology ePrint Archive, Paper 2024/619. https://eprint.iacr.org/2024/619

[17] Wen-jie Lu, Jun-jie Zhou, and Jun Sakuma. 2018. Non-interactive and Output Expressive Private Comparison from Homomorphic Encryption. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 67–74. doi:10.1145/3196494.3196503

[18] Jestine Paul, Benjamin Hong Meng Tan, Bharadwaj Veeravalli, and Khin Mi Mi Aung. 2022. Non-Interactive Decision Trees and Applications with Multi-Bit TFHE. *Algorithms* 15, 9 (2022), 333. doi:10.3390/a15090333

[19] Raymond K. H. Tai, Jack P. K. Ma, Yongjun Zhao, and Sherman S. M. Chow. 2017. Privacy-Preserving Decision Trees Evaluation via Linear Functions. In *Computer Security – ESORICS 2017*. Springer International Publishing, Cham, 494–512. doi:10.1007/978-3-319-66399-9_27

[20] Anselme Tueno, Yordan Boev, and Florian Kerschbaum. 2020. Non-interactive Private Decision Tree Evaluation. In *Data and Applications Security and Privacy XXXIV*. Springer International Publishing, 174–194. doi:10.1007/978-3-030-49669-2_10

[21] David J. Wu, Tony Feng, Michael Naehrig, and Kristin Lauter. 2015. Privately Evaluating Decision Trees and Random Forests. Cryptology ePrint Archive, Paper 2015/386. https://eprint.iacr.org/2015/386

[22] David J. Wu, Tony Feng, Michael Naehrig, and Kristin Lauter. 2016. Privately Evaluating Decision Trees and Random Forests. *Proceedings on Privacy Enhancing Technologies* 2016, 4 (2016), 335–355. doi:10.1515/popets-2016-0043

[23] Liang Xue, Dongxiao Liu, Cheng Huang, Xiaodong Lin, and Xuemin (Sherman) Shen. 2020. Secure and Privacy-Preserving Decision Tree Classification with Lower Complexity. *Journal of Communications and Information Networks* 5, 1 (2020), 16–25. doi:10.23919/JCIN.2020.9055107

A The CKKS scheme

CKKS is a ring LWE-based HE scheme that approximates complex vector in $\mathbb{C}^{N/2}$. The plaintext space of CKKS is R. Mapping complex vectors to elements of the R is called encoding, as described below.

Encoding. Let $\zeta_j = \exp(2\pi i \cdot 5^j/2N)$ for $0 \le j < N/2$. The map $m(X) \to (m(\zeta_j))_{0\le j<N/2} \in \mathbb{C}^{N/2}$ is an isomorphism between $\mathbb{R}[X]/(X^N+1)$ and $\mathbb{C}^{N/2}$. It is also an isometric transformation that behaves like the discrete Fourier transform. Therefore, for convenience, we will refer to this mapping as DFT. Similarly, we can define the inverse transform of DFT as DFT^{-1}. Given a $\Delta > 0$, the encoding/decoding algorithm of CKKS is defined as follows.

$$\mathsf{Ecd} : \mathbb{C}^{N/2} \to R : \mathbf{z} \to \left\lfloor \Delta \cdot \mathsf{DFT}^{-1}(\mathbf{z}) \right\rceil \in R.$$

$$\mathsf{Dcd} : R \to \mathbb{C}^{N/2} : m(X) \to \frac{1}{\Delta}\mathsf{DFT}(m(X)) \in \mathbb{C}^{N/2}.$$

Encryption. Given a plaintext $m \in R$, a ciphertext of m is a pair $\mathsf{ct} = (b, a) \in R_Q^2$ such that $(b, a) \cdot (1, s) = m + e \mod Q$ for a secret key $s \in R$ and a small error $e \in R$. Unlike other HE schemes, CKKS does not separate errors from plaintext, but manages the size of the error. Because of this difference, the CKKS scheme is called an approximate arithmetic scheme. The decryption algorithm is defined as

$$\mathsf{Dec}(\mathsf{ct}) = (b, a) \cdot (1, s) \in R_Q^2.$$

Homomorphic Operations. Given CKKS ciphertext $\mathsf{ct}_1 = (b_1, a_1), \mathsf{ct}_2 = (b_2, a_2) \in R_Q^2$ and CKKS plaintext $m(X)$, the addition algorithm is defined as

$$\mathsf{ct}_1 + \mathsf{ct}_2 = (b_1 + b_2,\ a_1 + a_2) \in R_Q^2.$$

Multiplication is not as simple as addition. First, the tensor product of two ciphertexts is defined as

$$\mathsf{Tensor}(\mathsf{ct}_1, \mathsf{ct}_2) = (d_0, d_1, d_2) = (b_1 b_2,\ b_1 a_2 + b_2 a_1,\ a_1 a_2) \in R_Q^3.$$

The output of the tensor product increases the size of the ciphertext and requires $(1, s, s^2)$ as the decryption key instead of the existing decryption key $(1, s)$. The process of reducing the size of the ciphertext using a key switching procedure that moves the secret s^2 to s is called relinearization. Relinearization is defined as

$$\mathsf{Relin}(d_0, d_1, d_2) = (d_0, d_1) + \mathsf{KeySwitch}_{s^2 \to s}(d_2)$$

Since two ciphertexts with scaling factor Δ are multiplied, the scale of the ciphertext must be reduced to Δ. This procedure is called rescaling and is performed by approximate division. Given a ciphertext $\mathsf{ct} = (b, a) \in R_Q^2$ and $q|Q$, the rescaling algorithm for q is defined as

$$\mathsf{Rescale}(\mathsf{ct}) = (\lfloor b/q \rceil, \lfloor a/q \rceil) \in R_{Q/q}^2.$$

Finally, CKKS multiplication is performed as follows

$$\mathsf{Mult}(\mathsf{ct}_1, \mathsf{ct}_2) = \mathsf{Rescale} \circ \mathsf{Relin} \circ \mathsf{Tensor}(\mathsf{ct}_1, \mathsf{ct}_2)$$

We are interested in how much homomorphic multiplication increases the error. Let the plaintexts of the ciphertexts $\mathrm{ct}_1, \mathrm{ct}_2$ be $\Delta m_1 + e_1, \Delta m_2 + e_2$, respectively. The result message of Mult is as follows

$$\Delta m_1 m_2 + (m_1 e_2 + m_2 e_1) + e$$

where e is key switching and rescaling error. This means that if the message size is too large, the multiplication error will increase.

In CKKS, in addition to addition and multiplication, there is a *conjugation* operation that conjugate the slot vector. The conjugation is defined as follows

$$\mathsf{Conj}(\mathsf{ct}) = (b(X^{5^{-1}}), 0) + \mathsf{KeySwitch}_{s(X^{5^{-1}}) \to s}(a(X^{5^{-1}}))$$

Bootstrapping. CKKS multiplication consumes modulus, so if there are insufficient modulus resources, multiplication can no longer be performed. The procedure by which CKKS recovers modulus resources while maintaining message precision is called bootstrapping. We will treat the conventional bootstrapping algorithm as a black box and discuss discrete CKKS bootstrapping in detail in next section.

A.1 Discrete CKKS

Discrete CKKS is a kind of modification that breaks away from the conventional concept of handling complex vectors and instead handles discrete data such as integers. Discrete CKKS differs from BFV/BGV and DM/CGGI, which were considered standard for existing integer data types. First, CKKS handles integers as a subspace of the complex number field rather than a finite field. This difference means that, unlike BFV/BGV, which has native reduction, there is no overflow phenomenon, and both real and integer data types can be handled simultaneously. The second difference comes from the CKKS philosophy of not separating errors. Since CKKS does not have a native algorithm to reduce errors, errors must be reduced as needed when processing discrete data. This process is called error cleaning.

Discrete Bootstrapping. State-of-the-art discrete bootstrapping techniques support functional bootstrapping on integers while simultaneously performing error cleaning. This paper addresses the simplest form of discrete bootstrapping. In what follows, we briefly describe each subroutine of the approach used in this work.

(1) **Slots-to-Coefficients** (StC) : Given a ciphertext $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{z}) \in R_q^2$ encrypting a message $\boldsymbol{z} \in \mathbb{Z}_t$, the output ciphertext $\mathsf{ct}' = \mathsf{Enc}(q_0/t \cdot z(X)) \in R_{q_0}^2$ where $z(X)$ is a polynomial in R whose coefficients corresponds to the entries of $\boldsymbol{z}$. This means moving the slot to the coefficient.
(2) **Modulus Raising** (ModRaise) : Given a ciphertext $\mathsf{ct} = \mathsf{Enc}(q_0/t \cdot z(X)) \in R_{q_0}^2$, the output ciphertext $\mathsf{ct}' = \mathsf{Enc}(q_0/t \cdot z(X) + q_0 \cdot I(X)) \in R_Q^2$ where $I(X)$ is a polynomial in R is a small integer polynoial and $Q > q_0$.
(3) **Coefficients-to-Slots** (CtS) : Given a ciphertext $\mathsf{ct} = \mathsf{Enc}(q_0/t \cdot z(X) + q_0 \cdot I(X)) = \mathsf{Enc}\big(q_0/t \cdot (z(X) + t \cdot I(X))\big) \in R_Q^2$, the output ciphertext $\mathsf{ct}' = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{z} + t \cdot \boldsymbol{I})$. This means moving the coefficient to the slot.
(4) **Homomorphic Exponentiation** (EvalExp) : Given a ciphertext $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{z} + t \cdot \boldsymbol{I})$, the output ciphertext $\mathsf{ct}' = \mathsf{Enc} \circ \mathsf{Ecd}(\exp(2\pi i \cdot \boldsymbol{z}/t))$ by homomorphically evaluating a complex exponential $x \to \exp(2\pi i \cdot x/t)$.
(5) **Homomorphic Look-up Table** (LUT_f) : Given a ciphertext $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}(\exp(2\pi i \cdot \boldsymbol{z}/t))$, the output ciphertext $\mathsf{ct}' = \mathsf{Enc} \circ \mathsf{Ecd}(f(\boldsymbol{z}))$ by homomorphically evaluating a kth order hermite interpolation $\exp(2\pi i \cdot x/t) \to f(x)$. Since Hermite interpolation is used, error cleaning is performed in this subroutine.

B Noise Analysis

B.1 Noise Analysis for Comparison Algorithm

For a simplified dominant noise analysis, we conduct the analysis under the following assumptions. We first decompose the algorithm into the base case, the combine phase, and the evaluation of ϕ. At each stage, we denote the maximum noise among the input ciphertexts by $e \ll 1$, and analyze the subsequent noise growth with respect to this value e.

- Base case. In the base case, the algorithm evaluates Equations (3) and (4). Since the input ciphertexts encrypt values in $\{0, 1\}$, the error bound of Equation (3) is $6e$, while the error bound of Equation (4) is $8e$.
- Combine phase. In the combine phase, the ciphertexts L_0, R_0, L_1, and R_1 encrypt values in $\{-1, 0, 1\}$. Therefore, the error bounds of the computations in Steps 15 and 17 are bounded by $2e$ and $3e$, respectively.
- Evaluation of ϕ. At this stage, the inputs are restricted to $\{-1, 0, 1\}$. Hence, the error bound of evaluating ϕ is given by $\frac{3}{2}e$.

Let $\{\boldsymbol{x}\}_{0 \le i < p}$ and $\{\boldsymbol{y}\}_{0 \le i < p}$ be two p-bit input sequences, and assume that the maximum noise of each input bit is bounded by $e \ll 1$. We further assume that the noise incurred in each base case is bounded by $8e$, and that each invocation of the combine phase introduces at most $3e$ noise. Since the combine phase is invoked $p/2 - 1$ times, we obtain the following rough worst-case upper bound on the accumulated noise of Algorithm 1 :

$$e_{\mathsf{cmp}} < 12 \cdot 3^{\log p} \cdot e$$

B.2 Noise Analysis for Traversal Algorithm

For a simplified dominant noise analysis of the blind branch selection procedure, we adopt the same analytical approach as in Section 3.2.1 and focus on the leading noise terms. Throughout the analysis, we denote by $e \ll 1$ the maximum noise among the input ciphertexts at each stage and track the subsequent noise growth with respect to e.

- Construction of Prod. We analyze the noise introduced in the construction of Prod independently of other stages. In Steps 1–9 of Algorithm 3, each entry of $\mathsf{Prod}_{d,E}$ and $\mathsf{Prod}_{d,O}$ is computed as a product of at most $d/2$ ciphertexts from $\{S_i\}_{0 \le i < d}$. If the maximum noise of the input ciphertexts $\{S_i\}$ is bounded by e, then the noise of each entry of $\mathsf{Prod}_{d,E}$ and $\mathsf{Prod}_{d,O}$ is bounded by $\frac{d}{2} \cdot e$.
- One-hot vector construction. We next consider the noise of the one-hot vector construction as a separate step. In Steps 10–18 of Algorithm 3, the one-hot vectors $\mathsf{OH}_{d,E}$ and $\mathsf{OH}_{d,O}$ are obtained by summing entries of $\mathsf{Prod}_{d,E}$ and

$\mathsf{Prod}_{d,O}$, respectively. Each sum consists of at most $d/2$ terms. Assuming that the maximum noise of the inputs is bounded by e, the noise of $\mathsf{OH}_{d,E}$ and $\mathsf{OH}_{d,O}$ is bounded by $\frac{d}{2} \cdot e$.

- Inner product evaluation. Finally, we analyze the noise of the inner product computation independently. In Steps 7–12 of Algorithm 4, the algorithm evaluates inner products involving a total of 2^d homomorphic additions. If the maximum noise of the input ciphertexts is bounded by e, then the noise of the resulting encrypted attribute-bit sequence $\{\hat{\boldsymbol{x}}_i\}$ and encrypted threshold-bit sequence $\{\hat{\boldsymbol{t}}_i\}$ is bounded by $2^d \cdot e$.

Let $\{S_i\}_{0 \le i < d}$ denote the encrypted decision values, and assume that the maximum noise among them is bounded by $e \ll 1$. We further assume that the construction of Prod introduces at most $\frac{d}{2} \cdot e$ noise per entry, and that the one-hot vector construction incurs an additional factor of at most $\frac{d}{2}$. Finally, the inner product evaluation involves 2^d homomorphic additions.

Under these assumptions, we obtain the following rough worst-case upper bound on the noise of the output attribute-bit sequence and threshold-bit sequence produced by BsgsBSS:

$$e_{\mathrm{bss}} \;<\; \frac{2^d \cdot d^2}{4} \cdot e.$$

C Optimzied Protocol via Complex Packing

When using the CKKS scheme, which supports arithmetic over complex numbers, it is often beneficial in applications that process only real-valued data to exploit the otherwise unused imaginary part as an optimization. In this section, we describe a protocol algorithm that is optimized by leveraging complex packing in CKKS.

C.1 Packing & Encoding

By leveraging complex packing, we reduce the client-side communication cost from $F \cdot p$ ciphertexts to $F \cdot p/2$ ciphertexts.

(1) Bit Decomposition: $\boldsymbol{x}_i \mapsto (\boldsymbol{x}_{i,0}, \ldots, \boldsymbol{x}_{i,p-1})$ for $0 \le i < F$.
(2) Complex Compression: $\boldsymbol{x}_i \mapsto \boldsymbol{x}_{i,j} \oplus \sqrt{-1} \cdot \boldsymbol{x}_{i,p/2+j}$, where $0 \le j < p/2$ and $\oplus$ denotes element-wise addition.
(3) Encryption: $\mathsf{Enc}(\boldsymbol{x}_{i,j})$ for $0 \le i < F$ and $0 \le j < p/2$.
(4) Transmission: The client sends $\{\mathsf{Enc}(\boldsymbol{x}_{i,j}) : 0 \le i < F,\ 0 \le j < p/2\}$ to the server.

C.2 Optimized Algorithm

In our algorithm, the attribute-vector ciphertexts transmitted by the client are used in both the GEQ procedure and the inner product computations of BsgsBBS.

The GEQ procedure requires ciphertext–ciphertext multiplications, during which the real and imaginary parts of complex-packed ciphertexts become mixed. Therefore, only in this stage, we apply the conjugation operation to separate the real and imaginary parts and then evaluate the original GEQ procedure. Afterwards, we apply the conjugation operation again to recombine the real and imaginary parts into a single ciphertext. This process requires p key-switching operations.

In contrast, for the inner product computations in BsgsBBS, even though ciphertext–ciphertext multiplications are involved, the one-hot vector ciphertexts take values in $\{0, 1\}$, which allows the computations to be carried out directly on complex-packed ciphertexts. With this optimization, the dominant term of the inner product computation is reduced from $p \cdot 2^{d/2}$ to $\frac{p}{2} \cdot 2^{d/2}$.

D Split Hamming Weight

Let $\mathsf{HW}(x)$ denote the Hamming weight of an integer x, i.e., the number of nonzero bits in its binary representation. In Algorithm 6, we describe a simple procedure that splits the set bits of x into two disjoint parts with nearly equal Hamming weights. The algorithm scans the set bits of x and assigns the first $\lfloor \mathsf{HW}(x)/2 \rfloor$ of them to x_0, while assigning the remaining set bits to x_1. As a result, x_0 and x_1 have no common set bit, their bitwise union reconstructs x, and their Hamming weights differ by at most one.

Algorithm 6 SplitHW

Input: A non-negative integer x
Output: Two integers x_0, x_1 such that $x_0 \,\&\, x_1 = 0$, $x_0 \mid x_1 = x$, and $|\mathsf{HW}(x_0) - \mathsf{HW}(x_1)| \le 1$.

$w \leftarrow \mathsf{HW}(x)$
$t \leftarrow \lfloor w/2 \rfloor$
$x_0 \leftarrow 0, x_1 \leftarrow 0$
$c \leftarrow 0$
for $i = \lfloor \log_2 x \rfloor$ down to 0 **do**
  if $((x \gg i) \,\&\, 1) = 1$ **then**
    if $c < t$ **then**
      $x_0 \leftarrow x_0 + 2^i$
      $c \leftarrow c + 1$
    else
      $x_1 \leftarrow x_1 + 2^i$
    end if
  end if
end for
return (x_0, x_1)