Non-Interactive Private Decision Tree Evaluation

[PRE-PRINT]

Anselme Tueno, Yordan Boev, Florian Kerschbaum

ABSTRACT: In this paper, we address the problem of privately evaluating a decision tree on private data. This scenario consists of a server holding a private decision tree model and a client interested in classifying its private attribute vector using the server's private model. The goal of the computation is to obtain the classification while preserving the privacy of both – the decision tree and the client input. After the computation, the client learns the classification result and nothing else, and the server learns nothing. Existing privacy-preserving protocols that address this problem use or combine different generic secure multiparty computation approaches resulting in several interactions between the client and the server. Our goal is to design and implement a novel client-server protocol that delegates the complete tree evaluation to the server while preserving privacy and reducing the overhead. The idea is to use fully (somewhat) homomorphic encryption and evaluate the tree on ciphertexts encrypted under the client's public key. However, since current somewhat homomorphic encryption schemes have high overhead, we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low. As a result, we are able to provide the first non-interactive protocol, that allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result. Our scheme has only one round and evaluates a complete tree of depth 10 within seconds.

Citing this paper

This is a pre-print of the paper that appears in the proceedings of the Data and Applications Security and Privacy XXXIV . https://doi.org/10.1007/978-3-030-49669-2

Bibtex for citation (open BibTex file)

```
@inproceedings{DBLP:conf/dbsec/TuenoBK20,
  author    = {Anselme Tueno and
               Yordan Boev and
               Florian Kerschbaum},
  title     = {Non-interactive Private Decision Tree Evaluation},
  booktitle = {Data and Applications Security and Privacy {XXXIV} - 34th Annual {IFIP}
               {WG} 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25-26,
               2020, Proceedings},
  pages     = {174--194},
  year      = {2020},
  crossref  = {DBLP:conf/dbsec/2020},
  url       = {https://doi.org/10.1007/978-3-030-49669-2\_10},
  doi       = {10.1007/978-3-030-49669-2\_10},
  timestamp = {Thu, 18 Jun 2020 13:47:47 +0200},
  biburl    = {https://dblp.org/rec/conf/dbsec/TuenoBK20.bib},
  bibsource = {dblp computer science bibliography, https://dblp.org}
}
```

Non-Interactive Private Decision Tree Evaluation

Anselme Tueno[1(✉)], Yordan Boev[1], and Florian Kerschbaum[2]

[1] SAP Security Research, Karlsruhe, Germany
anselme.kemgne.tueno@sap.com
[2] University of Waterloo, Canada

Abstract In this paper, we address the problem of privately evaluating a decision tree on private data. This scenario consists of a server holding a private decision tree model and a client interested in classifying its private attribute vector using the server's private model. The goal of the computation is to obtain the classification while preserving the privacy of both – the decision tree and the client input. After the computation, the client learns the classification result and nothing else, and the server learns nothing. Existing privacy-preserving protocols that address this problem use or combine different generic secure multiparty computation approaches resulting in several interactions between the client and the server. Our goal is to design and implement a novel client-server protocol that delegates the complete tree evaluation to the server while preserving privacy and reducing the overhead. The idea is to use fully (somewhat) homomorphic encryption and evaluate the tree on ciphertexts encrypted under the client's public key. However, since current somewhat homomorphic encryption schemes have high overhead, we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low. As a result, we are able to provide the first non-interactive protocol, that allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result. Our scheme has only one round and evaluates a complete tree of depth 10 within seconds.

Keywords: Machine Learning, Private Decision Tree Evaluation, Secure Multiparty Computation, Fully/Somewhat Homomorphic Encryption

1 Introduction

Machine learning (ML) classifiers are valuable tools in many areas such as healthcare, finance, spam filtering, intrusion detection, remote diagnosis, etc. [37]. To perform their task, these classifiers often require access to personal sensitive data such as medical or financial records. Therefore, it is crucial to investigate technologies that preserve the privacy of the data, while benefiting from the advantages of ML. On the one hand, the ML model itself may contain sensitive data. On the other hand, the model may have been built on sensitive data. It is known that white-box and sometimes even black-box access to a ML model allows so-called *model inversion attacks* [18, 33, 39], which can compromise the privacy of the training data. As a result, making the ML model public could violate the privacy of the training data.

In this paper, we therefore address the problem of private decision tree evaluation (PDTE) on private data. This scenario consists of a server holding a private decision tree model and a client wanting to classify its private attribute vector using the server's private model. The goal of the computation is to obtain the classification while preserving the privacy of both – the decision tree and the client input. After the computation, the classification result is revealed only to the client, and beyond that, nothing further is revealed. The problem can be solved using any generic secure multiparty computation.

Generic secure multiparty computation [16, 20] can implement PDTE. There exist frameworks such as ObliVM [27] or CBMC-GC [17], HyCC [8] that are able to automate the transformation of the plaintext decision tree program, written in a high level programming language, into oblivious programs suitable for secure computation. Their straightforward application to decision tree programs does certainly improve performance. However, the size of the resulting oblivious program is still proportional to the size of the tree. As a result generic solution are in general inefficient, in particular when the size of the tree is large.

Specialized protocols [2, 4, 7, 23, 24, 32, 34, 38] exploit the domain knowledge of the problem at hand and make use of generic techniques only where it is necessary, resulting in more efficient solutions. Existing protocols for PDTE have several rounds requiring several interactions between the client and the server. Moreover, the communication cost depends on the size of the decision tree, while only a single classification is required by the client. Finally, they also require computational power from the client that depends on the size of the tree.

Our goal is to design and implement a novel client-server protocol that delegates the complete tree evaluation to the server while preserving privacy and keeping the performance acceptable. The idea is to use fully or somewhat homomorphic encryption (FHE/SHE) and evaluate the tree on ciphertexts encrypted under the client's public key. As a result, no intermediate or final computational result is revealed to the evaluating server. However, since current SHE/FHE schemes have high overhead, we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low. At the end, the computational overhead might still be higher than in existing protocols, however the computation task can be parallelized resulting in a reduced computation time. As a result, we are able to provide the first non-interactive protocol, that allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result. Finally, existing approaches are secure in the semi-honest model (i.e., parties follow the protocol) and can be made *one-sided simulatable* using techniques that may double the computation and communication costs. A one-sided simulatable protocol forces the client to behave semi-honestly and guarantees that a malicious server learns nothing [22]. Our approach is one-sided simulatable by default, as the client does no more than encrypting its input and decrypting the final result of the computation (simulating the client is straightforward), while the server evaluates on ciphertexts encrypted with a semantically secure encryption under the client's public key.

Concrete motivation of our approach are machine learning settings (with applications in areas such as healthcare, finance etc.) where the server is computationally powerful, the client is computationally weak and the network connection is not very fast. Our contributions are as follows:

- We propose a non-interactive protocol for PDTE. Our scheme allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result.
- We propose Pdt-Bin which is an instantiation of the main protocol with binary representation of the input. We combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low.
- Finally, we implement and benchmark using HElib [21] and TFHE [11].

The remainder of the paper is structured as follows. We review preliminaries in Section 2 before defining correctness and security of our protocol in Section 3. The basic construction itself is described in Section 4. In Section 5, we describe implementation and optimization using a binary representation. We discuss implementation and evaluation details in Section 6. We review related work in Section 7 before concluding our work in Section 8. Due to space constraints, we discuss further details in the appendix.

2 Preliminaries

For ease of exposition, we abstract away the mathematical technicalities behind HE and refer to the literature [1,5,10,19,31]. We start with the notation.

Notation. In this paper, μ denotes the bit length of attribute values, n denotes the dimension of the attribute vector, m denotes the number of decision nodes, d denotes the depth of the decision tree.

Homomorphic Encryption. We focus on (lattice-based) homomorphic encryption (HE) schemes that allow many chained additions and multiplications to be computed on plaintext. In these schemes, the plaintext space is usually a ring $\mathbb{Z}_q[X]/(X^N+1)$, where q is prime and N might be a power of 2.

A HE consists of the usual algorithms for key generation $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KGen}(\lambda)$, encryption $\mathsf{Enc}(\mathsf{pk},\mathsf{m})$ (we denote $\mathsf{Enc}(\mathsf{pk},\mathsf{m})$ by $[\![\mathsf{m}]\!]$), decryption $\mathsf{Dec}(\mathsf{sk},\mathsf{c})$. HE has an additional evaluation algorithm $\mathsf{Eval}(\mathsf{pk},f,\mathsf{c}_1,\ldots,\mathsf{c}_n)$ that takes pk, an n-ary function f and ciphertexts $\mathsf{c}_1,\ldots\mathsf{c}_n$. It outputs a ciphertext c such that if $\mathsf{c}_i = [\![\mathsf{m}_i]\!]$ then it holds: $\mathsf{Dec}(\mathsf{sk},\mathsf{Eval}(f,[\![\mathsf{m}_1]\!],\ldots,[\![\mathsf{m}_n]\!])) = \mathsf{Dec}(\mathsf{sk},[\![f(\mathsf{m}_1,\ldots,\mathsf{m}_n)]\!])$. We require HE to be IND-CPA secure.

The encryption algorithm Enc adds 'noise' to the ciphertext which increases during homomorphic evaluation. While addition of ciphertexts increases the noise slightly, the multiplication increases it significantly [5]. If the noise becomes too large then correct decryption is no longer possible. To prevent this from happening, one can either keep the circuit's depth of the function f low enough or use *bootstrapping* procedure which reduces the noise in a ciphertext, i.e., fully HE (FHE). In this paper, we will consider both bootstrapping and the possibility of keeping the circuit's depth low by designing our PDTE using so-called leveled FHE. A leveled FHE has an extra parameter L such that the scheme can evaluate all circuits of depth at most L without bootstrapping.

Homomorphic Operations. We assume a BGV type HE scheme [5]. Plaintexts can be encrypted using an integer representation (an integer x_i is encrypted as $[\![x_i]\!]$) or a binary representation (each bit of the bit representation $x_i^{\mathsf{b}} = x_{i\mu} \dots x_{i1}$ is encrypted). We describe below HE operations in the binary representation (i.e., arithmetic operations $\bmod 2$). They work similarly in the integer representation.

The FHE scheme might support Smart and Vercauteren's ciphertext packing (SVCP) technique [31] to pack many plaintexts in one ciphertext. Using SVCP, a ciphertext consists of a fixed number s of slots, each capable of holding one plaintext, i.e. $[\![\cdot|\cdot|\dots|\cdot]\!]$. The encryption of a bit b replicates b to all slots, i.e., $[\![b]\!] = [\![b|b|\dots|b]\!]$. However, we can also pack the bits of x_i^{b} in one ciphertext and will denote it by $[\![\vec{x}_i]\!] = [\![x_{i\mu}|\dots|x_{i1}|0|\dots|0]\!]$.

The computation relies on some built-in routines, that allow homomorphic operations on encrypted data. The relevant routines for our scheme are: addition (SheAdd), multiplication (SheMult) and comparison (SheCmp). These routines are compatible with the ciphertext packing technique (i.e., operations are replicated on all slots in a SIMD manner).

The routine SheAdd performs a component-wise addition modulo two, i.e., we have: $\textsc{SheAdd}([\![b_{i1}|\dots|b_{is}]\!], [\![b_{j1}|\dots|b_{js}]\!]) = [\![b_{i1} \oplus b_{j1}|\dots|b_{is} \oplus b_{js}]\!]$. Similarly, SheMult performs component-wise multiplication modulo two, i.e., we have: $\textsc{SheMult}([\![b_{i1}|\dots|b_{is}]\!], [\![b_{j1}|\dots|b_{js}]\!]) = [\![b_{i1} \cdot b_{j1}|\dots|b_{is} \cdot b_{js}]\!]$. We will denote addition and multiplication by $\boxplus$ and $\boxdot$, respectively.

Let x_i, x_j be two integers, $b_{ij} = [x_i > x_j]$, $b_{ji} = [x_j > x_i]$, the routine SheCmp takes $[\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!]$, compares x_i and x_j and returns $[\![b_{ij}]\!]$, $[\![b_{ji}]\!]$: $([\![b_{ij}]\!], [\![b_{ji}]\!]) \leftarrow \textsc{SheCmp}([\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!])$. Note that, if the inputs to SheCmp encrypt the same value, then the routine outputs two ciphertexts of 0. This routine implements the comparison circuit described in [9].

If ciphertext packing is enabled, then we also assume that HE supports shift operations. Given a packed ciphertext $[\![b_1|\dots|b_s]\!]$, the *shift left* operation shifts all slots to the left by a given offset, using zero-fill, i.e., shifting $[\![b_1|\dots|b_s]\!]$ by i positions returns $[\![b_i|\dots|b_s|0|\dots|0]\!]$.

3 Definitions

This section introduces relevant definitions. With $[a, b]$, we denote the set of all integers from a to b. Let $c_0, \dots, c_{k-1}$ be the classification labels, $k \in \mathbb{N}_{>0}$.

Definition 1 (Decision Tree). *A* decision tree *(DT) is a function* $\mathcal{T} : \mathbb{Z}^n \rightarrow \{c_0, \dots, c_{k-1}\}$ *that maps an* attribute vector $x = (x_0, \dots, x_{n-1})$ *to a finite set of* classification labels. *A DT consists of* internal *or* decision nodes *containing a test condition, and* leaf nodes *containing a classification label. A* decision tree model *consists of a DT and the following functions: a function* thr *that assigns to each decision node a* threshold *value,* $\mathsf{thr} : [0, m-1] \to \mathbb{Z}$*; a function* att *that assigns to each decision node an* attribute index, $\mathsf{att} : [0, m-1] \to [0, n-1]$, *and; a labeling function* lab *that assigns to each leaf node a label,* $\mathsf{lab} : [m, M-1] \to \{c_0, \dots, c_{k-1}\}$*. The decision at each decision node is a 'greater-than' comparison between the assigned threshold and attribute values, i.e., the decision at node v is* $[x_{\mathsf{att}(v)} \geq \mathsf{thr}(v)]$*. We use* $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$ *to denote a* decision tree model.

Definition 2 (Decision Tree Evaluation). *Given an attribute vector* $x = (x_0, \dots, x_{n-1})$ *and* $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$*, then starting at the root, the* Decision Tree Evaluation *(DTE) evaluates at each reached node v the decision* $b \leftarrow [x_{\mathsf{att}(v)} \geq \mathsf{thr}(v)]$ *and moves either to the left (if $b = 0$) or right (if $b = 1$) subsequent node. The evaluation returns the label of the reached leaf as result of the computation. We denote this by* $\mathcal{T}(x)$*.*

Definition 3 (Private DTE). *Given a* client *with a* private $x = (x_0, \dots, x_{n-1})$ *and a* server *with a* private $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$*, a* private DTE (PDTE) *functionality evaluates the model $\mathcal{M}$ on input x, then reveals to the client the* classification label $\mathcal{T}(x)$ *and nothing else, while the server learns* nothing*, i.e.,* $\mathcal{F}_{\mathsf{PDTE}}(\mathcal{M}, x) \to (\varepsilon, \mathcal{T}(x))$*.*

Definition 4 (Correctness). *Given a* client *with a* private $x = (x_0, \dots, x_{n-1})$ *and a* server *with a* private $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$*, a protocol Π* correctly implements *a PDTE functionality if after the computation it holds for the result c obtained by the client that* $c = \mathcal{T}(x)$*.*

Two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are *computationally indistinguishable* (denoted $\mathcal{D}_1 \overset{c}{\equiv} \mathcal{D}_2$) if no probabilistic polynomial time (PPT) algorithm can distinguish them except with negligible probability. In SMC protocols, the *view* of a party consists of its input and the sequence of messages that it has received during the protocol execution [20].

Definition 5 (PDTE Security). *Given a* client C *with a* private *input* $x = (x_0, \dots, x_{n-1})$ *and a* server S *with a* private *model* $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$*, a protocol* Π_{PDTE} securely implements *the PDTE functionality in the* semi-honest model *if the following holds: there exists a PPT algorithm* $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{pdte}}$ *that simulates the server's view* $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{PDTE}}}$ *given only* $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$ *, i.e.,* $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{pdte}}(\mathcal{M}, \varepsilon) \overset{c}{\equiv} \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{PDTE}}}(\mathcal{M}, x)$*; there exists a PPT algorithm* $\mathsf{Sim}_{\mathsf{C}}^{\mathsf{pdte}}$ *that simulates the client's view* $\mathsf{View}_{\mathsf{C}}^{\Pi_{\mathsf{PDTE}}}$ *given only x and $\mathcal{T}(x)$, i.e.,* $\mathsf{Sim}_{\mathsf{C}}^{\mathsf{pdte}}(x, \mathcal{T}(x)) \overset{c}{\equiv} \mathsf{View}_{\mathsf{C}}^{\Pi_{\mathsf{PDTE}}}(\mathcal{M}, x)$*.*

A protocol Π_{PDTE} securely implements *the PDTE functionality with* one-sided simulation *if the following conditions hold: For every pair x, x' of different client's inputs, it holds* $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{PDTE}}}(\mathcal{M}, x) \overset{c}{\equiv} \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{PDTE}}}(\mathcal{M}, x')$*; and* Π_{PDTE} *is simulatable against every PPT adversary controlling C.*

4 The Basic Protocol

We present a modular description of our protocol starting by the data structure.

4.1 Data Structure

We follow the idea of previous protocols [4, 12, 32] of marking edges of the tree with comparison result. So if the comparison at node v is the bit b then we mark the right edge to v with b and the left edge with $1 - b$. For convenience, we will instead store this information at the child nodes of v and refer to it as cmp.

Definition 6 (Data Structure). *For a decision tree model* $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$, *we let* Node *be a data structure that for each node* v *defines the following fields:* $v.\mathsf{threshold}$ *stores the threshold* $\mathsf{thr}(v)$ *of* v*;* $v.\mathsf{aIndex}$ *stores the associated index* $\mathsf{att}(v)$*;* $v.\mathsf{parent}$ *stores the pointer to the parent node (null for the root);* $v.\mathsf{left}$ *stores the pointer to the left child node (null for each leaf);* $v.\mathsf{right}$ *stores the pointer to the right child node (null for each leaf);* $v.\mathsf{cmp}$ *is computed during the tree evaluation and stores the comparison bit* $b \leftarrow [x_{\mathsf{att}(v.\mathsf{parent})} \geq \mathsf{thr}(v.\mathsf{parent})]$ *if* v *is a right node. Otherwise it stores* $1 - b$*;* $v.\mathsf{cLabel}$ *stores the classification label if* v *is a leaf node and the empty string otherwise. We use* $\mathcal{D}$ *to denote the set of all decision nodes and* $\mathcal{L}$ *the set of all leaf nodes of* $\mathcal{M}$*. As a result, we use the equivalent notation* $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att}) = (\mathcal{D}, \mathcal{L})$.

With the data structure defined above, we now define the classification function.

Definition 7 (Classification Function). *Let* $x = (x_0, \ldots, x_{n-1})$ *be the attribute vector and* $\mathcal{M} = (\mathcal{D}, \mathcal{L})$ *be the DT model. We define the* classification function *to be* $\mathsf{f}_c(x, \mathcal{M}) = \mathsf{tr}(x, \mathsf{root})$, *where* root *is the root node and* tr *is the* traverse function *define as:*

$$\mathsf{tr}(x, v) = \begin{cases} \mathsf{tr}(x, v.\mathsf{left}) & \text{if } v \in \mathcal{D} \text{ and } x_{v.\mathsf{aIndex}} < v.\mathsf{threshold} \\ \mathsf{tr}(x, v.\mathsf{right}) & \text{if } v \in \mathcal{D} \text{ and } x_{v.\mathsf{aIndex}} \geq v.\mathsf{threshold} \\ v & \text{if } v \in \mathcal{L} \end{cases}$$

Lemma 1. *Let* $x = (x_0, \ldots, x_{n-1})$ *be an attribute vector and* $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att}) = (\mathcal{D}, \mathcal{L})$ *a DT model. We have* $\mathcal{T}(x) = b \cdot \mathsf{tr}(x, \mathsf{root.right}) + (1 - b) \cdot \mathsf{tr}(x, \mathsf{root.left})$, *where* $b = [x_{\mathsf{att}(\mathsf{root})} \geq \mathsf{thr}(\mathsf{root})]$ *is the comparison at the root node.*

Proof. The proof follows by induction on the depth of the tree. In the base case, we have a tree of depth one (i.e., the root and two leaves). In the induction step, we have two trees of depth d and we joint them by adding a new root.

4.2 Algorithms

Initialization. The Initialization consists of a one-time key generation. The client generates appropriate pair $(\mathsf{pk}, \mathsf{sk})$ of public and private keys for a HE scheme, and sends pk to the server. For each input classification, the client just

```
function EvalPaths($\mathcal{D}, \mathcal{L}$)
  let $Q$ be a queue
  $Q$.enqueue(root)
  while $Q$.empty() = false do
    $v \leftarrow Q$.dequeue()
    $[\![v.\mathsf{left.cmp}]\!] \leftarrow [\![v.\mathsf{left.cmp}]\!] \boxdot [\![v.\mathsf{cmp}]\!]$
    $[\![v.\mathsf{right.cmp}]\!] \leftarrow [\![v.\mathsf{right.cmp}]\!] \boxdot [\![v.\mathsf{cmp}]\!]$
    if $v.\mathsf{left} \in \mathcal{D}$ then
      $Q$.enqueue($v$.left)
    if $v.\mathsf{right} \in \mathcal{D}$ then
      $Q$.enqueue($v$.right)
```

Algorithm 1: Aggregating Decision Bits

encrypts its input and sends it to the server. The size of homomorphic ciphertexts are in general very large. To reduce the communication cost of sending client's input, one can use a trusted randomizer that does not take part in the real protocol and is not allowed to collaborate with the server. The trusted randomizer generates a list of random strings r and sends the encrypted strings $[\![r]\!]$ to server and the list of r's to the client. For an input x, the client then sends $x + r$ to the server in the real protocol. This technique is similar to the commodity based cryptography [3] with the difference that the client can play the role of the randomizer itself and sends the list of $[\![r]\!]$'s (when the network is not too busy) before the protocol's start.

Computing Decision Bits. The server starts by computing for each node $v \in \mathcal{D}$ the comparison bit $b \leftarrow [x_{\mathsf{att}(v)} \geq \mathsf{thr}(v)]$ and stores b at the right child node ($v.\mathsf{right.cmp} = b$) and $1 - b$ at the left child node ($v.\mathsf{left.cmp} = 1 - b$). We refer to this algorithm as EvalDnode($\mathcal{D}, [\![x]\!]$).

Aggregating Decision Bits. Then for each leaf node v, the server aggregates the comparison bits along the path from the root to v. We implement it using a queue and traversing the tree in BFS as illustrated in Algorithm 1.

Finalizing. After Aggregating the decision bits along the path to the leaf nodes, each leaf node v stores either $v.\mathsf{cmp} = 0$ or $v.\mathsf{cmp} = 1$. Then, the server aggregates the decision bits at the leaves by computing for each leaf v the value $[\![v.\mathsf{cmp}]\!] \boxdot [\![v.\mathsf{cLabel}]\!]$ and summing all the results. We refer to this algorithm as Finalize($\mathcal{L}$).

Putting It All Together. As illustrated in Protocol 2, the whole computation is performed by the server. It sequentially computes the algorithms described above and sends the resulting ciphertext to the client. The client decrypts and outputs the classification label. The correctness follows from Lemma 1.

5 Binary Implementation

In this section, we describe Pdt-Bin, an instantiation of the basic scheme that requires encoding the plaintexts using their bit representation. Hence, ciphertexts encrypt bits and arithmetic operations are done mod 2.

5.1 Input Encoding

We encrypt plaintext bitwise. For each plaintext x_i with bit representation $x_i^{\mathsf{b}} = x_{i\mu} \dots x_{i1}$, we use $[\![x_i^{\mathsf{b}}]\!]$ to denote the vector $([\![x_{i\mu}]\!], \dots, [\![x_{i1}]\!])$, consisting

Client		**Server**
Input: x		**Input:** $\mathcal{M} = (\mathcal{D}, \mathcal{L})$
Output: $\mathcal{T}(x)$		**Output:** ε
	$\xrightarrow{\llbracket x \rrbracket}$	
		$\textsc{EvalDnode}(\mathcal{D}, \llbracket x \rrbracket)$ $\textsc{EvalPaths}(\mathcal{D}, \mathcal{L})$ $\llbracket \mathcal{T}(x) \rrbracket \leftarrow \textsc{Finalize}(\mathcal{L})$
	$\xleftarrow{\llbracket \mathcal{T}(x) \rrbracket}$	

Protocol 2: The Basic Protocol

of encryptions of the bits of x_i. As a result, the client needs to send $n\mu$ ciphertexts for the n attribute values. Unfortunately, homomorphic ciphertexts might be quite large. We can already use the trusted randomizer as explained before to send blinded inputs instead of ciphertexts in this phase. This, however, improves only the online communication. We additionally want to use the SVCP SIMD technique that allows to pack many plaintexts into the same ciphertext and manipulate them together during homomorphic operations.

5.2 Ciphertext Packing

In the binary encoding, ciphertext packing means that each ciphertext encrypts s bits, where s is the number of slots in the ciphertext. Then we can use this property in three different ways. First, one could pack the bit representation of each classification label in a single ciphertext and allow the server to send back a single ciphertext to the client. Second, one could encrypt several attributes together and classify them with a single protocol evaluation. Finally, one could encrypt multiple decision node thresholds that must be compared to the same attribute in the DT model.

Packing Classification Label's Bits. Aggregating the decision bits using Algorithm 1 produces for each leaf $v \in \mathcal{L}$ a decision bit $\llbracket b_v \rrbracket$ which encrypts 1 for the classification leaf and 0 otherwise. Moreover, because of SVCP, the bit b_v is replicated to all slots. Now, let k be the number of classification labels (i.e., $|\mathcal{L}| = k$) and its bitlength be $|k|$. For each $v \in \mathcal{L}$, we let c_v denote the classification label $v.\mathsf{cLabel}$ which is $|k|$-bit long and has bit representation $c_v^{\mathsf{b}} = c_{v|k|} \dots c_{v1}$ with corresponding packed encryption $\llbracket \vec{c_v} \rrbracket = \llbracket c_{v|k|} | \dots | c_{v1} | 0 | \dots | 0 \rrbracket$. As a result, computing $\llbracket b_v \rrbracket \boxdot \llbracket \vec{c_v} \rrbracket$ for each leaf $v \in \mathcal{L}$ and summing over all leaves results

$$\begin{array}{l}
\llbracket cx_{i1} \rrbracket = \llbracket x_{i1}^{(1)} | \dots | x_{i1}^{(s)} \rrbracket \\
\qquad \dots \\
\llbracket cx_{i\mu} \rrbracket = \llbracket x_{i\mu}^{(1)} | \dots | x_{i\mu}^{(s)} \rrbracket
\end{array}
\qquad
\begin{array}{l}
\llbracket cy_{j1} \rrbracket = \llbracket y_{j1} | \dots | y_{j1} \rrbracket \\
\qquad \dots \\
\llbracket cy_{j\mu} \rrbracket = \llbracket y_{j\mu} | \dots | y_{j\mu} \rrbracket
\end{array}$$

(a) Packing many x_i (b) Packing y_j

Figure 3: Packing Attribute Values

in the correct classification label. Note that, this assumes that one is classifying only one vector and not many as in the the next case.

Packing Attribute Values. Let $x^{(1)}, \dots, x^{(s)}$ be s possible attribute vectors with $x^{(l)} = [x_1^{(l)}, \dots, x_n^{(l)}]$, $1 \le l \le s$. For each $x_i^{(l)}$, let $x_i^{(l)^{\mathsf{b}}} = x_{i\mu}^{(l)}, \dots, x_{i1}^{(l)}$ be the bit representation. Then, the client generates for each attribute x_i the ciphertexts $\llbracket cx_{i\mu} \rrbracket, \dots, \llbracket cx_{i2} \rrbracket, \llbracket cx_{i1} \rrbracket$ as illustrated in Figure 3a.

To shorten the notation, let y_j denote the threshold of the j-th decision node (i.e., $y_j = v_j.\mathsf{threshold}$) and assume $v_j.\mathsf{aIndex} = i$. The server just encrypts each threshold bitwise which automatically replicates the bit to all slots. This is illustrated in Figure 3b.

Note that $(\llbracket cy_{j\mu} \rrbracket, \dots, \llbracket cy_{j1} \rrbracket) = \llbracket y_j^{\mathsf{b}} \rrbracket$ holds because of SVCP. The above described encoding allows to compare s attribute values together with one threshold. This is possible because the routine SheCmp is compatible with SVCP such that we have: $\textsc{SheCmp}((\llbracket cx_{i\mu} \rrbracket, \dots, \llbracket cx_{i1} \rrbracket), \llbracket y_j^{\mathsf{b}} \rrbracket) = (\llbracket b_{ij}^{(1)} | \dots | b_{ij}^{(s)} \rrbracket, \llbracket b_{ji}^{(1)} | \dots | b_{ji}^{(s)} \rrbracket)$, where $b_{ij}^{(l)} = [x_i^{(l)} > y_j]$ and $b_{ji}^{(l)} = [y_j > x_i^{(l)}]$. This results in a single ciphertext such that the l-th slot contains the comparison result between $b_{ij}^{(l)}$.

Aggregating decision bits remains unchanged as described in Algorithm 1. It results in a packed ciphertext $\llbracket b_v \rrbracket = \llbracket b_v^{(1)} | \dots | b_v^{(s)} \rrbracket$ for each leaf $v \in \mathcal{L}$, where $b_v^{(l)} = 1$ if $x^{(l)}$ classifies to leaf v and $b_u^{(l)} = 0$ for all other leaf $u \in \mathcal{L} \setminus \{v\}$.

For the classification label c_v of a leaf node $v \in \mathcal{L}$, let $\llbracket c_v^{\mathsf{b}} \rrbracket = (\llbracket c_{v|k|} \rrbracket, \dots, \llbracket c_{v1} \rrbracket)$ denote the encryption of the bit representation $c_v^{\mathsf{b}} = c_{v|k|} \dots c_{v1}$. To select the correct classification label algorithm $\textsc{Finalize}(\mathcal{L})$ is updated as follows. We compute $\llbracket c_{v|k|} \rrbracket \boxdot \llbracket b_v \rrbracket, \dots, \llbracket c_{v1} \rrbracket \boxdot \llbracket b_v \rrbracket$ for each leaf $v \in \mathcal{L}$ and sum them componentwise over all leaves. This results in the encrypted bit representation of the correct classification labels.

Packing Threshold Values. In this case, the client encrypts a single attribute in one ciphertext, while the server encrypts multiple threshold values in a single ciphertext. Hence, for an attribute value x_i, the client generates the ciphertexts similar to in Figure 3a. Let m_i be the number of decision nodes that compare to the attribute x_i (i.e., $m_i = |\{v_j \in \mathcal{D} : v_j.\mathsf{aIndex} = i\}|$). The server packs all corresponding threshold values in $\lceil \frac{m_i}{s} \rceil$ ciphertext(s).

The packing of threshold values allows to compare one attribute value against multiple threshold values together. Unfortunately, we do not have access to the slots while performing homomorphic operation. Hence, to aggregate the decision bits, we make m_i copies of the resulting packed decision bits and shift left each decision bit to the first slot. Then the aggregation of the decision bits and the finalizing algorithm work as in the previous case with the only difference that only the result in the first slot matters and the remaining can be set to 0.

5.3 Efficient Path Evaluation

Homomorphic multiplication increases the noise significantly [5]. To evaluate paths, we need to keep the multiplication depth small.

Definition 8 (Multiplicative Depth). *Let f be a function, C_f be a boolean circuit that computes f and consists of AND-gates or multiplication (modulo 2) gates and XOR-gates or addition (modulo 2) gates. The* circuit depth *of C_f is the maximal length of a path from an input gate to an output gate. The* multiplicative depth *of C_f is the path from an input gate to an output gate with the largest number of of multiplication gates.*

Path evaluation requires to homomorphically compute $f([a_1,\dots,a_n]) = \Pi_{i=1}^{n} a_i$, where the a_i are comparison results on the path.

Lemma 2 (Logarithmic Multiplicative Depth). *Let $[a_1,\dots,a_n]$ be an array of n integers and f be the function: $f([a_1,\dots,a_n]) = [a'_1,\dots a'_{\lceil \frac{n}{2} \rceil}]$, where*

$$a'_i = \begin{cases} a_{2i-1} \cdot a_{2i} & \textit{if } (n \bmod 2 = 0) \lor (i < \lceil \frac{n}{2} \rceil), \\ a_n & \textit{if } (n \bmod 2 = 1) \land (i = \lceil \frac{n}{2} \rceil). \end{cases}$$

Moreover, let f be an iterated function where f^i is the i-th iterate defined as:

$$f^i([a_1,\dots,a_n]) = \begin{cases} [a_1,\dots,a_n] & \textit{if } i = 0, \\ f(f^{i-1}([a_1,\dots,a_n])) & \textit{if } i \geq 1. \end{cases}$$

The $|n|$-th iterate $f^{|n|}$ of f computes $\Pi_{i=1}^{n} a_i$ and has multiplicative depth $|n| - 1$ if n is a power of two and $|n|$ otherwise, where $|n| = \log n$ is the bitlength of n: $f^{|n|}([a_1,\dots,a_n]) = [\Pi_{i=1}^{n} a_i]$.

Due to space constraints, we provide the algorithm for path evaluation with multiplicative depth in the Appendix (Algorithm 6). This algorithm consists of a main function EvalPathsE that collects for each leaf v encrypted comparison results on the path from the root to v, and a sub-function EvalMul which multiplies comparison results according to Lemma 2.

Although highly parallelizable, EvalPathsE is still not optimal, as each path is considered individually. Since multiple paths in a binary tree share a common prefix (from the root), one would ideally want to handle common prefixes one time and not many times for each leaf. This can be solved using *memoization* technique which is an optimization that stores results of expensive function calls such that they can be used latter if needed. Unfortunately, naive memoization would require a complex synchronization in a multi-threaded environment and linear multiplicative depth. The next section describes a pre-computation on the tree, that would allow us to have the best of both worlds - multiplication with logarithmic depth along the paths, while reusing the result of common prefixes.

5.4 Improving Path Evaluation with Pre-Computation

The idea behind this optimization is to use directed acyclic graph which we want to define first.

Definition 9 (DAG). *A* directed acyclic graph *(DAG) is a graph with directed edges in which there are no cycles. A vertex v of a DAG is said to be* reachable *from another vertex u if there exists a non-trivial path that starts at u and ends at v. The reachability relationship is a partial order $\leq$ and we say that two vertices u and v are ordered as $u \leq v$ if there exists a directed path from u to v.*

We require our DAGs to have a unique maximum element. The edges in the DAG define dependency relation between vertices.

Definition 10 (Dependency Graph). *Let h be the function that takes two DAGs G_1, G_2 and returns a DAG G_3 that connects the maxima of G_1 and G_2. We define the function $g([a_1, \ldots, a_n])$ that takes an array of integers and returns:*

- *a graph with a single vertex labeled with a_1 if $n = 1$*
- *$h(g([a_1, \ldots, a_{n'}]), g([a_{n'+1}, \ldots, a_n]))$ if $n > 1$ holds, where $n' = 2^{|n|-1}$ and $|n|$ denotes the bitlength of n.*

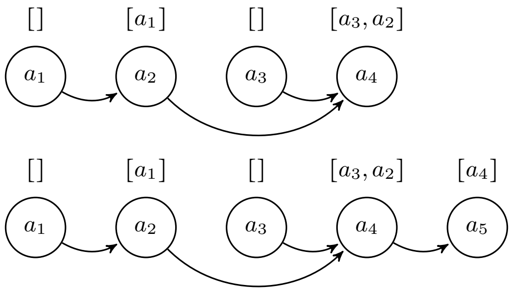
We call the DAG G generated by $G = g([a_1, \ldots, a_n])$ a dependency graph. *For each edge (a_i, a_j) in G such that $i < j$, we say that a_j* depends *on a_i and denote this by adding a_i in the* dependency list *of a_j. We require that if $L(j) = [a_{i_1}, \ldots, a_{i_{|L(j)|}}]$ is the dependency list of a_j then it holds $i_1 > i_2 > \ldots i_{|L(j)|}$.*

An example of dependency graph generated by the function $g([a_1, \ldots, a_n])$ is illustrated in Figure 4a for $n = 4$ and $n = 5$.

Lemma 3. *Let $[a_1, \ldots, a_n]$ be an array of n integers. Then $g([a_1, \ldots, a_n])$ as defined above generates a DAG whose maximum element is marked with a_n.*

Lemma 4. *Let $[a_1, \ldots, a_n]$ be an array of n integers, $G = g([a_1, \ldots, a_n])$ be a DAG as above, and $L(j) = [a_{i_1}, \ldots, a_{i_{|L(j)|}}]$ be the dependency list of a_j. Then the algorithm in Figure 4b computes $\Pi_{i=1}^{n} a_i$ with a multiplicative depth of $\log(n)$.*

The proofs of Lemmas 3 and 4 follow by induction similar to Lemma 2. Before describing the improved path evaluation algorithm, we first extend our Node data structure by adding to it a new field representing a stack denoted dag, that stores the dependency list. Moreover, we group the nodes of the DT by level and use an array denoted level[], such that level[0] stores a pointer to the root and level[i] stores pointers to the child nodes of level[$i-1$] for $i \geq 1$.



(a) DAG for $n = 4$ and $n = 5$

```
for $j = 1$ to $j = n$ do
    for $l = 1$ to $l = |L(j)|$ do
        $a_j \leftarrow a_j \cdot a_{i_l}$
```

(b) Mult. Algorithm

Figure 4: DAG Pre-computation and Multiplication Algorithm using DAG

Now, we are ready to describe the improved path evaluation algorithm which consists of a pre-computation step and an online step.

The pre-computation is a one-time computation that depends only on the structure of the DT and requires no encryption. As described in Algorithm 7, its main function ComputeDag uses the leveled structure of the tree and the dependency graph defined above to compute the dependency list of each node in the tree (i.e., the DAG defined above). The sub-function AddEdge is used to actually add nodes to the dependency list of another node (i.e., by adding edges between these nodes in the DAG).

The online step is described in Algorithm 8. It follows the idea of the algorithm in Figure 4b by multiplying decision bit level-wise depending on the dependency lists. The correctness follows from Lemma 4.

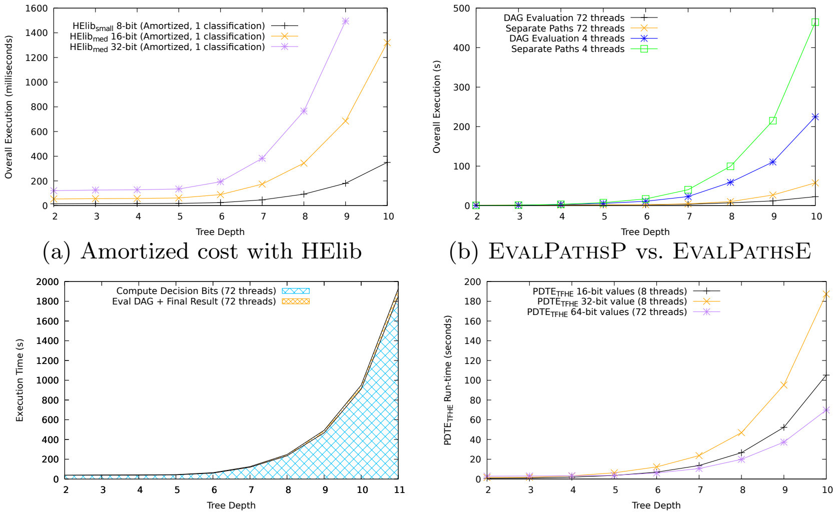
6 Evaluation

In this section, we discuss some implementation details and evaluate our schemes.

6.1 Implementation Details

We implemented our algorithms using HElib [21] and TFHE [10, 11]. We also implemented the main algorithm using an arithmetic circuit based on Lin-Tzeng comparison protocol [26,35]. We refer to this implementation as Pdt-Int. HElib is a C++ library that implements FHE. The current version includes an implementation of the BGV scheme [5]. HElib also includes various optimizations that make FHE runs faster, including ciphertext packing (SVCP) techniques [31].

TFHE is a C/C++ library that implements FHE proposed by Chillotti et al. [10]. It allows to evaluate any boolean circuit on encrypted data. The current version implements a very fast gate-by-gate bootstrapping, i.e., bootstrapping is performed after each gate evaluation. Future versions will include leveled FHE and ciphertext packing Chillotti et al. [10]. Dai and Sunar [13, 14] propose an implementation of TFHE on CUDA-enabled GPUs that is 26 times faster.



(a) Amortized cost with HElib

(b) EvalPathsP vs. EvalPathsE

(c) Using HElib$_{\text{med}}$ for 16-bit inputs

(d) Pdt-Bin Runtime with TFHE

Figure 5: Pdt-Bin Runtime

We evaluated our scheme on an AWS instance with Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz running Ubuntu 18.04.2 LTS; with 36 CPUs, 144 GB Memory and 8 GB SSD. As the bottleneck of our scheme is the overhead of the HE, we focus on the computation done by the server. We describe HE parameters and the performance of basic operations in Appendix A.

6.2 Performance of Pdt-Bin

In this section, we report on our experiment with Pdt-Bin on complete trees. Recall that for FHE supporting SIMD, we can use attribute values packing that allows to evaluate many attribute vectors together. We, therefore, focus on attribute packing to show the advantage of SIMD. Figure 5a illustrates the amortized runtime of Pdt-Bin with HElib. That is, the time of one PDTE evaluation divided by the number of slots in a ciphertext. As one can expect, the runtime clearly depends on the bitlength of the attribute values and the depth of the tree. The results show a clear advantage of HElib when classifying large data sets. For paths aggregation, we proposed EvalPathsE (Algorithm 6) and EvalPathsP (Algorithm 8). Figure 5b illustrates Pdt-Bin runtime using these algorithms in a multi-threaded environment and shows a clear advantage of EvalPathsP which will be used in the remaining experiments with Pdt-Bin. Figure 5c illustrates the runtime of Pdt-Bin with HElib$_{\text{med}}$ showing that the computation cost is dominated by the computation of decision bits which involves homomorphic evaluation of comparison circuits. In Figure 5d, we report the evaluation of Pdt-Bin using TFHE, which shows a clear advantage compared to HElib. For the same experiment with 72 threads, TFHE evaluates a

		Pdt-Bin (TFHE)	Pdt-Bin (HElib)		Pdt-Int (HElib)		[28] (HElib)	[32] (mcl [29])
λ		128	150		135		128	128
μ		16	16		16		12	64
#thd		16	16		16		16	-
	n, d, m	one	am.	one	am.	one	one	one
Heart-disease	13, 3, 5	0.94	0.05	40.61	0.0073	45.59	0.59	0.25
Housing	13, 13, 92	6.30	0.35	252.38	0.90	428.23	10.27	1.98
Spambase	57, 17, 58	3.66	0.24	174.46	0.72	339.60	6.88	1.80
Artificial	16, 10, 500	22.39	1.81	1303.55	0.75	2207.13	56.37	10.42

Table 1: Runtime (in seconds) on Real Datasets: λ is the security level. μ is the input bit length. #thd is the number of threads. Column 'one' is the time for one protocol run while 'am.' is the amortized time (e.g., the time for one run divided by s).

complete tree of depth 10 and 64-bit input in less than 80 seconds, while HElib takes about 400 seconds for 16-bit input. Recall that, a CUDA implementation [13,14] of TFHE can further improve the time of Pdt-Bin using TFHE.

6.3 Performance on Real Datasets

We also performed experiments on real datasets from the UCI repository [36]. For Pdt-Bin, we reported the costs for HElib (single and amortized) and the costs for TFHE. Since TFHE evaluates only boolean circuits, we only have implementation and evaluation of Pdt-Int with HElib. We also illustrate in Table 1 the costs of two best previous works that rely only on HE, whereby the figures are taken from the respective papers [28, 32]. For one protocol run, Pdt-Bin with TFHE is much more faster than Pdt-Bin with HElib which is also faster than Pdt-Int with HElib. However, because of the large number of slots, the amortized cost of Pdt-Bin with HElib is better. For 16-bit inputs, our amortized time with HElib and our time with TFHE outperform XCMP [28] which used 12-bit inputs. For the same input bitlength, XCMP is still much more better than our one run using HElib, since the multiplicative depth is just 3. However, our schemes still have a better communication and Pdt-Bin has no leakage. While the scheme of Tai et al. [32] in the semi-honest model has a better time for 64-bit inputs than our schemes for 16-bit inputs, it requires a fast network communication and at least double cost in the malicious model. The efficiency of Tai et al. is in part due to their ECC implementation of the lifted ElGamal, which allows a fast runtime and smaller ciphertexts, but is not secure against a quantum attacker, unlike lattice-based FHE as used in our schemes.

7 Related Work

Our work is related to secure multiparty computation (SMC) [16, 20], private function evaluation (PFE) [25,30] particularly privacy-preserving DT evaluation [2, 4, 7, 23, 24, 32, 34, 38] which we briefly review in this section and refer to the literature for more details. Brikell et al. [7] propose the first protocol for PDTE

Scheme	Rounds	Tools	Communication	Comparisons	Leakage
[7]	≈5	HE+GC	$\mathcal{O}(2^d)$	d	m, d
[2]	≈4	HE+GC	$\mathcal{O}(2^d)$	d	m, d
[4]	≥6	FHE/SHE	$\mathcal{O}(2^d)$	m	m
[38]	6	HE+OT	$\mathcal{O}(2^d)$	m	m
[32]	4	HE	$\mathcal{O}(2^d)$	m	m
[12]	≈9	SS	$\mathcal{O}(2^d)$	m	m, d
[34]	$\mathcal{O}(d)$	GC,OT ORAM	$\mathcal{O}(2^d)$ $\mathcal{O}(d^2)$	d	m, d
[28]	1	FHE/SHE	$\mathcal{O}(2^d)$	m	m
PDT-BIN	1	FHE/SHE	$\mathcal{O}(1)$, $\mathcal{O}(d)$	m	-
PDT-INT	1		$\mathcal{O}(2^d/s)$		m

Table 2: Comparison of PDTE protocols.

Scheme	SIMD	Generic	Output-expressive	Multiplicative Depth	Output Length
[28]	no	no	no	3	2^{d+1}
PDT-BIN	yes	yes	yes	$\lvert\mu\rvert + \lvert d\rvert + 2$	1 or d
PDT-INT	yes	yes	no	$\lvert\mu\rvert + 1$	$\lceil 2^d/s \rceil$

Table 3: Comparison of 1-round PDTE protocols.

by combining additively HE and garbled circuits (GC) in a novel way. Although the evaluation time of Brikell et al.'s scheme is sublinear in the tree size, the secure program itself and hence the communication cost is linear and therefore not efficient for large trees. Barni et al. [2] improve the previous scheme by reducing the computation costs by a constant factor. Bost et al. [4] represent the DT as a multivariate polynomial. The parties evaluate this polynomial interactively and encrypted under the client's public key using a fully HE Wu et al. [38] use different techniques that require only additively HE (AHE). They use the DGK protocol [15] for comparison and reveal to the server comparison bits encrypted under the client's public key. Tai et al. [32] use the DGK comparison protocol [15] and AHE as well, but mark the edges with the comparison results at each node. Tueno et al. [34] represent the tree as an array. Then, they traverse the tree interactively and use secure array indexing to select the next node and attribute. Kiss et al. [24] propose a modular design consisting of the sub-functionalities: selection of attributes, integer comparison, and evaluation of paths. De Cock et al. [12] follow the same idea as some previous schemes, but operate in the information theoretic model using secret sharing (SS) based SMC and commodity-based cryptography [3]. Using a polynomial encoding of the inputs and BGV homomorphic scheme [5], Lu et al. [28] propose a non-interactive comparison protocol called XCMP which is *output expressive* (i.e., it preserves additive homomorphism). They implement the scheme of Tai et al. [32] using XCMP. The resulting PDTE is non-interactive and efficient because of the small *multiplicative depth*. However, it is not *generic*, as it primarily works for small inputs and depends explicitly on BGV-type HE scheme. Moreover, it does not support *SIMD* operations and is no longer output expressive as XCMP. Hence, it cannot be extended to a larger protocol (e.g., random forest [6]) while preserving the non-interactive property. Finally, its *output length* (i.e., the number of resulted ciphertexts from server computation) is exponential in the depth d of the tree, while the output length of our binary instantiation Pdt-Bin is at most linear in d. A comparison of PDTE protocols is summarized in Tables 2 and 3. A more detailed complexity analysis of our schemes is described in Appendix C.

8 Conclusion

While almost all existing PDTE protocols require many interaction between the client and the server, we designed and implemented novel client-server protocols that delegate the complete evaluation to the server while preserving privacy and keeping the overhead low. Our solutions rely on SHE/FHE and evaluate the tree on ciphertexts encrypted under the client's public key. Since current SHE/FHE schemes have high overhead, we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low.

References

1. Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Hoffstein, J., Lauter, K., Lokam, S., Micciancio, D., Moody, D., Morrison, T., Sahai, A., Vaikuntanathan, V.: Homomorphic encryption security standard. Tech. rep., HomomorphicEncryption.org, Cambridge MA (March 2018)
2. Barni, M., Failla, P., Kolesnikov, V., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Secure evaluation of private linear branching programs with medical applications. In: ESORICS. pp. 424–439. Springer-Verlag, Berlin, Heidelberg (2009)
3. Beaver, D.: Commodity-based cryptography (extended abstract). In: STOC. pp. 446–455. ACM, New York, NY, USA (1997)
4. Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine learning classification over encrypted data. In: NDSS (2015)
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. ECCC **18**, 111 (2011)
6. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (Oct 2001)
7. Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: CCS. pp. 498–507. ACM, New York, NY, USA (2007)
8. Büscher, N., Demmler, D., Katzenbeisser, S., Kretzmer, D., Schneider, T.: Hycc: Compilation of hybrid protocols for practical secure computation. In: CCS '18. pp. 847–861 (2018)
9. Cheon, J.H., Kim, M., Kim, M.: Search-and-compute on encrypted data. In: FC. pp. 142–159 (2015)
10. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. IACR Cryptology ePrint Archive **2018**, 421 (2018)

11. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast fully homomorphic encryption library (August 2016), https://tfhe.github.io/tfhe/
12. Cock, M.D., Dowsley, R., Horst, C., Katti, R.S., Nascimento, A.C.A., Poon, W., Truex, S.: Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. IEEE Trans. Dependable Sec. Comput. **16**(2), 217–230 (2019)
13. Dai, W., Sunar, B.: cuhe: A homomorphic encryption accelerator library. In: BalkanCryptSec 2015 (September 2015)
14. Dai, W., Sunar, B.: Cuda-accelerated fully homomorphic encryption library (August 2019), https://github.com/vernamlab/cuFHE
15. Damgård, I., Geisler, M., Krøigaard, M.: Efficient and secure comparison for on-line auctions. In: ACISP. pp. 416–430 (2007)
16. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO '12. pp. 643–662 (2012)
17. Franz, M., Holzer, A., Katzenbeisser, S., Schallhart, C., Veith, H.: CBMC-GC: an ANSI C compiler for secure two-party computations. In: CC '14. pp. 244–249 (2014)
18. Fredrikson, M., Jha, S., Ristenpart, T.: Model inversion attacks that exploit confidence information and basic countermeasures. In: CCS. pp. 1322–1333 (2015)
19. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC. pp. 169–178. ACM, New York, NY, USA (2009)
20. Goldreich, O.: Foundations of Cryptography: Volume 2, Basic Applications. Cambridge University Press, New York, NY, USA (2004)
21. Halevi, S., Shoup, V.: Algorithms in helib. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 8616, pp. 554–571. Springer (2014)
22. Hazay, C., Lindell, Y.: Efficient Secure Two-Party Protocols: Techniques and Constructions. Springer-Verlag New York, Inc., New York, NY, USA, 1st edn. (2010)
23. Joye, M., Salehi, F.: Private yet efficient decision tree evaluation. In: DBSec. Lecture Notes in Computer Science, vol. 10980, pp. 243–259. Springer (2018)
24. Kiss, Á., Naderpour, M., Liu, J., Asokan, N., Schneider, T.: Sok: Modular and efficient private decision tree evaluation. PoPETs **2019**(2), 187–208 (2019)
25. Kiss, A., Schneider, T.: Valiant's universal circuit is practical. In: EUROCRYPT '2016. pp. 699–728 (2016)
26. Lin, H., Tzeng, W.: An efficient solution to the millionaires' problem based on homomorphic encryption. In: ACNS. pp. 456–466 (2005)
27. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: Oblivm: A programming framework for secure computation. In: SP. pp. 359–376 (2015)
28. Lu, W.j., Zhou, J.j., Sakuma, J.: Non-interactive and output expressive private comparison from homomorphic encryption. In: ASIACCS '18. pp. 67–74 (2018)
29. mcl library (September 2019), https://github.com/herumi/mcl/
30. Mohassel, P., Sadeghian, S.S., Smart, N.P.: Actively secure private function evaluation. In: ASIACRYPT. pp. 486–505 (2014)
31. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. Des. Codes Cryptography **71**(1), 57–81 (2014)
32. Tai, R.K.H., Ma, J.P.K., Zhao, Y., Chow, S.S.M.: Privacy-preserving decision trees evaluation via linear functions. In: ESORICS. pp. 494–512 (2017)
33. Tramèr, F., Zhang, F., Juels, A., Reiter, M.K., Ristenpart, T.: Stealing machine learning models via prediction apis. In: USENIX. pp. 601–618 (2016)
34. Tueno, A., Kerschbaum, F., Katzenbeisser, S.: Private evaluation of decision trees using sublinear cost. PoPETs **2019**, 266–286

35. Tueno, A., Kerschbaum, F., Katzenbeisser, S., Boev, Y., Qureshi, M.: Secure computation of the kth-ranked element in a star network. In: FC '20 (2020)

36. Uci repository. http://archive.ics.uci.edu/ml/index.php (2019)

37. Witten, I.H., Frank, E., Hall, M.A.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edn. (2011)

38. Wu, D.J., Feng, T., Naehrig, M., Lauter, K.: Privately evaluating decision trees and random forests. PoPETs **2016**(4), 335–355 (2016)

39. Wu, X., Fredrikson, M., Jha, S., Naughton, J.F.: A methodology for formalizing model-inversion attacks. In: CSF. pp. 355–370 (2016)

A Encryption Parameters and Basic Operations

Recall that FHE schemes − as considered in this paper − are usually defined over a ring $\mathbb{Z}[X]/(X^N + 1)$ and that the encryption scheme might be a leveled FHE with parameter L. For HElib, the parameters N and L determines how to generate encryption keys for a security level λ which is at least 128 in all our experiments. The degree of the ring polynomial in HElib is not necessarily a power of 2. The ring polynomial is chosen among the cyclotomic polynomials. For HElib, we abuse the notation and use N to denote the N-th cyclotomic polynomial. Given the value of L and other parameters, the HElib function $\mathsf{FindM}(\cdot)$ computes the N-th cyclotomic polynomial, that guarantees a security level at least equal to a given security parameter λ. Table 4 summarizes the parameters we used for key generation and the resulting sizes for encryption keys and ciphertexts. We refer to it as *homomorphic context* or just *context*. For HElib, one needs to choose L large enough than the depth of the circuit to be evaluated and then computes an appropriate value for N that ensures a security level at least 128. We experimented with tree different contexts ($\text{HElib}_{\text{small}}, \text{HElib}_{\text{med}}, \text{HElib}_{\text{big}}$) for the binary representation used in PDT-BIN, and $\text{HElib}_{\text{int}}$ for the integer representation used in PDT-INT. For TFHE, the default value of N is 1024 and the security level is 128 while L is infinite because of the gate-by-gate bootstrapping. We used the context TFHE_{128} to evaluate PDT-BIN with TFHE. The last two columns of Table 4 reports the average runtime for encryption and decryption over 100 runs.

Name	L	N	λ	**Slots**	sk	pk	**Ctxt**	Enc	Dec
			(bits)		(MB)	(MB)	(MB)	(ms)	(ms)
$\text{HElib}_{\text{small}}$	200	13981	151	600	52.2	51.6	1.7	59.21	26.08
$\text{HElib}_{\text{med}}$	300	18631	153	720	135.4	134.1	3.7	124.39	54.31
$\text{HElib}_{\text{big}}$	500	32109	132	1800	370.1	367.1	8.8	283.49	127.11
$\text{HElib}_{\text{int}}$	450	24793	138.161	6198	370.1	367.1	8.8	323.41	88.63
TFHE_{128}	∞	1024	128	1	82.1	82.1	0.002	0.04842	0.00129

Table 4: HE Parameters and Results: For HElib, column N is not the degree of the ring polynomial, but the N-th cyclotomic polynomial. It is computed in HElib using a function called $\mathsf{FindM}(\cdot)$.

```
Input: leaves set $\mathcal{L}$, decision nodes set $\mathcal{D}$
Output: Updated $v$.cmp for each $v \in \mathcal{L}$
function EvalPathsE($\mathcal{L}$, $\mathcal{D}$)
  for each $v \in \mathcal{L}$ do
    let $d$ = #nodes on the path (root $\rightarrow v$)
    let path = empty array of length $d$
    $l \leftarrow d$
    $w \leftarrow v$
    while $w \neq$ root do                          ▷ path to root
      path[$l$] $\leftarrow [\![w.\mathsf{cmp}]\!]$
      $l \leftarrow l - 1$
      $w \leftarrow w$.parent
    $[\![v.\mathsf{cmp}]\!] \leftarrow$ EvalMul(1, $d$, path)

Input: integers from, to; array of nodes path
Output: Product of elements in path
function EvalMul(from, to, path)
  if from ≥ to then
    return path[from]
  $n \leftarrow$ to − from + 1
  mid $\leftarrow 2^{|n-1|-1}$ + from − 1        ▷ $|n| = \log_2(n)$
  $[\![\mathsf{left}]\!] \leftarrow$ EvalMul(from, mid, path)
  $[\![\mathsf{right}]\!] \leftarrow$ EvalMul(mid + 1, to, path)
  return $[\![\mathsf{left}]\!] \boxdot [\![\mathsf{right}]\!]$
```

Algorithm 6: Paths Evaluation with log Multiplicative Depth

```
Input: integers up and low
Output: Computed $v$.dag for each $v \in \mathcal{D} \cup \mathcal{L}$
function ComputeDag(up, low)
  if up ≥ low then
    return                                  ▷ end the recursion
  $\delta \leftarrow$ low − up + 1
  mid $\leftarrow 2^{|\delta-1|-1} - 1$ + up    ▷ $|\delta|$ bitlength of $\delta$
  for each $v \in$ level[low] do
    AddEdge($v$, low, mid)
  for $i$ = mid + 1 to low − 1 do           ▷ non-deepest leaves
    for each $v \in$ level[$i$] $\cap \mathcal{L}$ do
      AddEdge($v$, $i$, mid)
  ComputeDag(up, mid)
  ComputeDag(mid + 1, low)

Input: Node $v$, integers currLvl and destLvl
Output: Updated $v$.dag
function AddEdge($v$, currLvl, destLvl)
  $w \leftarrow v$
  while currLvl > destLvl do
    $w \leftarrow w$.parent
    currLvl ← currLvl − 1
  $v$.dag.push($w$)                          ▷ dag is a stack
```

Algorithm 7: Pre-computation of Multiplication DAG

B Security Analysis

It is straightforward to see that our protocols are secure. There is no interaction with the client during the computation and a semi-honest server sees only IND-CPA ciphertexts. A semi-honest client only learns the encryption of the result. A malicious server can only return a false classification result. This is inherent to PFE where the function (the DT in our case) is an input to the computation. A malicious client can send a too 'noisy' ciphertext, such that after the computation at the server a correct decryption is not possible, leaking some information. This attack works only with level FHE and is easy to deal with, namely the computation of a ciphertext capacity is a public function which the server can use to check the ciphertexts before starting the computation. As Pdt-Bin returns the bit representation of the resulted classification label whose bitlength is public (i.e., the set of possible classification labels is known to the client), there is no leakage beyond the final output. Pdt-Int returns as many ciphertexts as there are leaves and, therefore, leaks the number of decision nodes.

```
Input: nodes stored by level in array level
Output: Updated $v$.cmp for each $v \in \mathcal{L}$
function EvalPathsP
    for $i = 1$ to $d$ do                    ▷ top to bottom level
        for each $v \in$ level[$i$] do
            while NOT $v$.dag.empty() do     ▷ stack
                $w \leftarrow v$.dag.pop()
                $[\![v.\mathsf{cmp}]\!] \leftarrow [\![v.\mathsf{cmp}]\!] \boxdot [\![w.\mathsf{cmp}]\!]$
```

Algorithm 8: Aggregate Decision Bits with DAG

C Complexity analysis

We now analyse the complexity of our schemes. We assume that the decision tree is a complete tree with depth d.

Complexity of Pdt-Bin. The SHE comparison circuit has multiplicative depth $|\mu - 1| + 1$ and requires $\mathcal{O}(\mu \cdot |\mu|)$ multiplications [9]. That is, the evaluation of all decision nodes requires $\mathcal{O}\left(2^d \mu \cdot |\mu|\right)$ multiplications. The path evaluation has a multiplicative depth of $|d-1|$ and requires for all 2^d paths $\mathcal{O}\left(d2^d\right)$ multiplications. The evaluation of the leaves has a multiplicative depth of 1 and requires in total 2^d multiplications. The total multiplicative depth for Pdt-Bin is, therefore, $|\mu - 1| + |d-1| + 2 \approx |\mu| + |d| + 2$ while the total number of multiplications is $\mathcal{O}\left(2^d \mu \cdot |\mu| + d2^d + 2^d\right) \approx \mathcal{O}\left(d2^d\right)$.

For the label packing, the bit representation of each classification label is packed in one ciphertext. This hold for the final result as well. As a result, if the tree is complete and all classification labels are distinct, then the server sends $\left\lceil \frac{d}{s} \right\rceil$ ciphertext(s) to client. In practice, however, $\left\lceil \frac{d}{s} \right\rceil = 1$ holds as d is smaller that the number s of slots.

For threshold packing, the decision bit b_v at node v is encrypted as $[\![b_v|0|...|0]\!]$. By encrypting the classification label $c_i = c_{i|k|}...c_{i1}$ as $[\![c_{i|k|}|0|...|0]\!], ..., [\![c_{i1}|0|...|0]\!]$, the final result c_l will be encrypted similarly such that with extra shifts, we can build the ciphertext $[\![c_{l|k|}|...|c_{l1}|0|...|0]\!]$. As a result, the server sends only 1 ciphertext back to the client.

For other cases (e.g., attribute packing, or no packing at all as in the current implementation of TFHE), the bits of a classification label are encrypted separately which holds for the final result as well. As a result the server sends back d ciphertexts to the client.

Complexity of Pdt-Int. The modified Lin-Tzeng comparison circuit has multiplicative $|\mu - 1|$ and requires $\mathcal{O}(\mu - 1)$ multiplications. As a result, the evaluation of all decision node requires $\mathcal{O}\left((\mu - 1)2^d\right)$ multiplications. In Pdt-Int, the path evaluation does not requires any multiplication. However, the leave evaluation has a multiplicative depth of 1 and requires in total 2^d multiplications. The total multiplicative depth for Pdt-Int is therefore $|\mu - 1| + 1 \approx |\mu| + 1$ while the total number of multiplications is $\mathcal{O}\left((\mu - 1)2^d + 2^d\right) \approx \mathcal{O}\left(2^d\right)$.

For Pdt-Int, it is not possible to aggregate the leaves as in Pdt-Bin. If the client is classifying many inputs, the server must send 2^d ciphertexts back. If the client is classifying only one input, then the server can use shifts to pack the result in $\left\lceil \frac{2^d}{s} \right\rceil$ ciphertext(s).