SortingHat: Efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering

Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira

imec-COSIC, KU Leuven, Leuven, Belgium.
{kelong.cong,debajyoti.das,Jeongeun.Park,HilderVitor.LimaPereira}@esat.kuleuven.be

Abstract. Machine learning as a service scenario typically requires the client to trust the server and provide sensitive data in plaintext. However, with the recent improvements in fully homomorphic encryption (FHE) schemes, many such applications can be designed in a privacy-preserving way. In this work, we focus on such a problem, private decision tree evaluation (PDTE) — where a server has a decision tree classification model, and a client wants to use the model to classify her private data without revealing the data or the classification result to the server. We present an efficient non-interactive design of PDTE, that we call *SortingHat*, based on FHE techniques. As part of our design, we solve multiple cryptographic problems related to FHE: (1) we propose a fast homomorphic comparison function where one input can be in plaintext format; (2) we design an efficient binary decision tree evaluation technique in the FHE setting, which we call *homomorphic traversal*, and apply it together with our homomorphic comparison to evaluate private decision tree classifiers, obtaining running times orders of magnitude faster than the state of the art; (3) we improve both the communication cost and the time complexity of *transciphering*, by applying our homomorphic comparison to the FiLIP stream cipher. Through a prototype implementation, we demonstrate that our improved transciphering solution runs around 400 times faster than previous works. We finally present a choice in terms of PDTE design: we present a version of SortingHat without transciphering that achieves significant improvement in terms of computation cost compared to prior works, and another version t-SortingHat with transciphering that has a communication cost about 20 thousand times smaller but comparable running time.

1 Introduction

There is a growing demand for machine learning (ML) as a cloud-based service to provide useful services like automatic health assessment, evaluation of property value, data classification, etc. Often this requires the user to provide the service with sensitive data, like the user's DNA profile, medical or financial records etc. Therefore it is crucial to ask whether such *machine learning as a service* can be used by consumers without giving up the privacy of their data.

In this work, we focus on *decision tree* algorithms [Qui86,RM05,AEM13,RM14], which are an important class of classifiers in machine learning, and useful in many scenarios such as health analysis, credit-risk assessment, spam-filtering, etc. [WFH11,FPS02,AEM13,KTG06,SG11] Our scenario consists of a server holding a decision tree model and a client wanting to classify their private data using the server's decision tree model without revealing the data to the server. An obvious solution would be the server sending the decision tree model to the client and the client running the computation locally — however, that beats the purpose of ML as a service. So, can we design an efficient *private decision tree evaluation* (PDTE) algorithm, where a server has a decision tree classification model and a client wants to use the computation power and the model of the server to classify their private data without revealing the data to the server?

The PDTE problem can be considered as a *secure two-party computation* problem [HV16,KO04], and generic algorithms (e.g., based on Yao's garbled circuits [Yao86,BHKR13,BHR12,KRRW18]) can be used to solve it. However, such generic algorithms have high communication costs, require the client to participate in the computation, and are computationally less efficient for the server compared to a solution specifically designed for the problem. There are a few recent works that attempt to provide solutions [WFNL16,TKK19,TBK20] specific to this problem based on fully homomorphic encryption (FHE) [GHS12,vGHV10,CRRV17] or somewhat homomorphic encryption (SHE) [LMSV12,CS16] techniques. Most of them are not very practical for ML as a service, because of the high communication and computation overhead for both the client and the server. The work by Tueno et al. [TBK20] proposes a non-interactive protocol and manages to bring down the overhead for the client. However, the computation overhead for the server is still very high (several seconds of computation time to run the classification on each input for a tree with $50-500$ decision nodes). Can we design a non-interactive protocol for PDTE with low computation and communication overhead, so that it can be used in practice?

1.1 Our Contribution

We answer the above question positively by designing a new efficient and non-interactive private decision tree evaluation (PDTE) protocol, SortingHat, that achieves low computation overhead for the server, $O(1)$ communication overhead[1] in the sense that a client does not need any interaction with a server apart from querying, and very low (only a few milliseconds per query) computation overhead for the client. Furthermore, we provide an even more bandwidth-efficient version of our design, which we call t-SortingHat, that uses an efficient transciphering technique to drastically improve the query size. In our design, we make the following cryptographic breakthroughs:

1. Homomorphic comparison, taking two ciphertexts as inputs, is a well-known and well-studied problem: there are several solutions to it and all past PDTE algorithms based on FHE use such solutions [CGGI20,IZ21,CKK20,LZS18]. We propose new homomorphic comparison algorithms that takes one ciphertext and one plaintext as inputs, and are more efficient than ciphertext-ciphertext comparison. As a result, we can run the comparisons much faster than existing generic homomorphic comparison functions. In some cases, our comparison function requires only two polynomial multiplications which is much cheaper than multiplication over two ciphertexts consisting of more than 16 polynomial multiplications as done in the similar work [LZS18].

2. We design an efficient binary decision tree evaluation technique using FHE, which we call *homomorphic traversal*. It requires only one external product for each decision node (excluding the cost of homomorphic comparison on each node). Compared to the previous works on FHE-based PDTE (e.g., [TBK20]), our method halves the number of multiplications to evaluate a decision tree. We instantiate our method using much less expensive homomorphic operations than the previous works, which had to choose between a really expensive FHE scheme (e.g., BGV [BGV12]) and an efficient GSW-like FHE scheme (e.g., TFHE [CGGI20]) that comes with the overhead of one bootstrapping for every decision node. We discuss more about these tradeoffs in Section 1.2. Our *homomorphic traversal* achieves the best of both worlds by using TFHE-based FHE while eliminating the requirement of bootstrapping completely. This improves the cost of tree traversal by a huge margin (e.g., each bootstrapping takes more than 600 external products for typical parameters).

3. We improve an existing transciphering method based on the stream cipher FiLIP [HMR20]. The main building blocks of FiLIP are a homomorphic Hamming weight (HW) and a comparison function. We propose a simpler way of computing the HW, which drastically reduces the number of

[1] The communication overhead also includes the ciphertext expansion factor which depends on the security parameter λ of the FHE scheme. However, that is independent of the size of the decision tree, and we drop that factor whenever we mention communication overhead, for ease of description.

homomorphic operations needed in this step, then we use our homomorphic comparison function instead of the general comparison algorithms. As a result, our new FiLIP evaluation is two orders of magnitude faster than [HMR20]. Our improvements for transciphering are useful in many other scenarios (as discussed in Section 2.2) and are of independent interests.

4. We combine the improved transciphering technique with our tree traversal algorithm. With this, we obtain the PDTE solution t-SortingHat which is slower than our first solution SortingHat, but has a much better communication cost, i.e., to classify an input with t bits under a security level of λ bits, a user just has to send $t + O(\lambda)$ bits to the server, which is optimal in the number of input bits and has a very weak dependency on the security parameter. In concrete terms, $t + O(\lambda) \approx t$ and we reduce the upload size by a factor of 20 thousand compared to previous solutions.

We instantiate all our constructions using the most modern FHE schemes, like TFHE and FINAL [BIP+22], since they support efficient binary circuit evaluation and slow noise growth. Using a prototype implementation, we demonstrate that SortingHat (without transciphering) provides a performance improvement of over three orders of magnitude over prior work [TBK20]. For each classification, SortingHat takes only 42.3 milliseconds for computation with only one thread, compared to 940 milliseconds with 16 threads of the previous work [TBK20] for the same dataset. Unfortunately, t-SortingHat (with transciphering) can not fully utilize the advantages of our homomorphic comparison technique used in SortingHat due to technical reasons discussed in Section 5.1. Still, the running times of t-SortingHat are comparable to previous PDTE algorithms, and for some databases, it even has better running times than them, despite having the additional computation overhead for transciphering; and the communication cost for the query is about 20 thousand times smaller. The computation overhead of transciphering in t-SortingHat varies from 0.7% to 35%, depending on the depth of the tree.

1.2 Existing Works

The problem of private decision tree evaluation (PDTE) can be solved using secure two-party computation techniques [HV16,KO04]. The generic algorithms for such problems based on Yao's garbled circuits [Yao86,BHR12,KRRW18] are not suitable for ML as a service because they have high computation overhead, and many rounds of interactions are required which results in a high communication overhead. There exist some specialized solutions [BPSW07,BFK+09,TKK19] based on a combination of garbled circuits and other techniques, however, they also face the problems of high communication and computation overhead, and therefore, not suitable for our purpose.

Some designs attempt to improve communication and computation overhead by employing homomorphic encryption schemes.[2] Bost et al. [BPTG15] propose a design based on FHE where they express the decision tree as a polynomial whose output is the result of the classification. Another line of work [WFNL16,TMZC17] introduce solutions based on additive homomorphic encryption. However, they all incur high computation (for both client and server) and communication overhead, and thus, not very practical.

Lu et al. [LZS18] improve on [TMZC17] by using a non-interactive comparison function, and provide a non-interactive decision tree design. However, the communication (of $O(2^d)$ for a tree depth d) and computation overhead for the client is still very high.

Tueno et al. [TBK20] propose a non-interactive PDTE in the FHE setting, and significantly improve the computation overhead and bring down the communication overhead to $O(1)$. By using

[2] By homomorphic encryption we consider partially or somewhat or fully homomorphic encryption.

Table 1: Protocol and system parameters for private decision tree evaluation

$\mathcal{T}$	The decision tree
m	Number of decision nodes
M	Total number of nodes
$\mathcal{D}$	Set of all decision nodes
$\mathcal{L}$	Set of all leave nodes
n	Dimension of the attribute vector
d	Depth of the decision tree
k	The number of classification labels
$\mathbf{x}$	Attribute vector—$\{x_0, \dots, x_n\}$
$\mathbf{a}$	Function that assigns an attribute index to each decision node
$\mathbf{t}$	Function that assigns a threshold value to each decision node
$\mathcal{M}$	The decision tree model—$(\mathcal{T}, \mathbf{t}, \mathbf{a})$
λ	Security parameter
$\langle \mathbf{v}, \mathbf{w} \rangle$	Dot product of two vectors $\mathbf{v}, \mathbf{w}$
$\mathbf{x}_i, \mathbf{x}[i]$	The i-th component of vector $\mathbf{x}$
$\|\cdot\|$	Infinity norm
$\log(\cdot)$	Logarithm function with base 2
N	The maximum degree of a polynomial
t	Plaintext modulus
q	Ciphertext modulus
$\mathcal{R}$	$\mathbb{Z}[X]/(X^N + 1)$ for a positive integer N
$\mathcal{R}_q$	$\mathbb{Z}[X]/(X^N + 1) \mod q$ for positive integers q, N
M_0	The constant term of a polynomial $M(X)$

BGV [BGV12] and SIMD slots [SV11], their work outperforms all the prior works in the amortized scenario if the client wants to classify hundreds or thousands of attribute vectors. However, their protocol comes with a few tradeoffs — BGV ciphertexts are much larger in size compared to TFHE ciphertexts, and the noise grows much faster with the number of multiplications. To keep the noise small, they use a tree-traversal algorithm which has a computational complexity of $O(2^d \cdot \log d)$. The authors provide a TFHE version of their protocol that has a computational complexity of $O(2^d)$ and much less communication overhead, 2 KBs in contrast to 1.7 MBs in the BGV version if the attribute vectors cannot be batched. This version performs at least five times faster than the BGV version for a single comparison, however, cannot support SIMD and does not get the advantage in the amortized scenario.

Our PDTE design SortingHat outperforms their TFHE version by a significant margin. (we refer to Section 7.2 for a detailed performance comparison). Our scheme without transciphering achieves the same communication overhead as their TFHE version. With transciphering we further reduce the communication overhead to *zero* for upload by compromising the server performance.

About Transciphering. Ciphertexts of a fully homomorphic encryption scheme is much larger than traditional encryption schemes [PT20,ACLS18]. This issue is typically addressed using transciphering, where the protocol messages are sent using a symmetric key cipher and the server needs to homomorphically decrypt these messages using the encrypted secret keys that it obtains during setup. A more detailed introduction to transciphering and a concrete scheme (FiLIP [MCJS19]) is given in Section 2.2, since our transciphering technique builds directly on top of the FiLIP design.

2 Preliminaries

Notation: We denote λ as the security parameter. We define vectors and matrices in lowercase bold and uppercase bold, respectively. Dot product of two vectors $\mathbf{v}, \mathbf{w}$ is denoted by $\langle \mathbf{v}, \mathbf{w} \rangle$. For a vector $\mathbf{x}$, both $\mathbf{x}_i$ and $\mathbf{x}[i]$ denote either the i-th component scalar or the i-th element of an ordered finite set. The norm notation $\|\cdot\|$ denotes the infinity norm. $\log(\cdot)$ is the logarithm function with base 2. Let $\mathcal{R}$ and $\mathcal{R}_q$ denote $\mathbb{Z}[X]/(X^N+1)$ and $\mathbb{Z}[X]/(X^N+1) \mod q$, respectively, for positive integers q, N. The constant term of a polynomial $M(X)$ is denoted by M_0. We summarize the notations in Table 1.

2.1 Homomorphic Encryption

Our algorithms can be instantiated with any GSW-like homomorphic encryption scheme, such as FHEW [DM15], TFHE [CGGI20], GAHE [Per21] or FINAL [BIP+22]. These schemes have the benefit of keeping noise overhead additive after a long sequence of homomorphic multiplications by utilizing asymmetric noise propagation. For concreteness, we describe them using the notation of FHEW and TFHE, that is, with scalar ciphertexts defined over the Learning With Errors (LWE) problem and its ring variant (RLWE). Let us discuss three different ciphertext formats which put together such schemes.

LWE, RLWE, and RGSW Ciphertexts. We define a ciphertext modulus as q and plaintext modulus as t, where $t \ll q$. Let us denote $\Delta = \lfloor q/t \rceil$.

– An LWE ciphertext is defined as $\mathbf{c} := (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, where $b = \langle \mathbf{a}, \mathbf{s} \rangle + \Delta \cdot m + e$ for a message $m \in \mathbb{Z}_t$ and a secret key $\mathbf{s} \in \mathbb{Z}^n$. $\mathbf{c}$ is denoted by $\mathsf{LWE}_{n,t,q}(m)$.

– An RLWE ciphertext is defined as $c := (a, b) \in \mathcal{R}_q^2$, where $b = a \cdot s + \Delta \cdot m + e$ for a message polynomial $m \in \mathcal{R}_t$ and a secret key $s \in \mathcal{R}$. c is denoted by $\mathsf{RLWE}_{N,t,q}(m)$.

– Given a base B_g and $\ell = O(\log q)$, we define a gadget vector $\mathbf{g} = (1, B_g, \ldots, B_g^{\ell-1})^t$. An RGSW ciphertext is a form of $\mathbf{C} := (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^{2\ell \times 2}$, where $\mathbf{b} = \mathbf{Z} + m \cdot \mathbf{G}$, where each row of $\mathbf{Z}$ is an RLWE encryption of 0 and $\mathbf{G}$ is a gadget matrix which is defined by $\mathbf{G} = \mathbf{I}_2 \otimes \mathbf{g}$.

Homomorphic Operations and Basic Algorithms. We introduce some homomorphic operations and basic algorithms that we use in this paper.

Homomorphic Addition. Let $c_1 := (a_1, b_1)$ and $c_2 := (a_2, b_2)$ be two RLWE ciphertexts. The addition between two ciphertexts is defined as $c_+ := c_1 + c_2 = (a_1 + a_2, b_1 + b_2)$.

Gate Operation. Homomorphic NOT gate of a LWE ciphertext $\mathbf{c}$ encrypting a bit m is efficiently instantiated as $\mathsf{NOT}(\mathbf{c}) := (\mathbf{0}, \Delta) - \mathbf{c}$. This operation does not require bootstrapping at the end unlike other gate operations such as XOR, AND, OR, and NAND [CGGI20], hence the computation is almost free.

Plaintext-Ciphertext Product. We define a multiplication between a plaintext polynomial $P(X) \in \mathbb{Z}_t[X]$ and an RLWE ciphertext $c := (a, b)$ as

$$\mathsf{Plain.Mult}(c, P(X)) \rightarrow c' := (a', b'),$$

where $a' = a' \cdot P(X)$ and $b' = b \cdot P(X)$.

External Product. We define a homomorphic multiplication between an RLWE ciphertext and an RGSW ciphertext, which is called external product in [CGGI16a], denoted as $\boxdot$, as following: $\mathbf{A} \boxdot \mathbf{b} = \mathbf{G}^{-1}(\mathbf{b}) \cdot \mathbf{A}$, where $\mathbf{A}$ is a $\mathsf{RGSW}(m_A) \in \{0,1\}$ and $\mathbf{b}$ is a $\mathsf{RLWE}(m_b)$ sample of $\mu_b \in \mathcal{R}_q$ and $\mathbf{G}^{-1}(\cdot)$ is the gadget decomposition function which satisfies $\langle \mathbf{G}^{-1}(a), \mathbf{G} \rangle = a$ for $a \in \mathcal{R}_q$.

Extract. Given an RLWE ciphertext encrypting $m(X) \in \mathcal{R}_t$, it outputs an LWE encryption of m_i which is the i-th coefficient of $m(X)$ for some $i \in [0, \dots N-1]$. This algorithm is defined as $\mathsf{SampleExtract}$ in [CGGI20] which does not add any noise in the output and the computation time is almost free in practice since it only rearranges the order of components of input vector/polynomial. In this paper we call this algorithm $\mathsf{Extract}_{\mathsf{RLWEtoLWE}}$.

Conversion. There is also an efficient conversion algorithm from a RLWE ciphertext encrypting a polynomial $m(X)$ to a RGSW ciphertext encrypting the constant term of $m(X)$ which is m_0 (Algorithm 4 of [CCR19]). In our case, the algorithm takes ℓ RLWE ciphertexts

$$\{\mathsf{RLWE}_{N,t,q}(\sum_{i=0}^{N-1} m_i \cdot B_g^j \cdot X^i))\}_{j \in [\ell]}$$

as input and then outputs $\mathsf{RGSW}(m_0)$. To run this algorithm, additional public evaluation key denoted by ksk sent by the client beforehand is needed. We call the algorithm $\mathsf{RLWEtoRGSW}$ throughout the paper.

2.2 Transciphering: a General Strategy to Reduce Communication Cost

The ciphertext expansion, i.e., the ratio between the *size of ciphertext* and the *size of plaintext*, is one way of measuring the efficiency of any cipher, as it represents the communication overhead incurred by sending encrypted data instead of the data in clear.

It is well known that FHE schemes perform badly on this metric because their ciphertexts are usually large (the ratio is almost 10 for most applications [PT20,ACLS18]). To amend this problem, several papers have proposed that the client could use symmetric ciphers to encrypt the data, and then the server could use FHE to evaluate the decryption of the symmetric cipher, thus obtaining FHE encryptions of the client's data. In this scenario, there is a setup phase where the client generates the symmetric key k, and the keys for the FHE scheme sk and pk. then the client encrypts k under sk, generating $\zeta := \mathsf{FHE.Enc}_{\mathsf{sk}}(k)$, and sends (ζ, pk) to the server.

To use the application provided by the server, the client encrypts their data x using the symmetric cipher and sends $\mathsf{Sym.Enc}(x)$ to the server instead of $\mathsf{FHE.Enc}(x)$. Since symmetric ciphers typically have very low ciphertext expansion, close to one, the amount of data that the client sends is close to the size of the data in clear. The server can compute $\mathsf{Eval}(\mathsf{Sym.Dec}, \mathsf{Sym.Enc}(x), \zeta)$, which yields $\mathsf{FHE.Enc}(\mathsf{Sym.Dec}_k(\mathsf{Sym.Enc}(x))) = \mathsf{FHE.Enc}(x)$. Then the server can proceed with the homomorphic computations as usual.

There are specialized FHE-friendly ciphers whose decryption functions can be easily evaluated by FHE schemes. In this work, we consider the FiLIP cipher [HMR20], a stream cipher specifically designed to be evaluated with GSW-like FHE schemes like TFHE and FHEW.

FiLIP Cipher. FiLIP is a binary stream cipher based on filter permutator and non-linear functions [MCJS19]. The encryption and decryption algorithms work as follows: let $K \in \{0,1\}^Z$ be the the secret key; for each bit m_i of the message, we use a forward secure PRNG to sample

- $\mathbf{s}_i$: a vector with z bits of K,

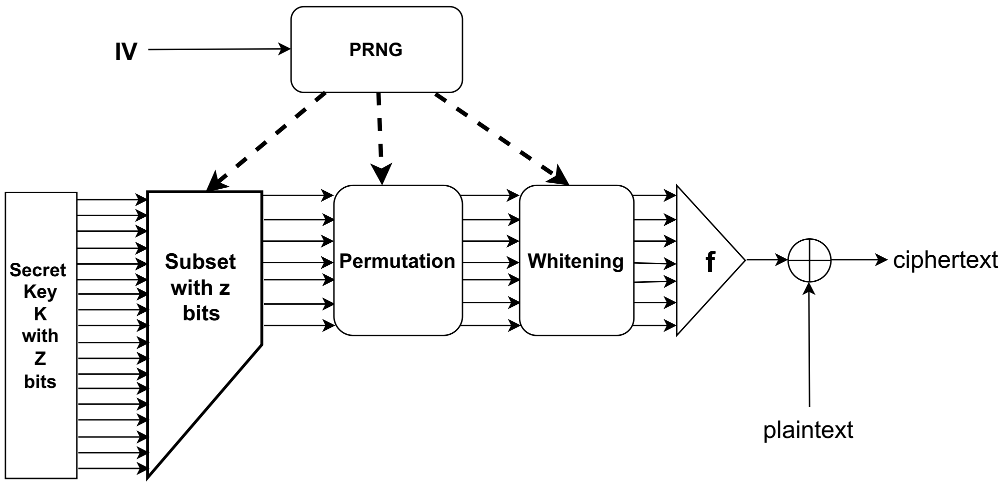

Fig. 1: FiLIP encryption of one bit. Using the PRNG, we select a subset of the bits of the keys, then we shuffle them and apply an XOR with each bit of the whitening vector. Finally, we apply the non-linear function f and XOR the result with the plaintext.

– P_i: an z to z permutation,
– $\mathbf{w}_i$: an z-dimensional binary vector called *whitening*.

Then, for some function $f : \{0,1\}^z \to \{0,1\}$ fixed beforehand, we compute $c_i := m_i \oplus f(P_i(\mathbf{s}_i) \oplus \mathbf{w}_i) \in \{0,1\}$. One round of FiLIP is illustrated in Figure 1.

We implemented the variant called FiLIP-144 in [HMR20], which consists in setting $Z = 2^{14}$, $z = 144$ and f as the function $\mathsf{XTHR}_{81,32,63}$ described in Definition 2. We note that those parameters of FiLIP-144 yield 128 bit security, following the analysis in [MCJS19].

Definition 1 (Threshold Function (Definition 10 of [HMR20])). *Let $s \in \mathbb{N}^*$. For any positive integer $d \leq s+1$, the boolean function $\mathsf{T}_{d,s}$ is defined as:*

$$\forall x = (x_1, \ldots, x_s) \in \mathbb{F}_2^s,\ \mathsf{T}_{d,s}(x) = \begin{cases} 1 & \text{if } \mathsf{W_H}(x) \geq d, \\ 0, & \text{otherwise} \end{cases}$$

where $\mathsf{W_H}(x)$ is the Hamming weight of a binary vector x.

Definition 2 (XOR-THR Function (Definition 11 of [HMR20])). *For any positive integers $k, d,$ and s such that $d \leq s+1$, and for all $z = (x_1, \ldots, x_k, y_1, \ldots, y_s) \in \mathbb{F}_2^{k+s}$, $\mathsf{XTHR}_{k,d,s}$ is defined as:*

$$\mathsf{XTHR}_{k,d,s}(z) = \mathsf{XOR}_k(x_1, \ldots, x_k) + \mathsf{T}_{d,s}(y_1, \ldots, y_s) \in \mathbb{F}_2,$$

where $\mathsf{XOR}_k(x_1, \ldots, x_k) = x_1 + \cdots + x_k \in \mathbb{F}_2$.

2.3 Decision Trees and Private Decision Tree Evaluation

Here we introduce the definitions and notations related to decision trees. Our notations are similar to that of previous works [TKK19,TBK20], and are summarized in Table 1, A decision tree implements a function $\mathcal{T} : \mathbb{Z}^n \to \{\tau_0, \ldots, \tau_{k-1}\}$ that maps an attribute vector $\mathbf{x} = (x_0, \ldots, x_{n-1})$ to a finite set of classification labels $\{\tau_0, \ldots, \tau_{k-1}\}$.[3] Decision trees are binary tree structures with a collection of decision nodes and leaf nodes. A decision tree model $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$ consists of the function $\mathcal{T}$ and the following functions on the nodes:

- $\mathbf{t}$ is a function that assigns to each decision node a threshold value, $\mathbf{t} : [0, m-1] \to \mathbb{Z}$.
- $\mathbf{a}$ is a function which assigns to each decision node an attribute index, $\mathbf{a} : [0, m-1] \to [0, n-1]$.
- lab is a labeling function that assigns to each leaf node a label, $\mathsf{lab} : [m, \ldots, M-1] \to \{\tau_0, ..., \tau_{k-1}\}$.

Node Indices. Given a decision tree, the index of a node is its order as computed by breadth-first traversal, starting at the root with index 0. If the tree is complete, then a node with index v has the left child at index $2v+1$ and the right child at index $2v+2$. With this indexing scheme, the leaves of the tree are read from left to right, corresponding with the ordering $\ell_0, \ldots \ell_{2^s-1}$, where 2^s is the number of nodes of the complete tree.

Decision Tree Evaluation. Given an attribute vector $\mathbf{x} = (x_0, \ldots, x_{n-1})$ and a decision tree model $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$, then starting at the root, decision tree evaluation functionality evaluates at each decision node $v \in [m]$ the decision $b \leftarrow \left[x_{\mathbf{a}(v)} \geq \mathbf{t}(v)\right]$ and moves either to the right (if $b = 0$) or the left (if $b = 1$) child node. The evaluation returns the label of the reached leaf as the result of the computation, denoted by $\mathcal{T}(x)$.

3 Building Blocks

In this section, we introduce our novel algorithms; homomorphic comparison and homomorphic traversal, which are key building blocks of PDTE.

3.1 Homomorphic Comparison

In this section, we introduce our novel homomorphic comparison algorithms when one operand is in clear. The techniques are different depending on how input messages are encoded. The first case assumes that the encrypted input is encoded in the exponent of the variable X of a polynomial. And the last one assumes that the inputs are encryptions of a bit.

A Simple Comparison Function for Fewer Bits. In a decision tree, an input value sent by the client is compared to a threshold value in each node. For our scenario, we require a homomorphic comparison function that outputs 1 if the input is larger than a threshold value d, 0 otherwise. Such functions are typically instantiated based on TFHE by using bootstrapping technique [ZS21] or deterministic automata [CGGI20]. Since the latter technique consists of only a few external products to implement the desired function per bit, it is much faster than the former. However, if we use deterministic automata, the client has to send two types of ciphertexts (RLWE and RGSW ciphertexts) in our scenario and the size of a RGSW ciphertext is larger than a RLWE ciphertext.

More importantly, such homomorphic comparison functions are built for generic scenarios, i.e., two values for comparison are encrypted. However, in some cases, a computing party runs the comparison function with its own data which is in cleartext. Therefore, we propose a more optimal technique for such scenarios, detailed in Algorithm 1, that outputs an RLWE ciphertext encrypting a polynomial with the constant term being desired output bit (either 1 or 0). If we

[3] Typically attribute vectors are in the space $\mathbb{R}^n$, so we assume a fixed-point encoding of the values.

use $\mathsf{Extract}_{\mathsf{RLWEToLWE}}$ at the end, one can get an LWE ciphertext of comparison output with zero computational overhead.

The idea of encoding the comparison result in the constant term of a polynomial is similar to [LZS18], however, both the inputs in their case are ciphertexts; so the computation is more expensive than ciphertext-plaintext operations in general.

Algorithm 1 Comparison function PolyComp.

```
1: Input: $c := \mathsf{RLWE}_{N,t,q}(X^M)$ and a threshold value $\bar{t}$.
2: Output: $\mathsf{RLWE}_{N,t,q}(M(X))$; $M_0 = 1$ if $M \geq \bar{t}$, otherwise $M_0 = 0$, where $M_0$ is the constant term of $M(X)$.
3: Let $T(X) := X^{2N-N} + X^{2N-(N-1)} + \cdots + X^{2N-\bar{t}}$
4: compute $\mathsf{Plain.Mult}(c, T(X)) \to c'$
5: return $c'$.
```

Correctness. Suppose we want to compare a value $M \leq N$ with a threshold $\bar{t}$. Let $c = (a, b = a \cdot s + \Delta \cdot X^M + e) \in \mathcal{R}_q^2$ be an input RLWE ciphertext which encrypts X^M. After the test vector $\mathcal{R}_q$ is multiplied to both initial a and b (after the line 4), it produces $c' = (a', b')$ where $a' = a \cdot T(X)$ and $b' = b \cdot T(X) = a \cdot T(X) \cdot s + \Delta \cdot X^{\mu} \cdot T(X) + e \cdot T(X)$. Therefore, the result c' is an RLWE ciphertext encrypting $T(X) \cdot X^M$. We want to have the comparison output bit on the constant term of the message. In other words, T_0 is 1 if $\bar{t} \leq M \leq N$, 0 otherwise. Therefore, the result holds.

Although the proposed comparison has a constant computation complexity up to $\log N$ bits, it has a limitation — it only supports up to $\log N$-bit comparisons. If one wants to compare larger number of bits, it is necessary to increase the underlying scheme's parameters or use the extension of the above algorithm (discussed in A). Therefore, we suggest another technique below to handle larger number of bits for the general case.

Amortized Comparison: A New Strategy to Compare an Encrypted Message With Several Plaintexts. We assume that m is encrypted bit by bit with any FHE scheme that allows us to execute binary gates efficiently, specifically AND gates.[4] We use dynamic programming to construct an algorithm that, given an encryption of m and several known values $v^{(1)}, ..., v^{(n)}$, outputs ciphertexts c_i's encrypting 1 if $m > v^{(i)}$ and 0 otherwise.

The main observation is that if we write the comparisons in a recursive way, by combining the results of the comparisons of subsequences of the bits, we can identify equal subsequences of bits in different values $v^{(i_1)}, ..., v^{(i_k)}$, and execute only one comparison for this subsequence, instead of k. For example, $v^{(1)}, ..., v^{(10)}$ have the same four most significant bits, then we can compare m with $\mathsf{msb}_4(v^{(1)})$ and reuse this result for all the other nine $v^{(i)}$'s. So we can run a single 4-bit comparison instead of 10 comparisons.

Let $m_0, ..., m_{\mu-1}$ and $v_0, ..., v_{\mu-1}$ be the binary decomposition of two μ-bit integers m and v, respectively, where m_0 and v_0 are the least significant bits. We define the homomorphic XNOR gate as in Algorithm 2. Notice that evaluating such a gate is almost for free, since a homomorphic NOT gate is evaluated with just a few subtractions. Then, we define the following quantity, which tells us if all the bits $\mu - 1, \ldots, k$ of m and v are equal or not:

$$X_k(v) = \mathsf{XNOR}(\mathsf{Enc}(m_k), v_k) \cdot \ldots \cdot \mathsf{XNOR}(\mathsf{Enc}(m_{\mu-1}), v_{\mu-1}).$$

[4] Schemes like FHEW [DM15], TFHE [CGGI20], GAHE [Per21] or FINAL [BIP+22], have a native homomorphic AND gate, while with other schemes, like BGV [BGV12], AND gates are performed with homomorphic multiplications.

Notice that $X_k(v) = \mathsf{Enc}(1)$ if $(m_k, ..., m_{\mu-1}) = (v_k, ..., v_{\mu-1})$, and $\mathsf{Enc}(0)$ otherwise. In particular, $X_0(v)$ indicates whether $m = v$.

Algorithm 2 XNOR.

```
1: Input: $\mathsf{Enc}(m_i)$ and $v_i$, where $m_i, v_i \in \{0, 1\}$
2: Output: $\mathsf{Enc}(b)$ such that $b = 1$ if $m_i = v_i$ and $b = 0$ otherwise.
3: if $v_i = 1$ then
       return $\mathsf{Enc}(m_i)$.
4: else
       return $\mathsf{NOT}(\mathsf{Enc}(m_i))$.
5: end if
```

Finally, using $\bar{v}_i$ to denote the negation of the i-bit of v, we define the following for the "greater than" comparison

$$\begin{aligned} R_k(v) = &\mathsf{Enc}(m_{\mu-1}) \cdot \bar{v}_{\mu-1} + X_{\mu-1}(v) \cdot \mathsf{Enc}(m_{\mu-2}) \cdot \bar{v}_{\mu-2} \\ &+ \ldots + X_{k+1}(v) \cdot \mathsf{Enc}(m_k) \cdot \bar{v}_k. \end{aligned}$$

Notice that in $R_k(v)$ each term $X_{\mu-i}(v) \cdot \mathsf{Enc}(m_{\mu-(i+1)}) \cdot \bar{v}_{\mu-(i+1)}$ is testing if all the bits from $\mu - i$ to $\mu - 1$ are equal *and* if $m_{\mu-(i+1)} > v_{\mu-(i+1)}$. Thus, the first most significant bit m_j larger than v_j will yield an encryption of 1 and all the other terms will be encryptions of 0. In this case, $R_k(v) = \mathsf{Enc}(1)$. But if there is no $m_j > v_j$, all the terms will be zero and $R_k(v) = \mathsf{Enc}(0)$. Thus, $R_k(v)$ encrypts 1 if and only if $\mathsf{msb}_{\mu-k}(m) > \mathsf{msb}_{\mu-k}(v)$. Otherwise, it encrypts 0. In particular, $R_0(v) = \mathsf{Enc}(m > v)$. We can write $R_k(v)$ and $X_k(v)$ recursively as:

$$\begin{aligned} R_{k-1}(v) &= R_k(v) + X_k(v) \cdot \mathsf{Enc}(m_{k-1}) \cdot \bar{v}_{k-1} \text{ and} \\ X_{k-1}(v) &= \mathsf{XNOR}(\mathsf{Enc}(m_k), v_{k-1}) \cdot X_k(v). \end{aligned}$$

These recursions are the base of our algorithm to compare $\mathsf{Enc}(m)$ with several different values of v. Essentially, once we compute $R_k(v)$, we store it, then to compare $\mathsf{Enc}(m)$ with some u, we can check if the $\mu - k$ most significant bits of u and v are the same, and in this case, we have $R_k(v) = R_k(u)$, so we do not need to compute the comparison on those bits of u. We do the same for $X_k(v)$, until we compute $R_0(v)$ and $X_0(v)$ for all different v's. We show it in detail in Algorithm 3, which is the base case of the recursion of our grouped comparison presented later.

As we show in Lemma 1, the running time of Algorithm 3 is minimized when $\log n$ is larger than μ. Thus, our final amortized comparison, presented in Algorithm 4, uses divide-and-conquer to recursively break down the comparison by first comparing the most significant bits, then the last significant bits, which means reducing μ but keeping n fix, until it reaches the base case, $\log n > \mu$, when we finally call Algorithm 3.

The time complexity of Algorithm 4 is $O(n \cdot \mu / \log n)$, which we prove in Lemma 2. This is asymptotically better than the naive strategy of comparing m with each $v^{(i)}$ by a factor of $\log n$. Moreover, we ran several experiments to verify if the constant hidden in the asymptotic notation is small. In more detail, we fixed some values of μ and varied $\log n$, then we generated random μ-bit integers and ran Algorithm 4 and n times the naive comparison, counting the number of AND gates used in both cases. We observed that for the naive comparison, the number of AND gates is always very close to $1.5 \cdot \mu$. Thus, to compare n integers, we need $1.5 \cdot n \cdot \mu$ AND gates. As it is shown in

Algorithm 3 BaseGroupComp: base case of grouped comparison

```
1:  Input: $\mathsf{Enc}(m_i)$ for $0 \le i \le \mu - 1$ and $\mu$-bit integers $v^{(1)}, ..., v^{(n)}$.
2:  Output: $R_0(v^{(i)})$ and $X_0(v^{(i)})$ for $1 \le i \le n$.
3:  $R_{\mu-1}[0] = \mathsf{Enc}(m_{\mu-1})$
4:  $R_{\mu-1}[1] = 0$
5:  $X_{\mu-1}[0] = \mathsf{NOT}(\mathsf{Enc}(m_{\mu-1}))$
6:  $X_{\mu-1}[1] = \mathsf{Enc}(m_{\mu-1})$
7:  $k = \mu - 2$
8:  while $k \ge 0$ do
9:      $R_k = [],\ X_k = []$                                        ▷ Empty lists
10:     for $1 \le i \le n$ do
11:         $v_{i,k} = v^{(i)}/2^k$                       ▷ $\mu - k$ most significant bits of $v^{(i)}$
12:         $v_{i,k+1} = v^{(i)}/2^{k+1}$
13:         $b = v_{i,k} \bmod 2$                                    ▷ $k$-th bit of $v^{(i)}$
14:         if $X_k[v_{i,k}] = $ null then
15:             $X_k[v_{i,k}] = \mathsf{XNOR}(\mathsf{Enc}(m_k), b) \cdot X_{k+1}[v_{i,k+1}]$
16:             if $b = 1$ then
17:                 $R_k[v_{i,k}] = R_{k+1}[v_{i,k+1}]$
18:             else
19:                 $R_k[v_{i,k}] = R_{k+1}[v_{i,k+1}] + X_{k+1}[v_{i,k+1}] \cdot \mathsf{Enc}(m_k)$
20:             end if
21:         end if
22:     end for
23:     $k = k - 1$
24: end while
25: return $R_0, X_0$
```

Fig. 2, even for very small values of n, like $n = 4$, our algorithm is already cheaper than the naive strategy.

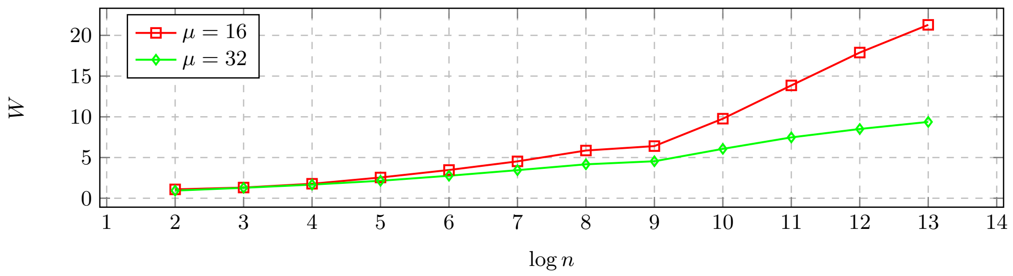

Fig. 2: Performance comparison between naive and recursive grouped comparison algorithms. W denotes the number of AND gates needed by the naive strategy divided by the number of AND gates required by our grouped comparison.

In the rest of this section, we prove our claims about the time complexity of algorithms 3 and 4.

Lemma 1. *The number of homomorphic AND gates executed by Algorithm 3 is less than* $4 \cdot n$ *if* $\log n \ge \mu$ *and less than* $2 \cdot \mu \cdot n - n \cdot (\log n + 1)$ *if* $\log n < \mu$*.*

Proof. First of all, notice that since v_i is known in clear we can evaluate the XNOR gate homomorphically simply by checking if v_i is 1 and returning the input itself, namely, $\mathsf{Enc}(m_i)$, or returning $\mathsf{NOT}(\mathsf{Enc}(m_i))$ if v_i is equal to 0. So, we do not count it in the number of homomorphic gates.

Algorithm 4 RecGroupComp: recursive grouped comparison

```
1: Input: $\mathsf{Enc}(m_i)$ for $0 \le i \le \mu - 1$ and $\mu$-bit integers $v^{(1)}, \ldots, v^{(n)}$.
2: Output: $R_0(v^{(j)})$ and $X_0(v^{(j)})$ for $1 \le j \le n$.
3: if $\mu \le \log(n)$ then
4:     return BaseGroupComp($\{\mathsf{Enc}(m_i)\}_{i=0}^{\mu-1}$, $\{v^{(j)}\}_{j=1}^{n}$)
5: end if
6: $k = \lfloor \mu/2 \rfloor$
7: $L = [\mathsf{Enc}(m_0), ..., \mathsf{Enc}(m_{k-1})]$
8: $M = [\mathsf{Enc}(m_k), ..., \mathsf{Enc}(m_{\mu-1})]$
9: $L_v = [\mathsf{lsb}_k(v^{(1)}), ..., \mathsf{lsb}_k(v^{(n)})]$
10: $M_v = [\mathsf{msb}_k(v^{(1)}), ..., \mathsf{msb}_k(v^{(n)})]$
11: $R_{\mathsf{msb}}, X_{\mathsf{msb}} = \mathsf{RecGroupComp}(M, M_v)$
12: $R_{\mathsf{lsb}}, X_{\mathsf{lsb}} = \mathsf{RecGroupComp}(L, L_v)$
13: $R = [],\ X = []$                                                ▷ Empty lists
14: for $1 \le i \le n$ do
15:     if $X[v^{(i)}] = \text{null}$ then
16:         $X[v^{(i)}] = X_{\mathsf{msb}}[M_v[i]] \cdot X_{\mathsf{lsb}}[L_v[i]]$
17:         $R[v^{(i)}] = R_{\mathsf{msb}}[M_v[i]] + X_{\mathsf{msb}}[M_v[i]] \cdot R_{\mathsf{lsb}}[L_v[i]]$
18:     end if
19: end for
20: return $R, X$
```

Let $k = \mu - j$. In the j-th iteration of the while loop, because each $v_{i,k}$ has j bits, there are up to $\min(2^j, n)$ different $v_{i,k}$'s. For each of them, we execute up to 2 AND gates (lines 16 and 20), thus, the whole algorithm requires $2 \cdot \sum_{j=2}^{\mu} \min(2^j, n)$ homomorphic binary gates. Now, we have two cases:

– If $n \ge 2^\mu$, then $n = \min(2^j, n)$ for all j and the number of homomorphic binary gates required by Algorithm 3 is $2 \cdot (2^{\mu+1} - 1 - 1 - 2) = 2^{\mu+2} - 2^3 < 4 \cdot n$.

– If $n < 2^\mu$, then let $\ell := \lfloor \log_2(n) \rfloor$. We have then

$$\begin{aligned}\sum\nolimits_{j=2}^{\mu} \min(2^j, n) &= \sum\nolimits_{j=2}^{\ell} 2^j + \sum\nolimits_{j=\ell+1}^{\mu} n \\ &= 2^{\ell+1} - 4 + (\mu - \ell - 1) \cdot n \\ &\le 2 \cdot n - 4 + (\mu - \ell - 1) \cdot n \\ &\le (\mu - \log n + 1) \cdot n - 4\end{aligned}$$

So, the number of homomorphic gates evaluated is essentially $2 \cdot (\mu - \log n + 1) \cdot n - 8$.

Lemma 2. *The number of homomorphic binary gates executed by Algorithm 4 is $O(n \cdot \mu / \log n)$.*

Proof. Let k be an integer such that $\mu/2^k \le \log n < 2 \cdot \mu/2^k$. Now, notice that the cost of that algorithm can be expressed as $T(\mu) = 2 \cdot T(\mu/2) + 2 \cdot \min(2^\mu, n)$. Accounting for k recursive levels, we have

$$\begin{aligned}T(\mu) &= 2^k \cdot T(\mu/2^k) + \sum\nolimits_{i=1}^{k} 2^i \cdot \min(2^{\mu/2^{i-1}}, n) \\ &= 2^k \cdot T(\mu/2^k) + \sum\nolimits_{i=1}^{k} 2^i \cdot n \\ &< 2^k \cdot T(\mu/2^k) + n \cdot 2^{k+1} \\ &< 2^k \cdot 2^{\mu/2^k+2} + n \cdot 2^{k+1} - 2^3 \cdot 2^k\end{aligned}$$

$$< 2^k \cdot 4 \cdot n + 2 \cdot n \cdot 2^k - 2^3 \cdot 2^k$$

and using the fact that $\log n < 2 \cdot \mu/2^k$, we obtain

$$T(\mu) < (2 \cdot \mu/\log n) \cdot 4 \cdot n + 2 \cdot n \cdot (2 \cdot \mu/\log n) \in O\left(\frac{n\mu}{\log n}\right).$$

3.2 Homomorphic Traversal

The goal of homomorphic traversal algorithm is to move a value from the root to a desired leaf in a binary tree of depth L. The path that the value on the root takes depends on the controller bits on every node. In this algorithm, we use a homomorphic selection algorithm as a sub-routine. Let's assume that there is a complete binary tree with level $L = \log m$, where m is the number of leaves. The goal of this sub-algorithm is to obliviously copy the value of a parent node to one of the two child nodes depending on a controller bit by using homomorphic multiplication. In more detail, if a controller bit is 1, a value of the parent node is copied to the right child, otherwise it is copied to the left child. We note that the homomorphic selection algorithm is executed in every internal node.

Let us denote the element contained in the parent node by b and the element of the child node is not determined/updated yet. If the controller c bit is 1, the value of the the parent node (b) is copied to the right child node and the left child node becomes 0. Otherwise, vice versa; The above computations in the left child node (denoted by $\mathsf{HS\text{-}L}$) and the right child node (denoted by $\mathsf{HS\text{-}R}$) are instantiated as follows:

$$\mathsf{HS\text{-}R}(b, c) := c \cdot b, \quad \mathsf{HS\text{-}L}(b, c) := (1 - c) \cdot b,$$

where $\cdot$ denotes homomorphic multiplication. The algorithm of homomorphic selection HomSel is described in Algorithm 5.

Algorithm 5 Homomorphic selection HomSel

```
1: Input: $c$, a ciphertext encrypting a bit, and the value of parent node $a$
2: Output: a new updated value of the left child node $b_l$, and the right child node $b_r$
3: Initialize the child node values $b_l = 0$ and $b_r = 0$
4: $b_r := \mathsf{HS\text{-}R}(a, c)$, $b_l := b - b_r$
5: return $(b_l, b_r)$
```

Now, we can think of a binary tree based structure for homomorphic traversal with $\log m$ depth. As a setup phase, all the nodes are initialized with 0 except the root which contains 1. The desired index i is represented as its bit-decomposed form of $\log m$ binary elements, and each bit is encrypted. Each of those (encrypted) bit is the controller bit of all nodes of each level. After $m-1$ homomorphic selections, the value of the root (which is 1) is copied to a leaf at the position of the desired index (if we read the index of the leaves from the right to left). In fact, our homomorphic traversal algorithm is essentially a homomorphic demultiplexer when there are $\log m$ choice bits, one for every level. This algorithm has a multiplicative depth of $\log m$ and the overall multiplication complexity is $O(m)$. Section 4 describes a more general case when there is a choice for every node, which is why we prefer the term "homomorphic traversal" defined in Algorithm 6. This algorithm emulates the decision tree evaluation homomorphically.

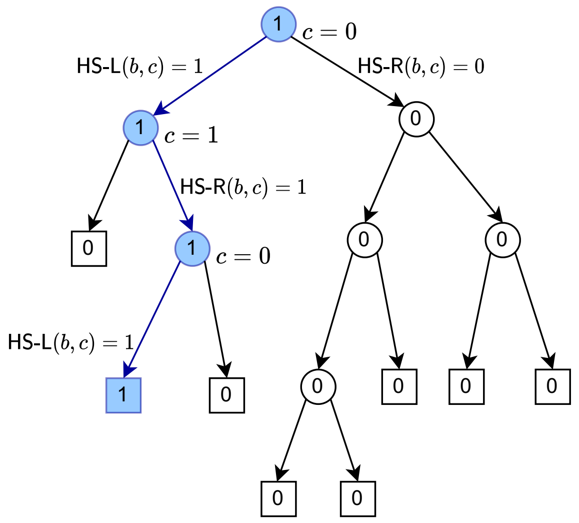

Fig. 3: Example homomorphic traversal on a binary decision tree: the circular nodes represent the internal nodes, the square nodes represent the leaf nodes. The root node is set to 1 initially; given the controller bit c on each internal node, the values of left child and right child are set based on our homomorphic selection algorithm. In the first layer, the left child is set to 1 because HS-L$(b, c) = 1$ (and the right child is set to 0 because HS-R$(b, c) = 0$), where b is the value of the parent (root) node. In the second layer, the right child is set to 1 because HS-R$(b, c) = 1$. Similarly, all the nodes on the blue path becomes 1, and all other nodes that are not on that path becomes 0 in our homomorphic traversal algorithm. Note that, the values of b, c, and the output of HS-R and HS-L functions are encrypted.

The advantage of this algorithm is that we can halve the number of multiplications by reusing the value of one child node for the other one. In other words, we can reuse the value of HS-R for HS-L without additional multiplication. Therefore, the number of multiplications is same as the number of decision nodes. This makes our PDTE perform much better than other existing works. We will discuss this later in Section 7.

Algorithm 6 Homomorphic traversal HomTrav

```
1: Input: $\log m$ controller elements $\{c_0, \dots, c_s\}$, the value of the root $a$.
2: Output: $m$ values of leaves: $z_0, \dots, z_{m-1}$
3: Initialize all the node values $b_i$ as 0 for $i \in \{1, \dots, 2m-2\}$
4: $b_1, b_2 :=$ HomSel$(c_0, a)$                      ▷ Compute first two children of root
5: for $i \leftarrow 2, \dots, s+1$ do
6:     for $j \leftarrow 0, \dots, 2^{i-1}-1$ do
7:         $(b_{2^i+2*j-1}, b_{2^i+2*j}) :=$ HomSel$(c_i, b_{2^{i-1}-1+j})$
8:     end for
9: end for
10: for $i \leftarrow 0, \dots, m-1$ do
11:     $z_i := b_{2^{s+1}-1+i}$
12: end for
13: return $z_0, \dots, z_{m-1}$
```

4 Our Private Decision Tree Evaluation: SortingHat

4.1 System and Security Model

We consider a "single client single server" scenario, where the server holds a decision tree model, and the client holds some private data that he does not want to reveal to the server; however he wants to use the decision tree model held by the server to classify the data. We want to design a protocol where the client provides the private data to the server in a format such that the server can only run the decision tree model on the data and return the classification result to the client; but cannot extract any information about the data or the classification result.

We want to design a non-interactive protocol, i.e., the client has to interact with the server only to send the query and to retrieve the classification result. A protocol with multi-round interactions between the server and the client not only increases the communication overhead, but also requires the client to be active throughout the protocol execution.

Threat Model. Our security goals are similar to previous works [WFNL16,TKK19,TBK20], namely, we want to protect the privacy of the client's data from the server. To do this, we consider the semi-honest model, also known as honest-but-curious adversary, i.e., the server is supposed to follow the protocol and perform all the computation correctly, but can store all the ciphertexts and other data sent by the client and, afterwards, act as any probabilistic polynomial time adversary and perform computation to try distinguish ciphertexts encrypting different messages, recover secret keys, etc. As usual in FHE-based protocols, we do not address circuit privacy.

We want to highlight that our work is complementary to work on differential privacy in the machine learning community. Differential privacy techniques aim to construct classifiers (such as decision tress) from sensitive user training data without leaking more than a bounded amount of information. Whereas, we try to protect the privacy of user data during the classification phase.

4.2 Data and Tree Structure

Let $\mathcal{T}$ be a binary tree with depth d which is is the length of the longest path from the root to a leaf. In each decision node v, the output bit of comparison function is stored as $\mathbf{t}(v)$. Each child node of the parent node v computes different homomorphic operations taking $\mathbf{t}(v)$ on input. The output of this operation, denoted by b_{2v+1} (resp. b_{2v+2}), is stored in the left (resp. right) child node.

Each leaf node l is associated with a classification result $\mathsf{lab}(l) = \tau_j$ for some $j \in [0, \ldots, k-1]$ and a value z_l which is initialized as 0 and updated to an output of homomorphic traversal at the end of the protocol.

4.3 Overall Description

In each decision node, a server computes the comparison function taking its assigned attribute and threshold value on input to obtain bit denoting which node to traverse. Let us denote the attribute vector of client by $\mathbf{x} := (x_0, \ldots, x_{n-1})$. Once the server obtains the output of the comparison function, he runs homomorphic traversal algorithm to obtain the final vector where the only component corresponding to the classification label is encryption of 1, the rest are encryptions of 0.

1. Server initializes the value of each node as 0, denoted by b_j for all $j \in \{0, \ldots, 2m-2\}$ except the root, denoted by b_0, which contains 1.

2. For every decision node v, the server computes a comparison function with its assigned attributes $x_{\mathbf{a}(v)}$ with a node-specific threshold value $\mathbf{t}(v)$. We denote the output of each node by c_v corresponding to the node v for $v \in \{0, \ldots, m-1\}$.
3. Then the server runs the homomorphic traversal algorithm $\mathsf{HomTrav}(\{c_v\}_{v \in \{0,\ldots,m-1\}}, \{b_i\}_{i \in \{0,\ldots,2m-2\}}) \to \{z_l\}_{l \in \mathcal{L}}$.

At the end of the protocol, the only one leaf has a value which is an encryption of 1, corresponding to the label $\mathsf{lab}(l)$ for some $l \in \mathcal{L}$ and the rest are encryptions of 0.

4.4 Our PDTE Instantiated with GSW-like HE Schemes

For each decision node v, the threshold value is already computed as $\mathbf{t}(v)$ in the server's tree model $\mathcal{T}$. Let k be the number of classification labels. The client sends RLWE ciphertexts as encryptions of its attributes. Once the server obtains encryptions of attributes from the client, it runs comparison function for each decision node with its assigned threshold value and stores the outputs as controller bits of the decision nodes described as the first step in the previous section. The output of our comparison function is an RLWE ciphertext, but the homomorphic traversal algorithm takes RGSW ciphertexts as its inputs. To that end, we use the efficient conversion algorithm $\mathsf{RLWEtoRGSW}$ for each output of the comparison function. However, this algorithm takes ℓ number of RLWE ciphertexts which encrypt the same message with different scaling factors. Hence, the client sends basically $n \cdot \ell$ RLWE ciphertexts which are $\mathsf{RLWE}_{N,t,q}(\frac{1}{\Delta} B_g^j \cdot X^{x_i})$ for $i \in [0, \ldots, n-1]$ and $j \in [0, \ldots, \ell-1]$. This setup for the client is addressed in Algorithm 7.

Algorithm 7 Setup of our PDTE.

```
Input: Security parameter $\lambda$.
      ▷ Client's computation
 1: Select FHE parameters for $\lambda$ bits of security
 2: $\mathsf{sk}, \mathsf{pk} \leftarrow \mathsf{FHE.KeyGen}()$
 3: Prepare attribute vector $(x_1, \ldots, x_n)$
 4: for $i \leftarrow 0, \ldots, n-1$ do
 5:     for $j \leftarrow 0, \ldots, \ell-1$ do
 6:         $c_{i,j} \leftarrow \mathsf{RLWE}_{N,t,q}(\frac{1}{\Delta} B_g^j \cdot X^{x_i})$
 7:     end for
 8: end for
 9: generate an evaluation key $\mathsf{ksk}$
10: Send $\mathsf{ksk}, \{c_{i,j}\}_{i \in [0,\ldots,n-1], j \in [0,\ldots,\ell-1]}$ to the server.
```

Next, server runs homomorphic traversal algorithm where the homomorphic multiplication is instantiated as external product between an RGSW ciphertext and an RLWE ciphertext. At the end of running the traversal, server's output is a k dimensional vector consisting of values of leaves which are RLWE ciphertexts encrypting either 1 or 0. The value of the leaf corresponding to the classification label is the only ciphertext encrypting 1. Then server computes dot product between the vector and the vector of classification labels to produce the correct classification label as the last step of our PDTE. The server sends the result as his answer. We describe the process of the server in Algorithm 8.

Algorithm 8 Server's computation for PDTE.

```
1: Input: ${c_{i,j}}_{i\in[0,\dots,n-1],j\in[0,\dots,\ell-1]}$, ksk and classification labels $\tau_0,\dots,\tau_{k-1}$
2: Output: $c := \mathsf{RLWE}_{N,t,q}(\mathcal{T}(x_0,\dots,x_{n-1}))$ which is the resulting classification label
3: $c \leftarrow 0$
4: $a \leftarrow 1$
5: for $i \leftarrow 0 \dots m-1$ do
6:     for $j \leftarrow 0 \dots \ell-1$ do
7:         Let $\mathbf{t}(i)$ be the threshold value of $i$-th node.
8:         Run $\mathsf{PolyComp}(c_{\mathbf{a}(i),j}, \mathbf{t}(i)) \rightarrow b_{i,j}$
9:     end for
10:    Run $\mathsf{RLWEtoRGSW}(\{b_{i,j}\}_{j\in[0,\dots,\ell-1]}, \mathsf{ksk}) \rightarrow \hat{c}_i$
11: end for
12: Run $\mathsf{HomTrav}(\{\hat{c}_i\}_{i\in[0,\dots,m-1]}, a) \rightarrow \{z_l\}_{l\in\mathcal{L}}$
13: for $l \in \mathcal{L}$ do
14:    $c \leftarrow c + \mathsf{Plain.Mult}(z_l, \mathsf{lab}(l))$
15: end for
16: return $c$.
```

Security of SortingHat. The attribute vector sent by the client, the output of the comparison function on each node, and outputs of HS-L() and HS-R(), and the final classification result are encrypted using a semantic secure (IND-CPA) homomorphic encryption scheme (privacy of the client data). Therefore, the server does not learn any information about the client's attribute vector (except the length), or the classification result.

5 Fast Transciphering via FiLIP cipher

We can further improve the communication overhead of SortingHat using standard transciphering techniques. However, transciphering comes with additional computation overhead for the server. We improve the performance of transciphering by providing a faster way to evaluate the decryption function of FiLIP [HMR20] homomorphically for any GSW-like homomorphic encryption scheme. Our transciphering takes as input a ciphertext $c = \mathsf{FiLIP.enc}(m)$ and outputs a low-noise LWE encryption c' of m, so that further homomorphic computation is possible in the usual way, i.e., by evaluating binary gates on c'.

Firstly, there is a setup phase where the client sends to the server RLWE and RGSW encryptions of each bit k_i of FiLIP's secret key K, and the server produces LWE encryptions of $\mathsf{NOT}(k_i)$ and GSW encryptions of both X^{k_i} and $X^{\mathsf{NOT}(k_i)}$. To reduce communication cost, each RLWE ciphertext encrypts N bits. This step is shown in detail in Algorithm 9

To efficiently evaluate the threshold function $\mathsf{T}_{32,63}(\mathbf{y})$, where $\mathbf{y} := (y_1,\dots,y_{63}) \in \{0,1\}^{63}$, our main idea is to start with GSW encryptions of X^{y_i}, then use external products to obtain a ring encryption of $X^{\sum y_i}$. Note that, because this sum is less than N, it holds that $X^{\sum y_i} = X^{\mathsf{W_H}(\mathbf{y})}$, i.e., we obtain the Hamming weight on the exponent. Finally, we multiply by a test-polynomial to perform the comparison with $d := 32$ and extract an LWE sample. At the end of this step, we have $\mathbf{c}' := \mathsf{LWE}_{n,2,q}(\mathsf{T}_{32,63}(\mathbf{y}))$.

The second step consists of computing XOR gates. In a naive implementation, one could use the homomorphic gates provided by the FHE scheme, however, because we are using scaling factor $\Delta = q/2$, we can simply use homomorphic additions, which are much cheaper. In more details, we

Algorithm 9 Setup of homomorphic FiLIP.

```
Input: Security parameter $\lambda$.
    ▷ Client's computation
 1: Select FiLIP and FHE parameters for $\lambda$ bits of security
 2: $\mathsf{sk}, \mathsf{pk} \leftarrow \mathsf{FHE.KeyGen}()$
 3: $K \leftarrow \mathsf{FiLIP.keygen}()$
 4: for $1 \leq i \leq Z$ do
 5:     Let $k_i$ be the $i$-th bit of $K$
 6:     $\mathbf{C}_i \leftarrow \mathsf{RGSW}(k_i)$
 7: end for
 8: for $1 \leq i \leq \lceil Z/N \rceil$ do
 9:     Let $k(X) := \sum_{j=(i-1)\cdot N}^{i\cdot N} k_j \cdot X^j$
10:     $\tilde{\mathbf{c}}_i \leftarrow \mathsf{RLWE}_{N,2,q}(k(X))$
11: end for
12: Send $\mathsf{pk}, \{\mathbf{C}_i\}_{i=1}^{Z}, \{\tilde{\mathbf{c}}_i\}_{i=1}^{\lceil Z/N \rceil}$ to the server.
    ▷ Server's computation
13: for $1 \leq i \leq Z$ do
14:     $u \leftarrow i \bmod N$
15:     $v \leftarrow \lfloor i/N \rfloor$
16:     $\mathbf{c}_i \leftarrow \mathsf{Extract}_{\mathsf{RLWEtoLWE}}(\tilde{\mathbf{c}}_v, u)$        ▷ $\mathsf{LWE}_{n,2,q}(k_i)$
17: end for
18: for $1 \leq i \leq Z$ do
19:     $\bar{\mathbf{c}}_i \leftarrow \mathsf{FHE.NOT}(\mathbf{c}_i)$        ▷ $\mathsf{LWE}_{n,2,q}(\mathsf{NOT}(k_i))$
20:     $\mathbf{C}_i \leftarrow \mathsf{RGSW}(1) + (X-1) \cdot \mathbf{C}_i$        ▷ encryption of $X^{k_i}$
21:     $\bar{\mathbf{C}}_i \leftarrow \mathsf{RGSW}(X) + (1-X) \cdot \mathbf{C}_i$        ▷ encryption of $X^{\mathsf{NOT}(k_i)}$
22: end for
```

assume we have $\mathbf{c}_i := \mathsf{LWE}_{n,2,q}(x_i)$ for the bits x_i of FiLIP's secret key and we compute

$$\mathbf{c} = \mathbf{c}' + \sum_{i=1}^{81} \mathbf{c}_i = \mathbf{c}' + \mathsf{LWE}_{n,q}(\mathsf{XOR}_k(x_1, \ldots, x_k))$$
$$= \mathsf{LWE}_{n,2,q}(\mathsf{XTHR}_{81,32,63}(\mathbf{x}, \mathbf{y}))$$

The last step consists in simply adding this to the FiLIP ciphertext. This procedure is shown in detail in Algorithm 10. The subset, the permutation, and the whitening vector are in clear, therefore, applying them to the encrypted bits of FiLIP's secret key boils down to selecting some ciphertexts pre-computed in the setup phase, thus, this is done almost for free.

5.1 Bandwidth Efficient PDTE: t-SortingHat

As a result of employing our improved transciphering technique, we achieve our bandwidth efficient PDTE t-SortingHat, which we describe below. The client and the server first run the transciphering setup described in Algorithm 9. Then, every time the client wants to classify one input with n attributes and μ bits per attribute, the following protocol is executed:

Client. The client samples an IV v and use it to encrypt the $n \cdot \mu$ bits of the input with FiLIP cipher. The client then send v and the $n \cdot \mu$ encrypted bits to the server.

Server. The server executes the following steps:

1) Runs homomorphic FiLIP decryption function (Algorithm 10) per bit. At the end of this step, server obtains $n \cdot \mu$ LWE ciphertexts.

Algorithm 10 Homomorphic FiLIP.dec.

```
Input: $c = \mathsf{FiLIP.enc}(b)$, an IV, and ciphertexts $\mathbf{c}_i, \bar{\mathbf{c}}_i, \mathbf{C}_i, \bar{\mathbf{C}}_i$ computed in the setup phase.
Output: $\mathsf{LWE}_{n,2,q}(b)$
 1: Sample a random subset $S := \{s_1, ..., s_z\} \subseteq \{1, ..., Z\}$
 2: Sample a random permutation $\pi : S \to S$.
 3: Sample a random whitening vector $\mathbf{w} \in \{0,1\}^z$
 4: for $1 \le i \le z$ do
 5:     $j \leftarrow \pi[s_i]$
 6:     if $w_i = 0$ then
 7:         $\mathbf{c}'_i := \mathbf{c}_j,\ \mathbf{C}'_i := \mathbf{C}_j$
 8:     else
 9:         $\mathbf{c}'_i := \bar{\mathbf{c}}_j,\ \mathbf{C}'_i := \bar{\mathbf{C}}_j$
10:     end if
11: end for
    ▷ Compute the function $\mathsf{XOR}_k(x_1, \dots, x_k)$
12: $\mathbf{x} := \mathsf{LWE}_{n,2,q}(0)$
13: for $1 \le i \le k$ do
14:     $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{c}'_i$
15: end for
    ▷ Compute the threshold function $\mathsf{T}_{d,s}(\mathbf{y})$
16: $\mathbf{y} := \mathsf{RLWE}_{N,2,Q}(1)$
17: for $k+1 \le i \le z$ do
18:     $\mathbf{y} \leftarrow \mathbf{C}'_i \boxdot \mathbf{y}$
19: end for
20: $\mathbf{u} \leftarrow \mathsf{PolyComp}(\mathbf{y}, d)$
21: $\mathbf{t} \leftarrow \mathsf{Extract}_{\mathsf{RLWEtoLWE}}(\mathbf{u}, 0)$
    ▷ Add $\mathsf{XTHR}_{k,d,s}(\mathbf{x}, \mathbf{y})$ to the FiLIP ciphertext
22: return $\mathbf{x} + \mathbf{t} + (\mathbf{0}, (q/2) \cdot c)$
```

2) Runs the group comparison function (Algorithm 4) n times using the threshold values of all decision nodes $\{\mathbf{t}(v_j)\}_{j \in [0,\dots,m-1]}$, where v_j's are decision nodes. We note that the better this algorithm performs the more all $\mathbf{t}(v_j)$'s have common bits.

3) With the output value of the previous step, we run homomorphic traversal (Algorithm 6). We note that the multiplication of homomorphic traversal is instantiated as homomorphic AND gate in our instantiation.

4) Sends the output to the client.

We have to consider a technical issue when it is instantiated with GSW-like homomorphic encryption schemes. Our transciphering method only outputs an encryption of a bit (plaintext modulus $t = 2$) since FiLIP is designed for bit operations. Moreover, it is not straightforward to tweak the algorithm to work over larger plaintext space. We only obtain ciphertexts like $\mathsf{RLWE}_{n,2,q}(m)$ from Algorithm 10 for $m \in \{0,1\}$. Therefore, it is inevitable to use homomorphic bit operations for the rest of computation. As a result, we employ our amortized comparison method addressed in 3.1 and instantiate our homomorphic traversal by using homomorphic AND gate. Despite of this downside, our practical result is better than the existing works [TBK20,LZS18] which also uses homomorphic AND gate, since we use more efficient algorithms as our building blocks, e.g., our comparison function and homomorphic traversal. The efficiency comparison of each building block is discussed in 7.

We notice that now the client just has to send $n \cdot \mu + O(\lambda)$ bits to the server, while in previous solutions the upload cost is at least $\Theta(n \cdot \mu \cdot \lambda \cdot \log \lambda)$, because each FHE ciphertext sent by the client has bit length $\Theta(\lambda \cdot \log \lambda)$.

6 Implementation

We have developed a proof-of-concept implementation of SortingHat; as well as an implementation of our improved transciphering technique for FiLIP cipher and grouped comparison used in t-SortingHat.

Our homomorphic traversal algorithm consists of binary circuits and has $\log m$ multiplicative depth for a complete tree of depth $\log m$, moreover, all $\log m$ multiplications are done sequentially. Therefore, TFHE [CGGI20] which supports efficient binary circuit evaluation and slower noise growth is the best choice for our construction to be instantiated. Our instantiations use the following parameters which supports 128 bits of security.

- Ciphertext modulus $q = 2^{64}$.
- Ring dimension $N = 2048$.
- The standard deviation of the error distribution $\sigma \approx 2^9$ (Note: the σ is interpreted as $\sigma/q \approx 2^{-55}$ in Torus, so is in TFHE library.)
- Decomposition parameters for key switching $(B, \ell) = (2^4, 8)$.
- Decomposition parameters for everything else $(B, \ell) = (2^4, 7)$.

Our PDTE algorithm written in the Rust programming language using the Concrete library [CJL$^+$20]. The original TFHE library [CGGI16b] and Concrete use the same FFT library (FFTW [FJ05]) for polynomial multiplication so we do not expect the performance to change significantly if our PDTE is implemented in TFHE. Finally, we include the dependency manifest (`Cargo.lock`) to ensure reproducible results.

Our homomorphic FiLIP and grouped comparison is implemented using C++ with the FINAL library, which we instantiated with the original parameters proposed in [BIP$^+$22, Table 1], except by the decomposition base of the FINAL GSW-like ciphertexts used during the transciphering, which is set as 2^7. During the tree traversal, even the decomposition base is the same as in the original parameter set. The FINAL library also uses the same FFT library (FFTW).

7 Performance Evaluation

In this section we evaluate the performance of our design based on our prototype implementation. As we already explain in Section 1.2, only Tueno et al. [TBK20] achieve non-interactive PDTE design with practical computation and communication overhead for the client.[5] Therefore, we mainly compare our performance with them.

7.1 Computation Complexity

PDTE: SortingHat and t-SortingHat. In order to compare our performance with the work of Tueno et al. [TBK20], we first count the number of homomorphic operations (mainly multiplications)

[5] The design by Lu et al. [LZS18] also achieves non-interactive PDTE, however, the computation and communication overhead for the client render their design impractical.

of both works in Table 2. As we discussed in Section 3.1, our comparison function only consists of one multiplication up to $\log N$ bits due to our novel optimization technique for plaintext-ciphertext comparison. For general bit length of inputs, we suggest amortized comparison method when the threshold bits of decision nodes have some common subsequence bits. The homomorphic multiplication complexity for m comparison of the entire tree is $O(\frac{m\cdot\mu}{\log m})$, for input bit length μ. As we discussed in Section 5.1, we instantiated the multiplication with homomorphic AND gate which consists of $\bar{n}$ external products, where $\bar{n}$ is the dimension of an LWE secret key. Note that evaluating one external product takes $2 \times \ell$ polynomial multiplications, where $\ell = O(\log q)$, q is a ciphertext modulus. On the other hand, each comparison function used in [TBK20] consists of $\mu \log \mu$ homomorphic AND gates for any μ bit inputs.

Next, the tree evaluation step (`EvalPath` of [TBK20]) consists of a chain multiplication for every path per each leaf. Even though they mentioned their complexity of homomorphic multiplication is $O(m \log m)$, their actual algorithm consists of $2 \cdot m$ multiplication over ciphertext by reusing previously computed value. Furthermore, each multiplication is instantiated as TFHE AND gate ($2 \cdot m$ homomorphic AND gates in total) As a result, their tree evaluation requires $4 \cdot m \cdot \bar{n} \cdot \ell$. The parameters of TFHE scheme yielding 128 bits security are chosen as $\bar{n} = 630, q = 2^{32}$, and $\ell = 3$ in the original paper [CGGI20].

On the contrary, our tree evaluation `EvalTree` is instantiated in two ways depending on types of input ciphertext (thus, depending on protocols). We note that our homomorphic traversal algorithm consists of m homomorphic multiplication.

1. SortingHat) RLWE ciphertexts encrypting integer: `EvalTree` consists of two sub-algorithms: RLWEtoRGSW and HomTrav. The algorithm RLWEtoRGSW consists of $\log N$ external products for each decision node, where N is the degree of the ciphertext polynomial. The homomorphic multiplication HomTrav is instantiated with external product. Therefore, this step takes m external product, resulting in $m \cdot (\log N + 1)$ external products in total.

2. t-SortingHat) RLWE ciphertexts encrypting bit: `EvalTree` is mainly evaluating homomorphic traversal algorithm which consists of m homomorphic AND gate as discussed in Section 5.1.

Table 2: Efficiency comparison in terms of the number of polynomial multiplication for each protocol. We denote the complexity of evaluating comparison function `Comp`. μ denotes the bit length of the input. The tree evaluation is denoted by `EvalTree`. We count the number of external product for this tree evaluation.

	[TBK20]	SortingHat	t-SortingHat
`Comp`	$O(m \cdot \bar{n} \cdot \mu \cdot \log \mu \cdot \log q)$	$O(m)$	$O(\frac{m\cdot\mu}{\log m})$
`EvalTree`	$2 \cdot m \cdot \bar{n}$	$m \cdot (\log N + 1)$	$m \cdot \bar{n}$

Transciphering: FiLIP In [HMR20], the authors propose some algorithms to evaluate FiLIP's decryption homomorphically. The fastest algorithm takes GSW ciphertexts as input and outputs an LWE ciphertext. This is also the case in our Algorithm 10, except that we also need LWE encryptions of the bits. Thus, in their setup phase, the client has to send only $\mathsf{RGSW}(k_i)$ for every bit k_i of FiLIP's secret key $K \in \{0,1\}^Z$, while in our case there are additional Z/N RLWE ciphertexts. We stress that this is a small overhead, since LWE ciphertexts are very small compared to GSW ciphertexts. For example, in our proof-of-concept implementation presented in Section 7.2, the size of all RLWE ciphertexts sent during the setup phase is about 78 KB. Moreover, this setup is executed only once.

For the homomorphic decryption, [HMR20] proposes a Boolean circuit to compute the Hamming weight, perform the comparison of the threshold function, and all the needed XOR gates. Then, they use external products to compute AND gates. Since Hamming weight and comparison are not functions that can be easily expressed as circuits, even with their refined analysis, they need $(s-d)d+s-2=\Theta(sd)$ homomorphic multiplications to evaluate $\mathsf{XTHR}_{k,d,s}$, while we just need s homomorphic multiplications. We summarize this in Table 3

Table 3: Comparison of communication cost of the setup phase and number of operations needed in our homomorphic FiLIP and in the one from [HMR20].

	Setup phase		Homomorphic $\mathsf{FiLIP.dec}$	
	RGSW	RLWE	Add.	Mult.
[HMR20]	Z	0	$(s-d)(2d-1)+k$	$(s-d)d+s-2$
Ours	Z	Z/N	k	s

7.2 Experimental Results

PDTE: SortingHat. We experimentally evaluate the performance by running our protocol on real datasets from the UCI repository [DG17] and compare with prior works [TBK20,LZS18].

Datasets and Training. Many of our dataset is obtained from the UCI repository [DG17] via the OpenML [VvRBT13] service. To compare with prior work, we try to use the same dataset but not all of them were available. [6] For the ones that are not available, we generate artificial models that have the same depth and the same number of internal and leaf nodes. We also select a few more datasets and train it with a variety of constraints on the number of nodes and depths to understand our performance characteristics. The exact dataset is listed in Table 4 in the form of OpenML ID.[7] The training is performed using scikit-learn [PVG+11].

Results and Discussion. We evaluate various decision tree models using our method with a personal computer running on AMD Ryzen 5 5600X 6-Core Processor @ 3.70 GHz. In the experiment we use either one thread $\tau=1$ or six threads $\tau=6$, to evaluate multiple attribute vectors in parallel. Our results are given in Table 4. All results are amortized over the client input. Real models are used to compare with existing work ([TBK20, Table VII] and [LZS18, Table 6]) when available, otherwise we generate the model. Our results range between 20x to 90x faster when compared to [TBK20] and is roughly 50x times faster than [LZS18]. Although the CPU used in the experiment is different between our work and existing work, our single-threaded version still significantly outperforms the existing work.

We also note that the input size differs depending on protocol in Table 4. In our case, we use 11 bits integers as input since our parameter N is 2^{11}, however [TBK20] uses 16 bits, and [LZS18] uses 12 bits. The dataset that we worked on has actually small input size, which 11 bits are more than enough. Moreover, if it is necessary to increase input size upto 22 bits, one can always build a bigger tree by splitting one feature into two of 11 bits. Or one can use the general extension of $\mathsf{PolyComp}$ which gives the same computation time for input size from 12 bits to 22 bits with some computation overhead, which is discussed in Appendix A.

[6] The "Housing" dataset, used by both [TBK20] and [LZS18], does not exist in the UCI repository.
[7] Accessing the dataset can be done via, e.g., https://www.openml.org/search?type=data&id=31 for ID 31.

Table 4: PDTE results for various datasets. The model attribute description can be found in Section 2.3. Dataset marked with asterisk (*) are generated artificially. ID is the OpenML ID given for reproducibility purposes. The time is given in milliseconds. τ is the number of threads.

					[TBK20]	[LZS18]	SortingHat	
dataset	ID	d	m	n	$\tau = 16$	$\tau = 16$	$\tau = 1$	$\tau = 6$
heart	1565	3	5	13	940	590	42.3	10.5
breast	1510	7	17	30	-	-	154	34.8
steel	1504	5	6	33	-	-	51.9	12.3
housing*	N/A	13	92	13	6300	10270	892	190
spam	44	16	58	57	3660	6880	553	115
artificial*	N/A	10	500	16	22390	56370	4787	1045

Tueno et al. [TBK20] showed an alternative approach to speed up using SIMD slots [SV11] that uses BGV [BGV12]. The authors use between 600 and 6198 plaintext slots in one ciphertext during evaluation. In other words, hundreds or thousands of attribute vectors are evaluated at once. The amortized efficiency gains from this approach is significant and can perform better than our design, however, it is only applicable when the client can batch such large number of queries. If an individual wishes to perform secure decision tree evaluation using his or her health data, for example, then there would be only one attribute vector. Similarly, on-device spam detection also cannot batch hundreds of email or text messages due to the latency requirement. Furthermore, the batching technique only applies to attribute vectors using the same key. Whereas our multi-threaded approach does not have this constraint.

Overhead for the Client. In our design (and also for [TBK20]), the client only decrypts one ciphertext to retrieve the result. This operation consists of one polynomial multiplication that takes only a few milliseconds. The design by Lu et al. [LZS18] requires 320 milliseconds for the client to decrypt the result for the "heart" dataset, and 3.35 seconds for the "spam" dataset.

Transciphering: FiLIP Our proof-of-concept implementation of homomorphic FiLIP uses the FHE scheme FINAL [BIP+22] instead of TFHE [CGGI20]. Thus, each GSW ciphertext in our protocol is around 12 KB long, while GSW ciphertexts in [HMR20] require 98 KB each. The authors present a few different instantiations of homomorphic FiLIP and we compare our running times with their most efficient version using a personal computer with a AMD Ryzen 5 5600X 6-Core Processor. Our running times are nearly 400 times faster than theirs. We summarize these results in Table 5. It is not clear if the prior work used multiple threads, hence, to be conservative, we are assuming they used only one thread and we also run our experiments on a single thread. Similarly, these results do not rule out influences of the difference in CPU. Nevertheless, we still expect our results to be two orders of magnitude faster when hardware differences are accounted.

Table 5: Comparison between the communication cost of the setup phase and the running times of the homomorphic FiLIP decryption.

	[HMR20]	Ours	Improvement
Setup	800 MB	200 MB	$\times 4$
Timing	1018 ms	2.62 ms	$\times 388$

Grouped Comparison. In Table 6 we present our results of PDTE evaluation when the transciphering is used. We ran the experiments on the same machine as before (AMD Ryzen 5 5600X 6-Core Processor). These results only show the homomorphic comparisons executed in each decision node and the tree traversal step, i.e., not including transciphering. But the total timing can be easily derived from Table 5. For example, the artificial dataset has 16 features which are 16 bits each, then transciphering would take an additional 670 ms on top of the results in Table 6, which is marginal. The key observation we make is that when the number of nodes are much higher than the number of features, e.g., in the case of the artificial dataset, then our comparison technique start to outperform the naive approach. Notice that previous works used several threads to execute their experiments, while we are using a single thread. We expect our results to improve significantly with a parallel implementation.

Considering the communication cost of [TBK20], to run the classification on an input with n attributes, each one with μ bits, the client has to send $n \cdot \mu$ ciphertexts, and each ciphertext has more than 20 thousand bits. In our case, thanks to the transciphering, the client just has to encrypt the input with the stream cipher FiLIP, thus, there is no data expansion. In addition, the client has to send an IV with λ bits. Thus, our communication cost is simply $n \cdot \mu + \lambda$ bits. Therefore, the client has to upload around $2 \cdot 10^4$ times less data with our solution.

Table 6: Performance results when using grouped comparison on a single thread for attribute vectors with 16 bits. The timing is listed in seconds. The details of the dataset can be found on Table 4. Notice that the communication cost is about $2 \cdot 10^4$ times lower in our solution, due to the use of transciphering.

dataset	[TBK20] ($\tau = 16$)	Naive ($\tau = 1$)	Recursive ($\tau = 1$)
heart	0.94	1.51	1.52
housing*	6.3	30.18	28.60
spam	3.66	20.3	21.49
artificial*	22.39	145.9	92.44

8 Conclusion and Future Work

In this paper, we have presented a new non-interactive design SortingHat for the private decision tree evaluation problem. SortingHat has low computation (few milliseconds) and communication overhead for the client, and because of our efficient homomorphic comparison and homomorphic traversal techniques the computation cost for the server is 20–90 times lower than existing solutions — and these optimizations make SortingHat suitable for practical use. Moreover, we provided a solution in the form of t-SortingHat to further reduce the communication overhead by reducing the ciphertext expansion drastically using a computationally efficient transciphering method. Unfortunately, the transciphering method is not compatible with our fast homomorphic comparison function. An important future work would be design a computationally efficient transciphering technique that is compatible with our fast comparison function.

Acknowledgement

This work is partially supported by the Research Council KU Leuven under the grant C24/18/049, CyberSecurity Research Flanders with reference number VR20192203, Defense Advanced Research

Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract numbers FA8750-19-C-0502 and HR0011-21-C-0034.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the funders. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

ACLS18. Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.

AEM13. Ahmad Azar and Shereen El-Metwally. Decision tree classifiers for automated medical diagnosis. *Neural Computing and Applications*, 23:2387–2403, 11 2013.

BFK$^+$09. Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Secure evaluation of private linear branching programs with medical applications. In Michael Backes and Peng Ning, editors, *ESORICS 2009*, volume 5789 of *LNCS*, pages 424–439. Springer, Heidelberg, September 2009.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

BHKR13. Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

BHR12. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

BIP$^+$22. Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster fhe instantiated with ntru and lwe. Cryptology ePrint Archive, Report 2022/074, 2022. https://ia.cr/2022/074.

BPSW07. Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 498–507. ACM Press, October 2007.

BPTG15. Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS 2015*. The Internet Society, February 2015.

CCR19. Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 345–360. ACM Press, November 2019.

CGGI16a. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2016.

CGGI16b. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

CGGI20. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

CJL$^+$20. Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Concrete: Concrete operates on ciphertexts rapidly by extending tfhe. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, volume 15, 2020.

CKK20. Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 221–256. Springer, Heidelberg, December 2020.

CRRV17. Ran Canetti, Srinivasan Raghuraman, Silas Richelson, and Vinod Vaikuntanathan. Chosen-ciphertext secure fully homomorphic encryption. In Serge Fehr, editor, *PKC 2017, Part II*, volume 10175 of *LNCS*, pages 213–240. Springer, Heidelberg, March 2017.

CS16. Ana Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 325–340. Springer, Heidelberg, February / March 2016.

DG17. Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

DM15. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 617–640. Springer, Heidelberg, April 2015.

FJ05. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

FPS02. Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano. Improving induction decision trees with parallel genetic programming. *Proceedings 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 181–187, 2002.

GHS12. Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 465–482. Springer, Heidelberg, April 2012.

HMR20. Clément Hoffmann, Pierrick Méaux, and Thomas Ricosset. Transciphering, using FiLIP and TFHE for an efficient delegation of computation. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *INDOCRYPT 2020*, volume 12578 of *LNCS*, pages 39–61. Springer, Heidelberg, December 2020.

HV16. Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. On the power of secure two-party computation. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 397–429. Springer, Heidelberg, August 2016.

IZ21. Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for bgv and bfv. *Proceedings on Privacy Enhancing Technologies*, 2021(3):246–264, 2021.

KO04. Jonathan Katz and Rafail Ostrovsky. Round-optimal secure two-party computation. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 335–354. Springer, Heidelberg, August 2004.

KRRW18. Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 365–391. Springer, Heidelberg, August 2018.

KTG06. Hian Chye Koh, Wei Chin. Tan, and Chwee Peng Goh. A two-step method to construct credit scoring models with data mining techniques. *The International Journal of Business and Information*, 1, 2006.

LMSV12. Jake Loftus, Alexander May, Nigel P. Smart, and Frederik Vercauteren. On CCA-secure somewhat homomorphic encryption. In Ali Miri and Serge Vaudenay, editors, *SAC 2011*, volume 7118 of *LNCS*, pages 55–72. Springer, Heidelberg, August 2012.

LZS18. Wenjie Lu, Jun-Jie Zhou, and Jun Sakuma. Non-interactive and output expressive private comparison from homomorphic encryption. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 67–74. ACM Press, April 2018.

MCJS19. Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. Improved filter permutators for efficient FHE: Better instances and implementations. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 68–91. Springer, Heidelberg, December 2019.

Per21. Hilder Vitor Lima Pereira. Bootstrapping fully homomorphic encryption over the integers in less than one second. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 331–359. Springer, Heidelberg, May 2021.

PT20. Jeongeun Park and Mehdi Tibouchi. SHECS-PIR: Somewhat homomorphic encryption-based compact and scalable private information retrieval. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 86–106. Springer, Heidelberg, September 2020.

PVG$^+$11. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Qui86. J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

RM05. Lior Rokach and Oded Maimon. Top–down induction of decision trees classifiers–a survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, page 487, 2005.

RM14. Lior Rokach and Oded Maimon. *Data Mining with Decision Trees.* WORLD SCIENTIFIC, 2nd edition, 2014.

SG11. Anima Singh and John V. Guttag. A comparison of non-symmetric entropy-based classification trees and support vector machine for cardiovascular risk stratification. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 79–82, 2011.

SV11. N.P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Cryptology ePrint Archive, Report 2011/133, 2011. https://eprint.iacr.org/2011/133.

TBK20. Anselme Tueno, Yordan Boev, and Florian Kerschbaum. Non-interactive private decision tree evaluation. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 174–194. Springer, 2020.

TKK19. 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.

TMZC17. Raymond K. H. Tai, Jack P. K. Ma, Yongjun Zhao, and Sherman S. M. Chow. Privacy-preserving decision trees evaluation via linear functions. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 494–512. Springer, Heidelberg, September 2017.

vGHV10. Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 24–43. Springer, Heidelberg, May / June 2010.

VvRBT13. Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

WFH11. Ian H Witten, Eibe Frank, and Mark A Hall. Data mining: Practical machine learning tools and techniques, 2011.

WFNL16. David J. Wu, Tony Feng, Michael Naehrig, and Kristin E. Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016(4):335–355, October 2016.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

ZS21. Martin Zuber and Renaud Sirdey. Efficient homomorphic evaluation of k-nn classifiers. *Proceedings on Privacy Enhancing Technologies*, 2021:111 – 129, 2021.

A General Extension of PolyComp for Arbitrary Number of Bits.

An extension of our comparison algorithm (Section 3.1) for inputs of size larger than $\log N$ is discussed below. First, we decompose the two inputs of the comparison, say A and B, with the base 2^N, i.e., $A = 2^{kN} \cdot A_k + \cdots + 2^0 \cdot A_0$ and $B = 2^{kN} \cdot B_k + \cdots + 2^0$, where A_i's and B_i's are $\log N$ bit integers $\forall i \in \{0, \ldots, k\}$ and $k \in \mathbb{Z}_{\geq 0}$. Suppose we have an efficient equality check algorithm that outputs $y_i = 1$ if $A_i = B_i$, $y_i = 0$ otherwise; and a "greater than" algorithm that outputs $x_i = 1$ if $A_i > B_i$, $x_i = 0$ otherwise. We define our comparison function as follows:

$$\mathsf{PolyComp}_{A,B \geq N} := x_k + y_k \cdot x_{k-1} + y_k \cdot y_{k-1} \cdot x_{k-2} + y_k \cdot y_{k-1} \cdot y_{k-2} \cdot x_{k-3} + \cdots \Pi_{i=1}^{k}(y_i) \cdot x_0$$

Note that if $x_i = 1$, $y_i = 0$; and if $y_i = 1$, $x_i = 0$. We start from the most significant digits, A_k and B_k, gradually proceeding towards lower significant digits until an inequality is found, i.e., $y_j = 0$ for some j and all $y_i = 1$ for $j < i \leq k$. If also $x_j = 1$, we conclude that $A > B$; otherwise $A < B$.

Now, it is time to build efficient "greater than" and "equal to" function. All we need to do is modify the test vector $T(X)$ of Algorithm 1. For "equal to" function, we set $T(X) = X^{2N-B_i}$; for "greater than", $T(X)$ is set as $X^{(2N-N)} + \cdots + X^{2N-(B_i+1)}$.

The downside of this technique, however, is that instantiating those multiplications between y_i's and x_j's homomorphically is not straightforward. The best methods are either converting the output of the above sub-algorithms (for less than $\log N$ bits) to RGSW ciphertexts then do external product between them, or evaluate bootstrapping. If k is small, nevertheless, either way is suitable since the complexity in terms of homomorphic multiplications is $O(\frac{k \cdot \mu}{N})$, where μ is the bit length of the inputs A and B.