Anselme Tueno*, Florian Kerschbaum, and Stefan Katzenbeisser

Private Evaluation of Decision Trees using Sublinear Cost

Abstract: Decision trees are widespread machine learning models used for data classification and have many applications in areas such as healthcare, remote diagnostics, spam filtering, etc. In this paper, we address the problem of privately evaluating a decision tree on private data. In this scenario, the server holds a private decision tree model and the client wants to classify its private attribute vector using the server's private model. The goal is to obtain the classification while preserving the privacy of both – the decision tree and the client input. After the computation, only the classification result is revealed to the client, while nothing is revealed to the server. Many existing protocols require a constant number of rounds. However, some of these protocols perform as many comparisons as there are decision nodes in the entire tree and others transform the whole plaintext decision tree into an oblivious program, resulting in higher communication costs. The main idea of our novel solution is to represent the tree as an array. Then we execute only d – the depth of the tree – comparisons. Each comparison is performed using a small garbled circuit, which output secret-shares of the index of the next node. We get the inputs to the comparison by obliviously indexing the tree and the attribute vector. We implement oblivious array indexing using either garbled circuits, Oblivious Transfer or Oblivious RAM (ORAM). Using ORAM, this results in the first protocol with sub-linear cost in the size of the tree. We implemented and evaluated our solution using the different array indexing procedures mentioned above. As a result, we are not only able to provide the first protocol with sublinear cost for large trees, but also reduce the communication cost for the large real-world data set "Spambase" from 18 MB to 1.2 MB and the computation time from 17 seconds to less than 1 second in a LAN setting, compared to the best related work.

Keywords: Private Decision Tree Evaluation, Garbled Circuits, ORAM, Oblivious Transfer

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

***Corresponding Author: Anselme Tueno:** SAP SE, E-mail: anselme.kemgne.tueno@sap.com

1 Introduction

Decision trees are common and very popular classifiers because of their simplicity and ease of use. A decision tree consists of two types of nodes. Internal nodes are *decision nodes* that are used to compare an attribute to a constant. *Leaf nodes* give a classification that applies to all instances that reach the leaf. To classify an unknown instance, the tree is traversed according to the values of the attributes tested in successive nodes, and when a leaf is reached the instance is classified according to the class assigned to that leaf [65].

Setting. Machine learning (ML) classifiers are valuable tools in many areas such as healthcare, finance, spam filtering, intrusion detection, remote diagnosis, etc [65]. To perform its task the classifier requires access to user's data, which is most of the time sensitive information, such as medical or financial data. Therefore, it is crucial to investigate privacy-preserving ML classification. On the one hand, the model itself may contain sensitive data. For example, a bank that uses a decision tree for credit assessment of its customers may not want to reveal any information about the model. 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* [27, 61, 67], which can compromise the privacy of the training data.

Scenario. In this paper, we address the problem of privately evaluating a decision tree on private data. In this scenario, a client wants to classify its private attribute vector using a server's private decision tree model. The goal is to obtain the classification, while preserving privacy of both the decision tree and the client input. After the computation, the result of the classification is revealed only to the client and nothing

Florian Kerschbaum: University of Waterloo, E-mail: florian.kerschbaum@uwaterloo.ca
Stefan Katzenbeisser: TU Darmstadt, E-mail: katzenbeisser@seceng.informatik.tu-darmstadt.de

else is revealed either to the client or the server.

Solution Approach. The main idea of our novel solution is to represent the tree as an array. Then we execute only d comparisons, where d denotes the depth of the tree. The result of each comparison allows to obliviously select the index of the next node which is never revealed to any party in clear. This selection of the next node is computed using a small garbled circuit (GC), which is independent of the position in the tree and its size. This GC is handcrafted and executed using the ObliVM GC runtime. However, it is also possible to use other GCs runtime environment like SCAPI [25]. The framework ObliVM was chosen because of its implementation of the state-of-the-art ORAM [62]. We get the inputs to the comparison by obliviously indexing the tree and the attribute vector. We construct oblivious array indexing using either garbled circuits (GC), Oblivious Transfer (OT) or Oblivious RAM (ORAM).

Using ORAM our protocol is the first that achieves sub-linear communication and computation cost in the size of the tree, and hence likely the first able to evaluate very large trees with thousands to millions of nodes, which we will see with the future growth of big data [14]. Currently, communication cost is an important asset in mobile settings. Remote clients using smartphones often do not have the bandwidth to run heavy protocols. However, ORAM has large constants hidden in its asymptotic complexity and a significant setup cost that needs to be amortized over many invocations of the protocol. Hence, we aim not only at improving asymptotic communication cost, but also at the practical cost on real-world data sets. Other alternative indexing protocols, particularly OT, have much smaller constants and help improve the practical communication cost for smaller trees. By using OT instead of ORAM, we reduce the cost for the large real-world data set "Spambase" in the UCI repository from 18 MB to 1.2 MB and the computation time from 17 seconds to less than 1 second in a LAN setting, compared to the best related work.

Generic Solution. While generic secure multiparty computation [17, 29, 68] can implement a decision tree classifier, they are not efficient, in particular when the size of the tree is large. For example, frameworks such as ObliVM [49] or CBMC-GC [26] are able to transform plaintext programs into oblivious programs suitable for secure computation. Their straightforward application to decision tree programs does certainly improve performance over a manual construction. However, the size of the resulting oblivious program is still proportional to the size of the tree. Decision programs can be seen as nested $if-$instructions; the ObliVM compiler transforms the exemplary program IF (s) THEN $x = 1$; ELSE $x = 2$; where the Boolean expression s involves secret variables, into $x_1 = 1$; $x_2 = 2$; $x = \text{MUX}(s, x_1, x_2)$, where MUX is a multiplexer that returns either x_1 or x_2, depending on s being true or false. Any framework implementing the whole program will generate for each condition a corresponding oblivious computation. This results in an oblivious program whose size is linear in the number of decision nodes, which might be too large to be practical for some applications (e.g., Spam filtering). Therefore, optimized protocols, which exploit domain knowledge of the problem at hand and make use of generic techniques only where it is necessary, yield more efficient solutions [1, 4, 9, 11, 12]. Many proposed protocols for privately evaluating decision trees have a constant number of rounds at the cost of performing as many comparisons as there are decision nodes or transforming the whole plaintext decision tree into an oblivious program.

Application to ML-as-a-service. As concrete motivation for the usefulness of privately evaluating a decision tree on private data consider remote diagnostic services [12], healthcare [4]. Many cloud providers are already proposing platforms that allow users to build machine learning applications [1]. A hospital may want to use such a platform to offer a medical expert system as a ML-as-a-service application to other doctors or even its patients. A software provider may leverage ML-as-a-service to allow its customers to detect the cause of a software error. Software systems use log files to collect information about the system behavior. In case of an error these log files can be used to find the cause of the crash. Both examples (medical data and log files) contain sensitive information which is worth to protect.

Contribution. Our contributions are as follows:

– The idea is to represent the tree as an array, where each element contains a node of the tree and pointers to its child nodes. Then, while traversing the tree, we obliviously select the next node and secret-

1 https://bigml.com/
https://azure.microsoft.com/de-de/services/machine-learning/
https://aws.amazon.com/machine-learning
https://cloud.google.com/prediction/docs/
http://predictionio.incubator.apache.org/index.html

share it to the parties. The comparison at each node is performed using a GC, that takes secret-share of that node and the corresponding attribute value and returns secret-shares of the index of the next node.
– We instantiate our protocol by the different indexing procedures mentioned above. In particular, instantiating our protocol with OT is more efficient for small to mid-size trees. With ORAM [62] our scheme has sublinear communication.
– Finally, we implement and evaluate our scheme and demonstrate its practicality regarding runtime and bandwidth. For small size trees our scheme competes with previous protocols, but outperforms them if the size of the decision tree becomes larger – using OT for small to mid-size trees and ORAM for very large trees.

The remainder of the paper is structured as follows. We review related work in Section 2 and preliminaries in Section 3 before defining correctness and security of our protocol in Section 4. In Section 4 we also define a primitive called oblivious array indexing on which our scheme relies. Our main construction itself is described in Section 5. In Section 6 we instantiate oblivious array indexing with garbled circuits and Oblivious Transfer. In Section 7 we discuss some optimizations. We discuss implementation details and the evaluation in Section 8 before concluding our work in Section 9. Due to space constraints, we discuss remaining aspects such complexity analysis, correctness and security proofs in appendix.

2 Related Work

Our work is related to secure multiparty computation (SMC) [8, 13, 15, 17, 20, 21, 29, 39, 68], private function evaluation (PFE) [44, 51] and privacy-preserving machine learning. SMC allows several parties to compute a public function on their private inputs without revealing any information other than the function's output. PFE is a special case of SMC where the function to be computed is not public but private. Privacy-preserving machine learning takes advantage of SMC techniques to build classifiers on private databases [35, 45–47] or to classify private data with private models [4, 11, 12, 16, 32, 35, 52, 66]. Since our work falls under the second category, we concentrate in this section only on privacy-preserving classifiers, particularly decision trees.

Solutions Based on Program Transformation. Brikell et al. [12] combine homomorphic encryption (HE) and GCs in a novel way. In an initial phase the server non-interactively transforms the plaintext decision tree in a secure program, by permuting the tree and replacing each decision node by a small GC implementing offset integer comparison, and each leaf node by an encryption of the corresponding classification label. The GC at a decision node will allow the client in the evaluation phase to learn the decryption key of one child node according to the result of the comparison. In the second phase the parties execute an oblivious attribute selection protocol, where the client uses a homomorphic scheme to encrypt each element of the attribute vector under his public key. The server receives the encrypted vector, permutes it, homomorphically blinds each element and sends it back to client. The client decrypts the vector and the two parties execute Oblivious Transfers that allow the client to learn the keys corresponding to its input. In the last phase, the client receives the secure program and evaluates it.

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 hence not efficient for large trees. Barni et al. [4] improve the previous scheme by not including the leaf node in the transformed secure program, thereby reducing costs by a constant factor, however maintaining linear communication cost. Although more efficient, it is still suitable only for small trees.

Solutions Based on Homomorphic Encryption. Using homomorphic encryption (HE) Bost et al. [11] propose a privacy-preserving protocol for different classifiers including decision trees. They represent the decision tree as a polynomial P whose output is the result of the classification. The constant values of the polynomial are the classification labels and the variables represent the results of the Boolean conditions at the decision nodes. Then the parties privately compute the inputs to this polynomial by comparing each threshold of the tree with the corresponding element of the attribute vector. Finally, the server privately evaluates the polynomial and returns the result to the client. Privacy of the tree is guaranteed by the fact that the server evaluates the polynomial non-interactively. The evaluation is done homomorphically on inputs encrypted under the client public key. The number of invocations of the comparison protocol and the size of the polynomial are linear in the size of the tree. Moreover, the evaluation requires fully HE or at least somewhat HE.

Wu et al. [66] improve the protocol of [11] by using different techniques so that the protocol requires only additive HE. Using the protocol from [18], they also perform as many comparisons as there are decision nodes. The server receives each comparison bit, encrypted under the client public key, and uses them to evaluate the decision tree. The evaluation returns the index of the corresponding classification label to the client. Finally, the parties execute Oblivious Transfer to allow the client to learn the classification label. Their protocol is more efficient than [11] because it relies on additive HE, implemented using a variant of ElGamal based on elliptic curve cryptography.

Tai et al. [60] follow the same blueprint as in [66] by using the comparison protocol of [18]. Then they mark the left and right edge of each node with the cost b and $1-b$ respectively, where b is the result of the comparison at that node. Finally, they sum for each path of the tree the cost along it. The label of the path which costs sum to zero, is the classification label.

Solution Based on Secret Sharing. De Cock et al. [16] follow the same blueprint as the two previous scheme by first comparing each threshold with the corresponding attribute. In contrast to all other protocols (ours included), which are secure in the computational setting, they operate in the information theoretic model using secret sharing (SS) based SMC and utilize commodity-based cryptography [5] to reduce the number of interactions. This results in a protocol that performs better than all other protocols (ours included) for small trees. However, their protocol is less efficient for large trees, since it is also linear in the size of the tree.

Secure Computation Frameworks. Despite being a powerful tool for privacy-preserving application, SMC incurs large overheads and requires expert knowledge to develop efficient protocols. Therefore a variety of frameworks (the software tools implementing the underlying - or a combination of - generic SMC protocol(s)) has been developed to address these problems. There exist three predominant approaches. Some frameworks are tailored for Garbled Circuits e.g., FairPlay [50], FairPlayMP [7], LEGO [55], TASTY [34], SCAPI [25], JustGarble [6], CBMC-GC [26] or ObliVM [49]. Among the GC frameworks, some (such as [26, 49]) can transform Java, C/C++ programs into GC, other tools can run the garbling process and execute the generated GCs [6, 25, 49]. The second approach of frameworks is based on secret sharing, e.g., Sharemind [10], VIFF [19], SPDZ [21], SPDZ-2 [20], TinyOT [13] or MASCOT [39]. They allow to share the parties' inputs among many computing parties using additive secret sharing. Addition of secret values can be evaluated by just adding the shares locally. Multiplication requires interactions and can be speed up using a pre-processing based on Beaver technique [5]. So called multiplication triples are generated in an offline phase and later use in the online protocol to reduce the number of interactions. Finally, the third approach consists of hybrid frameworks that combine GC, additively HE and secret sharing, e.g., ABY [23].

Symbol	Interpretation
l	Bit length of attribute values
n	Dimension of the attribute vector
$x = x_0, \ldots, x_{n-1}$	Attribute vector
M	Number of nodes
m	Number of decision nodes
d	Depth of the decision tree
$\mathcal{N} = (w, vl, vr, i)$	Node representation (Def. 5.1)
$N = N_0, \ldots, N_{M-1}$	Array of nodes (Def. 5.1)
$\langle v \rangle$	Secret sharing of a variable v
v_C	Client's share of a variable v
v_S	Server's share of a variable v
$[\![v]\!]$	Additively HE of v

Table 1. Notations.

Scheme	Rounds	Tools	Communication	Comparisons.
[12]	≈ 5	HE+GC	$O(M)$	d
[4]	≈ 4	HE+GC	$O(m)$	d
[11]	≥ 6	FHE/SHE	$O(m)$	m
[66]	6	HE+OT	$O(m)$	m
[60]	4	HE	$O(m)$	m
[16]	≈ 9	SS	$O(M)$	m
This Work	$4d$	GC,OT	$O(M)$	d
	$d^2 + 3d$	ORAM [62]	$O(d^4)$	
	$4d$	ODS [64]	$O(d^3)$	
	$4d$	FLORAM [24]	$O(d^2)$	

Table 2. Summary of private decision tree evaluation protocols.

Summary. We summarize the properties of private decision trees protocols in Table 2. Already from the table we can conclude that our protocol has the best asymptotic communication complexity. Since the size of the tree is in the worst case exponential in the depth of the tree d, $M = O(2^d)$ and $m = O(2^d)$, we can expect our protocol to outperform alternative approaches for large trees. However, we also aim to improve the practical communication cost and computation time. Hence, we

compare our implementation to the protocol of Wu et al. [66] on relevant, real-world data sets from the UCI repository. We chose Wu et al. because they perform an extensive comparison to the other protocols and have the best performance in the computational, two-party setting. The results are summarized in our evaluation in Section 8.

3 Preliminaries

We begin by clarifying our problem statement and by introducing relevant cryptographic tools but refer the interested reader to the bibliographical references.

3.1 Problem Definition

We consider a client that holds a private attribute vector and a server holding a private decision tree classifier. Each internal node of the decision tree contains a threshold value and an integer value that indexes an element of the attribute vector. Leaf nodes contain only the corresponding classification label. To evaluate the tree on the client input, we traverse the tree beginning from the root by comparing the threshold of the current node with the corresponding attribute value. Depending on the result of the comparison, we move either to the left or right subsequent node. Once we reach a leaf node, we return its classification label as result of the computation. To respect the privacy of the inputs, the computation has to be done in a secure way.

We make the following assumption for our scheme. We assume the elements of the attribute vector and the threshold to be l-bit integers. As in previous work [16, 66], we assume the tree to be complete. Note that the evaluation of a non-complete tree may leak information to both parties on the opponent's input: For example, if the evaluation ends after a number of iteration smaller than the depth of the tree, the server might infer the classification result. In addition, the client also learns information about the structure of the tree. If the tree is not complete, we can insert internal dummy nodes. In particular, all leaves in the subtree of a dummy node have the same classification label.

The basic insecure decision tree evaluation algorithm is sublinear in the size of the tree. To ensure privacy most previous protocols evaluate all decision nodes by comparing each node's threshold with the corresponding attribute value, yielding a linear complexity in the size of the tree. Our goal is to evaluate only the nodes that lead to the correct classification label and to rely as much as possible on symmetric cryptography. This provides a significant boost in performance.

3.2 Garbled Circuits

Yao's initial protocol for secure two-party computation uses a technique called garbled circuits (GC). In GC protocols, a party called Generator garbles a Boolean circuit representing the function to be computed and sends it with the keys corresponding to its input to a second party called Evaluator. Then both parties engage in an Oblivious Transfer protocol, that allows the Evaluator to learn the keys corresponding to its inputs without revealing any information on the actual inputs to the Generator. Finally, the Evaluator evaluates the GC and outputs the result. For a detailed, technical description of circuit garbling and its implementation see [25, 47–49, 57, 68].

3.3 Oblivious Transfer

Oblivious Transfer (OT) is a fundamental cryptographic protocol that allows a receiver to choose 1 out of n values held by a sender, without revealing to the sender which value was chosen, while revealing to the receiver only the chosen value [29]. In a 1-out-of-2 OT (OT_2^1), the sender has two inputs x_0 and x_1 and the receiver has an index $i \in \{0, 1\}$. After the protocol the receiver learns only x_i and the sender learns nothing. There are several efficient OT protocols including [53]. Nevertheless, all these schemes require some public key operations. In [36] the authors describe a hybrid cryptographic technique to extend a constant number of OTs to polynomially many OTs using symmetric cryptography. Recently, the OT extension protocol was improved by [2, 3, 38]. There are also number of 1-out-of-n OT protocols (OT_n^1). For our protocol, we rely on the OT_n^1 by [54] that uses $\log(n)$ OT_2^1 protocols to realize OT_n^1.

3.4 Oblivious RAM

Oblivious RAM (ORAM) is a cryptographic primitive that allows a client to outsource its encrypted storage to an untrusted server and to hide the data access patterns of the client from the server [28, 30]. Efficient ORAM schemes replace each read/write access to the

original storage with a randomized series of read/write accesses. Many schemes have improved the performance of ORAM including [62], which we have used in our implementation. The idea of using ORAM in secure computation was first mentioned in [31, 63].

Our implementation relies on Circuit ORAM [62], which is a tree-based ORAM. Tree-based ORAM was first introduced by [59] and has been improved by many other works, including [62]. Tree-based ORAM schemes have sublinear complexity. For a database with n elements Circuit ORAM [62] has $O(\log^3(n))$ costs. It has been implemented in the ObliVM framework [49]. In Tree-based ORAM a position map stores for each block the index of the path, where this block lies. It can be stored recursively on the server to reduce the client storage. Tree-Based ORAMs differ mostly in their eviction strategy but adopt a similar data structure. We refer to [59, 62] for more details.

Another relevant concept also related to ORAM are Oblivious Data Structures (ODS) [64]. ODS ensure that for any two sequences of k operations to the data structure, their resulting access patterns are indistinguishable. They apply to data structures such as binary trees or heaps with sparse access graphs. For such data structures, [64] introduces a pointer-based technique. Thereby, the key idea is to store each node with the indexes and labels of its child nodes. This eliminates the need to search through the position map (and hence the recursion step for tree based ORAM) for a child node label, achieving $O(\log^2(n))$ cost. Using similar techniques, Keller and Scholl [40] proposed schemes for ODS based on secret sharing . Their schemes are very efficient, translate to the multiparty setting and provide active security as they fit naturally with the SPDZ framework, which uses a pre-processing phase to generate secret random multiplication triples and random bits. The resulting online protocol is actively secure against a dishonest majority.

In [24] Doerner and Shelat introduced FLORAM an ORAM scheme based on function secret sharing, a primitive that allows sharing a function f to $p >= 2$ parties such that on input x, each party learns a share $f_i(x)$ and $\sum f_i(x) = f(x)$. Despite its linear computation complexity, the access time is better than tree-based ORAM for trees up to 2^{30} large. The communication is $O(log(n))$ for a database with n elements. Most importantly, the initialization does not require secure computation and is therefore very fast. For instance, an ORAM with 2^{20} 4-byte elements can be initialized in less than 200 milliseconds. However, the security model requires two non-colluding servers.

4 Definitions

In this section we introduce relevant definitions and notations for our scheme. Our definitions and notations (Table 1) are similar to previous work [16, 66]. With $[a, b]$, we denote the set of all integer from a to b. Let $c_0, \ldots, c_{k-1}$ be the classification labels, $k \in \mathbb{N}_{>0}$.

Definition 4.1 (Decision Tree)**.** *A decision tree (DT) is a function $\mathcal{T} : \mathbb{Z}^n \to \{c_0, \ldots, c_{k-1}\}$ that maps an n-dimensional attribute vector $x = (x_0, \ldots, x_{n-1})$ to a finite set of classification labels. The tree consists of:*
- *internal nodes (decision nodes) containing a test condition and*
- *leave nodes containing a classification label.*

A decision tree model consists of a decision tree and the following functions:
- *a function $\mathbf{t}$ that assigns to each decision node a threshold value, $\mathbf{t} : [0, m-1] \mapsto \mathbb{Z}$,*
- *a function $\mathbf{a}$ that assigns to each decision node an attribute index, $\mathbf{a} : [0, m-1] \mapsto [0, n-1]$, and*
- *a labeling function $\mathbf{c}$ that assigns to each leaf node a label, $\mathbf{c} : [m, M-1] \mapsto \{c_0, \ldots, c_{k-1}\}$.*

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 $[x_{\mathbf{a}(v)} \geq \mathbf{t}(v)]$.

Definition 4.2 (Node Indices)**.** *Given a decision tree, the index of a node is its order as computed by breadth-first search (BFS) traversal, starting at the root with index 0. If the tree is complete, then a node with index v has left child $2v+1$ and right child $2v+2$.*

We will also refer to the node with index v as the node v. W.l.o.g, we will use $[0, k-1]$ as classification labels (i.e., $c_j = j$ for $0 \leq j \leq k-1$) and we will label the first (second, third, ...) leaf in BFS traversal with classification label 0 (1, 2, ...). For a complete decision tree with depth d the leaves have indices ranging from $2^d, 2^d+1, \ldots 2^{d+1}-2$ and classification labels ranging from $0, \ldots, 2^d-1$ respectively. Since the classification labeling is now independent of the tree, we use $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$ to denote a *decision tree model* consisting of a tree $\mathcal{T}$ and the labeling functions $\mathbf{t}, \mathbf{a}$ as defined above. We also assume that the tree parameters d, m, M can be derived from $\mathcal{T}$.

Definition 4.3 (Decision Tree Evaluation)**.** *Given $x = (x_0, \ldots, x_{n-1})$ and $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$, then starting at the root, Decision Tree Evaluation (DTE) evaluates at each reached node v the decision $b \leftarrow [x_{\mathbf{a}(v)} \geq \mathbf{t}(v)]$ 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 4.4 (Private DTE)**.** *Given a client with a private $x = (x_0, \ldots, x_{n-1})$ and a server with a private $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$, 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., $(\mathcal{M}, x) \mapsto (\emptyset, \mathcal{T}(x))$.*

Definition 4.5 (Correctness)**.** *Given a client with a private $x = (x_0, \ldots, x_{n-1})$ and a server with a private $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$, a protocol Π correctly implements a PDTE functionality if after the computation it holds for the result c obtained by the client that $c = \mathcal{T}(x)$.*

Besides correctness parties must learn only what they are allowed to. To formalize this, we need the following two definitions. A function $\mu : \mathbb{N} \to \mathbb{R}$ is *negligible* if for every positive polynomial $p(.)$ there exists an ϵ such that for all $n > \epsilon$: $\mu(n) < 1/p(n)$. Two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are *computationally indistinguishable* (denoted $\mathcal{D}_1 \stackrel{c}{\equiv} \mathcal{D}_1$) 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 [29]. The protocol is said to be secure if for each party, one can construct a simulator that, given only the input of that party and the output, can generate a distribution that is computationally indistinguishable to the party's view.

Definition 4.6 (Semi-Honest Security)**.** *Given a client with a private $x = (x_0, \ldots, x_{n-1})$ and a server with a private $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$, a protocol Π securely implements the PDTE functionality in the semi-honest model if the following conditions hold:*

- *there exists a PPT algorithm SIM_S that simulates the server's view $view_S^{\Pi}$ given only the private decision tree model $(\mathcal{T}, \mathbf{t}, \mathbf{a})$ such that:*

$$SIM_S(\mathcal{M}, \emptyset) \quad \stackrel{c}{\equiv} \quad view_S^{\Pi}(\mathcal{M}, x), \tag{1}$$

  and
- *there exists a PPT algorithm SIM_C that simulates the client's view $view_C^{\Pi}$ given only the depth d of the tree, $x = (x_0, \ldots, x_{n-1})$ and a classification label $\mathcal{T}(x) \in \{0, \ldots, k-1\}$ such that:*

$$SIM_C(\langle d, x \rangle, \mathcal{T}(x)) \quad \stackrel{c}{\equiv} \quad view_C^{\Pi}(\mathcal{M}, x). \tag{2}$$

Before moving to our main construction, we describe a primitive called oblivious Array Indexing (OAI), which will serve as a sub-protocol. Private array indexing allows a party C holding a private index i to privately access the i-th element of an array held by a party S. We will use array indexing as an intermediate step, and therefore execute it obliviously such that the index and the indexed element are secret-shared to the parties.

Definition 4.7 (Oblivious Array Indexing)**.** *Let $\boldsymbol{A} = [A_0, \ldots, A_{n-1}]$ be an array and $i \in [0, n-1]$ be an index. An Oblivious Array Indexing (OAI) functionality consists of:*

- *a party S holding privately $\boldsymbol{A}$ and a share i_S of i,*
- *a party C holding privately another share i_C of i.*

The functionality computes two shares A_{i_S} and A_{i_C} of A_i and returns them to party S and C respectively, i.e,

$$((\boldsymbol{A}, i_S), i_C) \mapsto (A_{i_S}, A_{i_C}),$$

such that if $i = i_S \odot i_C$ then $A_i = A_{i_S} \odot A_{i_C}$, where $\odot \in \{\oplus, +\}$.

5 Our PDTE Protocol

In this section, we present our scheme and discuss its correctness and security.

5.1 Intuition

At first, the server transforms the decision tree into an array which is formalized in the following definition.

Definition 5.1 (Data Structure)**.** *Let $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$ with M and m as above. A node data structure (DS) of a decision node $v \in [0, m-1]$ consists of the tuple $(\mathbf{t}(v), \textsc{Left}(v), \textsc{Right}(v), \mathbf{a}(v))$, where $\textsc{Left}(v)$ and $\textsc{Right}(v)$ are indices of left and right child node of v in BFS traversal. A node DS of a leaf node $v \in [m, M-1]$ consists of the tuple $(\mathbf{c}(v), \textsc{Null}, \textsc{Null}, \textsc{Null})$. A tree DS consists of the array $\boldsymbol{N} = N_0, \ldots, N_M$ such that N_v is the node DS of $v \in [0, M-1]$.*

Hence, besides a threshold and an index to x, each node stores the index to its two child nodes, similar to the pointer-based technique of [64]. The tree DS $N_0, \ldots, N_M$ is equivalent to $\mathcal{M} = (\mathcal{T}, \mathbf{t}, \mathbf{a})$ and will be used instead.

The server secret-shares the root node with the client and the protocol loops d times. In each iteration, the parties select the attribute value that corresponds to

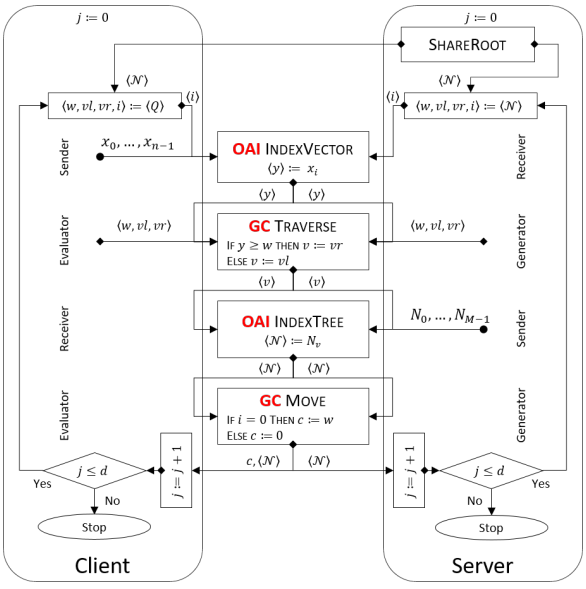


Fig. 1. Illustration of the Protocol: The attribute vector x is indexed with i (i.e., $y = x_i$). The array of tree nodes N is indexed with v (i.e., $\mathcal{N} = N_v, N_0$ is the root node).

the current node (initially the root). Then, they execute a GC to compute an index which they use to obliviously select one child node of the current node. Finally, they execute another GC to check if the selected child node is a leaf. Otherwise, this node is evaluated in the next iteration. An overview is illustrated in Figure 1, where each OAI block can be instantiated with any protocol that allows secret indexing, such as GC, OT, ORAM. Each GC block can be implemented with any generic secure 2-party computation approach.

5.2 Algorithms

In our algorithms, we use v to denote the index of a node, i.e., T_v. Moreover, we use $\mathcal{N} = (w, vl, vr, i)$ to denote the content of nodes. If it is a decision node then $w = \mathbf{t}(v), vl = \textsc{Left}(v), vr = \textsc{Right}(i)$ and $i = \mathbf{a}(v)$. Otherwise $w = \mathbf{c}(v)$ and $vl = vr = i = \textsc{Null}$. For a variable V that is secret-shared between server and client, let V_S and V_C denote the respective shares, i.e., $\mathcal{N}_S = (w_S, vl_S, vr_S, i_S)$ denotes the server's share of $\mathcal{N}$. In our GCs, we use the following basic operations:

- $\textsc{And}(b_1, b_2) = (b_1 \wedge b_2)$,
- $\textsc{Eq}(a_1, a_2) = (\textsc{If}\ a_1 = a_2\ \textsc{Then}\ 1\ \textsc{Else}\ 0)$,
- $\textsc{Geq}(a_1, a_2) = (\textsc{If}\ a_1 \geq a_2\ \textsc{Then}\ 1\ \textsc{Else}\ 0)$,
- $\textsc{Mux}(b, a_1, a_2) = (\textsc{If}\ b\ \textsc{Then}\ a_2\ \textsc{Else}\ a_1)$,
- $\textsc{Not}(b) = (\neg b)$, and
- $\textsc{Xor}(a_1, a_2) = (a_1 \oplus a_2)$.

Additionally, our algorithms use a combination of GC and simple secret sharing. We use the following operations to denote re-sharing of a value in a GC:

- $\textsc{Reshare}(r, a) = (r, a \odot r)$, where $\odot \in \{\oplus, +\}$,
- $\textsc{Reshare}(\vec{r}, \vec{a}) = (\vec{r}, \vec{a} \odot \vec{r})$, where $\vec{a}, \vec{r}$ are vectors and $\odot$ is applied componentwise.

The value to be re-shared (a or $\vec{a}$) is an intermediate result (e.g., the index of the subsequent node), which should not be revealed in clear to the parties. The value r or $\vec{r}$ is randomly chosen by the GC generator and used to blind an intermediate result.

The Evaluation Algorithm. Fig. 2 shows our PDTE scheme. It always starts at the root, i.e., the first element of the tree DS. Let the initial current node $\mathcal{N} = (w, vl, vr, i)$ be the node DS of the root. With the procedure ShareRoot, the server first shares the root node by choosing a random $\mathcal{N}_S = (w_S, vl_S, vr_S, i_S)$ and sending $\mathcal{N}_C = \mathcal{N} \oplus \mathcal{N}_S = (w \oplus w_S, vl \oplus vl_S, vr \oplus vr_S, i \oplus i_S)$ to the client. The values $\mathcal{N}_S, \mathcal{N}_C$ are used as shares of the current node in the first iteration of the following loop. In each iteration the parties successively perform the following operation:

- Select the corresponding attribute value (Step 5): Using the procedure IndexVector, the parties then execute an oblivious array indexing (see Section 6) to select and secret-share an array element. The server's input is its share of i and the client's input consist of its share of i and the attribute vector x. The result of Step 5 consists of two shares of the attribute value corresponding to the current decision node.
- Compute the index of the next node (Step 7): The parties execute a GC which evaluates the current node (i.e., compares the current attribute value with the current node's threshold) and uses the comparison result to select the index of the next node in the tree. The resulted index is secret-shared to the parties. This step is fully specified in Fig. 3.
- Select the next node in the tree (Step 8): The parties use their secret shares from the previous step as input to the procedure IndexTree to select and secret-share the next node in the tree. The additional input to the array indexing operation is the array. The result is used to update the current node.
- Move (Step 10): Finally, another GC, which is also fully specified in Fig. 4, checks if the current node $\mathcal{N}$ is a leaf and computes the classification label and re-shares the index i. Note that, as we assumed a complete tree, it is enough to perform this check only for the last iteration. In this case the Move

Input : $(\boldsymbol{N}, x)$
Output : $(\emptyset, c)$

1: $(\mathcal{N}_S, \mathcal{N}_C) \leftarrow$ShareRoot$(root, \emptyset)$
2: $j \leftarrow 1$
3: **while** $j \leq d$ **do**
4: $(w_p, vl_p, vr_p, i_p) \leftarrow \mathcal{N}_p$ ▷ $p = S, C$
5: $(y_S, y_C) \leftarrow$IndexVector$(i_S, (x, i_C))$
6: $\mathcal{N}_p \leftarrow (w_p, vl_p, vr_p, y_p, r_p)$ ▷ $p \in \{S, C\}, r_C = \emptyset$
7: $(v_S, v_C) \leftarrow$Traverse$(\mathcal{N}_S, \mathcal{N}_C)$
8: $(\mathcal{N}_S, \mathcal{N}_C) \leftarrow$IndexTree$((\boldsymbol{N}, v_S), v_C)$
9: $(w_p, vl_p, vr_p, i_p) \leftarrow \mathcal{N}_p$ ▷ $p = S, C$
10: $((0, i_S), (c, i_C)) \leftarrow$Move$((w_S, i_S), (w_C, i_C, r_C))$
11: $j \leftarrow j + 1$
12: **end while**
13: **return** $(\emptyset, c)$

Fig. 2. Our Private Decision Tree Evaluation Protocol

Input : $((w_S, vl_S, vr_S, y_S, r_S), (w_C, vl_C, vr_C, y_C))$
Output : (v_S, v_C)

1: $\mathcal{N}_p \leftarrow (w_p, vl_p, vr_p, y_p)$ ▷ $p = S, C$
2: $(w, vl, vr, y) \leftarrow$Xor$(\mathcal{N}_S, \mathcal{N}_C)$
3: $b \leftarrow$Geq(y, w)
4: $v \leftarrow$Mux(b, vl, vr)
5: $(v_S, v_C) \leftarrow$Reshare(r_S, v)

Fig. 3. Traverse Algorithm

Input : $((w_S, i_S), (w_C, i_C, r_C))$
Output : $((\text{Null}, i_S), (R, i_C))$

1: $(w, i) \leftarrow$Xor$((w_S, i_S), (w_C, i_C))$
2: $b \leftarrow$Eq(i, Null)
3: $R \leftarrow$Mux$(b, 0, w)$
4: $(i_C, i_S) \leftarrow$Reshare(r_C, i)

Fig. 4. Move Algorithm

algorithm only needs to re-share the index i for all iterations but the last one.

The algorithm loops d times and $\mathcal{N}$ is evaluated in the next iteration. Steps 4, 6, 9, and 11 are local steps and do not require any secure computation.

We now turn to the description of the algorithms Traverse and Move, which are implemented using Yao's garbled circuits approach.

The Traverse Algorithm. Fig. 3 describes the GC that is used in Fig. 2 (Step 7) to compute the index of the next node. The GC is generated by the server and evaluated by the client. Each party inputs its share of (w, vl, vr, y) which is then recovered (Step 2) in the execution of the GC. Then the result of the comparison $y \geq w$ (Step 3) is used to select (Step 4) the index of the next node between vl and vr. The selection is implemented using an $l-$bit multiplexer Mux(b, a_1, a_2). Finally, this index is secret-shared (Step 5) and returned to the parties. Note that, the output v_S is identical to the random string r_S that is also part of the server's input, such that in Step 5, $v_C \leftarrow v \odot r_S$ is computed and returned only to the client.

The Move Algorithm. At the end of each iteration in Fig. 2, the GC of Fig. 4 receives shares of (w, i) and recombines them (Step 1) in the secure computation. Then it checks (Step 2) if i equals Null and returns either 0 or w. Again, as we assumed a complete tree, it is enough to perform this check only for the last iteration. The selection (Step 3) is also implemented as an $l-$bit multiplexer. The final step re-shares the index i to both parties. For this GC as well, the server and the client act as generator and evaluator respectively.

We stress that the algorithms in Fig. 3 and 4 represent the full specification of the underlying GCs that are neither dependent on the position in the tree nor on its size. It is straightforward to see that the resulting GCs are indeed small. Their cost depends on the operations Eq, Geq, Mux, and Reshare, as the performance metric for GC is the number of AND-gates (XOR being free). Assuming inputs are l-bit integers, operations Eq, Geq, Mux contain each exactly l AND-gates [42, 43], which results in $4l$ ciphertexts each. The Reshare operation contains AND-gates only when the sharing is additive. In this case re-sharing is implemented as full-adder with l AND-gates [42]. The *halfGate* optimization [69] reduces the number of ciphertexts per AND-gate by a factor of 2 at the cost for the evaluator to perform two cheap symmetric operations, rather than one. Also, notice that the size of Eq and Geq depends on the size of the attribute vector n and the tree M respectively, whose bit-length can be much more smaller than the input's bit-length in practice (i.e., $\log(n)$ and $\log(M)$ are smaller than l). In the worst case, the communication cost of our scheme (Fig. 2) is dominated by the cost of the array indexing on the tree (in general the tree is larger than the attribute vector) which is $O(M), O(M), O(d^3), O(d^2), O(d)$ for GC, OT, Circuit ORAM, ODS and FLORAM respectively.

Input : $((A_0, \ldots, A_{n-1}, i_S, r_S), i_C)$
Output : (a_S, a_C) s.t. if $i = i_S \oplus i_C$ then $A_i = a_S \oplus a_C$

1: $i \leftarrow$XOR(i_S, i_C)
2: $a_C \leftarrow 0$
3: **for** $j = 0$ **to** $n-1$ **do**
4: $\quad b \leftarrow$EQ(i, j)
5: $\quad a_C \leftarrow$MUX(b, a_C, A_j)
6: **end for**
7: $(a_S, a_C) \leftarrow$RESHARE(r_S, a_C)

Fig. 5. OAI with Garbled Circuit

Input : $([A_0, \ldots, A_{n-1}], i)$
Output : $(\emptyset, A_i)$

1: S chooses F_K and pairs $(K_0^0, K_0^1), \cdots, (K_{h-1}^0, K_{h-1}^1)$
2: $\forall \iota \in [0, n-1]$: S sends $c_\iota = A_\iota \oplus \oplus_{j=0}^{h-1} F_{K_j^{\iota_j}}(\iota)$ to C
3: $\forall j \in [0, h-1]$: $(\emptyset, K_j^{i_j}) \leftarrow \mathrm{OT}_2^1((K_j^0, K_j^1), i_j)$
4: $A_i \leftarrow$ DECRYPT$((K_0^{i_0}, \ldots, K_{h-1}^{i_{h-1}}), c_i)$

Fig. 6. Naor and Pinkas OT_n^1 Protocol

Input : $((A_0, \ldots, A_{n-1}, r, s), k = i + r \bmod n)$
Output : (a_S, a_C) s.t. $A_i = a_S \oplus a_C$

1: For all $0 \le i < n$, S sets $A'_{j+r \bmod n} \leftarrow A_j \oplus s$
2: S and C execute $(\emptyset, A'_k) \leftarrow \mathrm{OT}_n^1(A', k)$
3: $a_S \leftarrow s$ and $a_C \leftarrow A'_k = A'_{i+r \bmod n} = A_i \oplus s$

Fig. 7. OAI with Oblivious Transfer

6 Implementing OAI

OAI can be instantiated with any protocol that allows secret indexing, such as GC, OT, ORAM. Notice that, OAI only makes sense when used as a sub-protocol. In the overall protocol the array indexing is always preceded by a GC step that computes and returns the shares i_S and i_C to server and client (i.e., in Step 5 of TRAVERSE and in Step 4 of MOVE). To initially index the attribute vector the shares are computed by the server in Step 1 of Fig. 2. This is not a security problem as the classification always starts at the root.

As already mentioned, we assume the array elements and the indices to be $l-$bit integers. In this section, we refer to party S and party C as *Sender* and *Receiver* respectively.

6.1 OAI with Garbled Circuits

The GC for indexing is described in Fig. 5. The circuit uses three sub-circuits: XOR, EQ, MUX. The input of the sender also contains a random string r_S that is used in line 7 to randomize the output. The algorithm scans the array and uses MUX to select the indexed element. Since the sender is also the generator of the GC, the evaluator does not have to send back the result of the evaluation. Using the same metric explained in Section 5.2, the GC of Fig. 5 contains $l(n+1)$ AND-gates. The communication cost is clearly linear in the array size, i.e., $O(n)$.

Alternatively, we can also represent the index i as a vector of bits with 0 everywhere except at position i. This results in a more efficient GC by getting rid of the equality check in line 4, which requires l AND-gates. However, since we use l bits to represent the index, we run this more efficient alternative only when the size of the array is smaller or equal to l.

6.2 OAI with Oblivious Transfer

For the oblivious indexing with OT, we use the OT_n^1 protocol by Naor and Pinkas, which we briefly review first and refer to [54] for details.

Let $A_0, \ldots A_{n-1}$ be an array, i an index, $h = \log(n)$ and F_K a PRF. Let ι be an index in $[0, n-1]$ and $\iota_0 \ldots \iota_{h-1}$ its bit representation. Then sender S and receiver C perform $O(n)$ and $O(\log(n))$ operations respectively. The scheme is described in Fig. 6.

The idea of OAI with OT is to share the array index i with *additive sharing* by choosing a random r as sender's share and $i + r$ as receiver's share. Then the sender rotates the array by r and the parties execute OT_n^1 on the new array and $i + r$. Hence, let $r \in \{0,1\}^l$ be a random $l-$bit integer and $i_S = r$ and $i_C = i + r$. The protocol is described in Fig. 7.

In [37], Jarrous and Pinkas used a similar idea in a protocol called HDOT (Hamming Distance based OT) in which parties C and S have private binary strings $\alpha = \alpha_0 \ldots \alpha_t$ and $\beta = \beta_0 \ldots \beta_t$ of length $t = \log(n)$. The sender S additionally has a private dataset $\boldsymbol{A} = [A_0, \ldots, A_{n-1}]$. The parties then run a protocol that privately computes the Hamming distance $i = d_H$ and uses it to reveal A_i to C using OT_n^1. To select A_i without revealing d_H to the parties, C first computes $[\![\alpha]\!] = ([\![\alpha_0]\!], \ldots, [\![\alpha_t]\!])$ under its public key using an additively HE. Then S receives $[\![\alpha]\!]$ and computes $[\![i]\!] = \prod [\![\alpha_j \oplus \beta_j]\!] = [\![\sum \alpha_j \oplus \beta_j]\!]$. Finally, S chooses a random r and sends $[\![i + r]\!]$ to C.

Alternatively, we can use the following proposition to share the index with *exclusive-OR sharing* instead.

The benefit is that the re-sharing of the index in the GC (e.g., in Step 5 of Fig. 3) is more efficient as it requires only exclusive-OR operations. In our implementation, we use this alternative, whenever it is possible.

Proposition 6.1. *Let $B = \{0,1\}^l$ and $x \in B$ then we have $x \oplus B = B$.*

Proof. $\forall y \in B, x \oplus y$ is clearly in B, since B contains alls $l-$bit strings. Moreover, $\forall y_1, y_2 \in B$ with $y_1 \neq y_2$ it holds $x \oplus y_1 \neq x \oplus y_2$, because otherwise $x \oplus y_1 = x \oplus y_2$ implies $y_1 = y_2$. □

From the above proposition, it follows that if the length n of the array is a power of 2 with $\log(n) = h$, then $[0, n-1]$ is isomorphic to $\{0,1\}^h$. As a result, we can modify the algorithm in Fig. 7 as follows: we set $i_S = r$ and $i_C = i \oplus r$, and replace Step 1 by $A'_{j \oplus r} \leftarrow A_j \oplus s$, $0 \leq i < n$. If n is not a power of 2, one may consider padding the array with 0s to an array of length $n' = 2^{h+1}$. However, since the OT_1^n requires sending n ciphertexts (resulting in $O(n)$ communication cost) we will use the alternative protocol only if n is a power of 2.

6.3 OAI with Oblivious RAM

For an array of size N, tree-based ORAM organizes the data into blocks stored in a binary tree of height $\log(N)$ at the server. Each node of the tree is a bucket of $\log(N)$ blocks. Each block has the form $\{\texttt{idx}\|\texttt{label}\|\texttt{data}\}$, with `idx` a block index, `label` a leaf identifier specifying the path on which the block resides and `data` the actual data. The client stores a stash STASH for buffering overflowing blocks and a position map POSMAP mapping `idx` to `label`. The position map can be reduced to $O(1)$ by recursively storing it in smaller ORAMs at the server. There are two basic operations. The first one, READANDRM, reads and removes a block from its current position and the second one, EVICT, randomly pushes blocks down to their path. They are used to implement the ACCESS operation (Algorirthm 8 [62]) which allows two functionalities: ORAM.READ(idx):= ORAM.ACCESS(idx, $\emptyset$) and ORAM.WRITE(idx, data):= ORAM.ACCESS(idx, data) [59, 62].

OAI with ORAM requires only READ(). In an initial step, the server places the array in an ORAM, and secret-shares the resulting ORAM with the client. OAI with ORAM (Fig. 9) is similar to the case of GC, with the only difference that the array is stored in the shared ORAM and the computation of the indexed element is replaced by ORAM.READ(IDX). In fact, the read and write operations are implemented using GCs. In our implementation we used the recursive circuit ORAM [62]. However, OAI with ORAM can be optimized by using oblivious data structures [40, 64] or FLORAM [24] instead. Using ODS is straightforward since ODS schemes are basically optimized tree-based ORAM like circuit ORAM. OAI with FLORAM requires to implement the PDTE as a 3-party protocol (one client and two servers). Because of our 2-party security model, we see the integration of FLORAM to our scheme as a future work.

Input: (idx, data)
Output: out

1: $\{\texttt{idx}\|\texttt{lbl}\|\texttt{out}\} \leftarrow$ READANDRM(idx, POSMAP[idx])
2: POSMAP[idx] $\leftarrow$ UNIFORMRANDOM$(0, \ldots, N-1)$
3: **if** (data $= \emptyset$) **then** data $\leftarrow$ out
4: STASH.ADD$(\{\texttt{idx}\|\text{POSMAP}[\texttt{idx}]\|\texttt{data}\})$
5: EVICT()
6: **return** out

Fig. 8. Algorithm ORAM.ACCESS

Input: $((ORAM_S, i_S, r_S), (ORAM_C, i_C))$
Output: (a_S, a_C) s.t. if $i = i_S \oplus i_C$ then $A_i = a_S \oplus a_C$

1: $i \leftarrow$XOR(i_S, i_C)
2: $a_C \leftarrow$ ORAM.READ(i)
3: $a_C \leftarrow$XOR(A_{i_C}, r_S)
4: $a_S \leftarrow r_S$

Fig. 9. OAI with ORAM

7 Optimizations

We now discuss some optimizations of our scheme.

7.1 Level Indexing

In each iteration, our protocol executes array indexing on the tree to select the index of the next node. Since this node is always a child node of the current node, we do not have to use the whole tree during array indexing. It is sufficient to use only the nodes of the next level. Therefore, before invoking INDEXTREE (Step 8 of Fig. 2) at level d', we construct an array containing only the nodes of level $d'+1$, where for each node vl and vr are recomputed according to the number of nodes at level $d'+2$. For ORAM, each level of the tree is stored in its own ORAM during initialization. Of course, this reveals

Input : $((w_S, i_S, u_S, e_S, \vec{r}_S), (w_C, i_C, u_C, e_C))$
Output : $((i'_S, u'_S, e'_S), (i'_C, u'_C, e'_C))$

1: **function** FSTCALL
2: $(w, i) \leftarrow \text{XOR}((w_S, i_S), (w_C, i_C))$
3: $b \leftarrow \text{EQ}(i, \text{NULL})$
4: $(u, e) \leftarrow \text{MUX}(b, (\text{NULL}, 0), (w, 1))$
5: $v' \leftarrow (i, u, e)$
6: $((i'_S, u'_S, e'_S), (i'_C, u'_C, e'_C)) \leftarrow \text{RESHARE}(\vec{r}_S, v')$
7: **end function**
8: **function** ITHCALL
9: $v_p \leftarrow (w_p, i_p, u_p, e_p)$ ▷ $p = S, C$
10: $(w, i, u, e) \leftarrow \text{XOR}(v_S, v_C)$
11: $b_1 \leftarrow \text{EQ}(e, 1)$, $b_2 \leftarrow \text{EQ}(i, \text{NULL})$
12: $b \leftarrow \text{AND}(\text{NOT}(b_1), b_2)$
13: $(u, e) \leftarrow \text{MUX}(b, (u, e), (w, 1))$
14: $v' \leftarrow (i, u, e)$
15: $((i'_S, u'_S, e'_S), (i'_C, u'_C, e'_C)) \leftarrow \text{RESHARE}(\vec{r}_S, v')$
16: **end function**

Fig. 10. Garbled Circuit MOVE For Sparse Trees

the number of nodes per level to the client. However, if we assume that the tree is complete, then it is not a new leakage. Otherwise, we can still extend all levels to the size of the longest level, which is smaller than 2^d, leaking only this longest size. To be secure for sparse trees, this optimization has to be combined with the *Handling Sparse Trees* optimization below. We note that for efficiency reasons, previous schemes also leak either the number of decision nodes, the number of nodes, the number of paths or the depth of the tree to the client.

7.2 Pre-processing the Vector Indexing

One can also avoid indexing the attribute vector x in each step. Let $x = (x_0, \ldots, x_{n-1})$, then in an initial step, the client computes $[\![x]\!] = ([\![x_0]\!], \ldots, [\![x_{n-1}]\!])$ under its public key using additively HE. The server receives $[\![x]\!]$, chooses n random numbers $r_0, \ldots, r_{n-1} \in \{0, \ldots, 2^{l+\sigma}\}$ (σ is a security parameter that determines the statistical leakage [22], e.g., $\sigma = 32$) and computes $([\![x_0 + r_0]\!], \ldots, [\![x_{n-1} + r_{n-1}]\!])$. Then the server chooses a random permutation π and sends back $\pi([\![x_0 + r_0]\!], \ldots, [\![x_{n-1} + r_{n-1}]\!])$ to the client. Finally, the server replaces each decision node $\mathcal{N} = (w, vl, vr, i)$ of the tree by $\mathcal{N} = (w + r_{\pi(i)}, vl, vr, \pi(i))$. Note that, we can reduce the number of ciphertexts sent, by packing many plaintexts in one ciphertext [58]. During the evaluation, the client learns $\pi(i)$ in each iteration, selects the attribute value $x_{\pi(i)} + r_{\pi(i)}$ locally and uses it in the TRAVERSE algorithm which evaluates $\text{GEQ}(x_{\pi(i)} + r_{\pi(i)}, w + r_{\pi(i)})$ instead of $\text{GEQ}(x_i, w)$. Hence, with this optimization the server's share xi_S in the TRAVERSE algorithm is empty, while $xi_C = x_{\pi(i)} + r_{\pi(i)}$. To preserve privacy, this pre-processing step must be recomputed for each tree evaluation. We have not yet implemented and evaluated this optimization and intend to do it in future work.

7.3 Handling Sparse Trees

We assumed in our protocol that the tree is complete. However, this is inefficient for sparse trees. We therefore optimize our protocol to handle sparse trees efficiently. This optimization requires only small changes to Fig. 3 and 4.

Let M_j denotes the number of nodes at level j. Then, for a sparse tree M_j is much more smaller than 2^j. However, a path may end at a level $j < d$. The idea is to stored each level j in an array of size at least M_j, instead of 2^j. When we are traversing a path that ends at level $j < d$, we secret-share the corresponding classification label to the parties in iteration $j - 1$. Then we simulate the remaining $d - j$ iterations and refresh the shares of the classification label each time.

We therefore use two additional variables u and e to assign the classification label and a bit respectively. During the tree evaluation, if we reached a leaf at level $j < d$, we assign the corresponding classification label to u and set e to 1. Then, we re-share both variables to the parties in each iteration.

For Fig. 3 parties receive additionally shares of variables u and e. Then, after the comparison (Step 3), we check if $e = 1$ (i.e., a leaf node was reached at level $j < d$) and choose v' randomly. This check is however only necessary for levels $j \geq 1$ (i.e., not at the root).

The modification for Fig. 4 is described in Fig. 10, where FSTCALL is used only for the first iteration (i.e., at the root node) and ITHCALL for all other iterations. The inputs are w (either a threshold or a classification label), i and u and e. For the first iteration, FSTCALL checks if the next node is a leaf (Step 3), set u and e appropriately (Step 4), and re-share i, u, e (Step 6). After the first iteration, ITHCALL checks if e is still 0 and if the next node is a leaf (Step 11 and 12). If this is true, then we set u, e to $w, 1$. Otherwise, we maintain their previous values (Step 13). Finally, we re-share i, u, e (Step 15).

Dataset	n	d	m	Time (s)				Bandwidth (KB)		
				[66]	F	M	S	Upload	Download	Total
ECG	6	4	6	0.344	**0.154**	**0.236**	**0.497**	- **(116.4)**	- **(164.7)**	101.9 **(281.0)**
Nursery	8	4	4	0.269	**0.127**	**0.273**	**0.479**	- **(116.4)**	- **(162.6)**	101.7 **(279.0)**
Breast-cancer	9	8	12	0.545	**0.256**	**0.376**	**0.927**	73.7 **(233)**	132.0 **(325.5)**	205.7 **(558.5)**
Heart-disease	13	3	5	0.370	**0.118**	**0.137**	**0.471**	73.3 **(87.4)**	43.9 **(124.5)**	117.2 **(211.8)**
Housing	13	13	92	4.081	**0.445**	**0.548**	**1.480**	115.7 **(378.7)**	1795.2 **(531.8)**	1910.9 **(910.5)**
Credit-screening	15	4	5	0.551	**0.164**	**0.306**	**0.474**	49.9 **(116.5)**	45.0 **(164.6)**	94.9 **(281.1)**
Spambase	57	17	58	16.595	**0.562**	**0.767**	**1.969**	463.4 **(490.5)**	17363.3 **(684.5)**	17826.7 **(1174.9)**

Table 3. Performance on UCI datasets: The numbers on the left are taken from [66]. The numbers on the right and bold are our costs using OT/OT. The columns F (LAN 10Gbps for Server and Client), M (LAN 1Gbps for Server and Wifi 72Mbps for Client), S (Wifi 144Mbps for Server and Wifi 72Mbps for Client) represent the type of network which has no impact on the bandwidth.

Dataset	Bandwidth (KB) IndexVector			Bandwidth (KB) IndexTree			Bandwidth (KB) GC		
	↑	↓	Total	↑	↓	Total	↑	↓	Total
ECG	35.80	22.83	61.63	14.76	30.94	45.7	65.68	110.76	176.44
Nursery	36.00	22.18	58.18	13.84	31.68	45.52	66.47	108.63	175.10
Breast-cancer	71.76	45.05	116.81	33.17	64.19	95.36	127.90	216.15	344.05
Heart-disease	26.88	16.57	43.45	11.74	23.48	35.22	48.65	84.31	132.96
Housing	116.57	72.53	189.10	54.51	108.80	163.31	207.50	350.33	557.83
Credit-screening	36.14	26.74	62.88	11.49	24.65	36.14	68.67	113.13	181.80
Spambase	152.53	92.32	244.83	71.04	144.56	215.60	266.78	447.48	714.26

Table 4. Detailed Bandwidth Costs on UCI datasets using OT/OT. The symbols ↑ and ↓ stand for upload and download.

8 Experiments

We have implemented our scheme with the *level indexing* optimization and performed some experiments which will be discussed in this section.

8.1 Experimental Setup

We evaluated our scheme with the *ObliVM* [49] which is a Java framework for secure computation. It offers a compiler for a domain-specific language *ObliVM-lang* and a GC backend *ObliVM-GC*, which primarily supports semi-honest GC protocol. We implemented our scheme in Java (1.8) using only the GC backend.

ObliVM-GC supports a standard garbling scheme with garbled row reduction, *FreeXOR* and *HalfGate*. It also implements the OT extension protocol proposed by [36] and a basic OT protocol by [53] based on the decisional Diffie-Hellman assumption in $\mathbb{Z}_p$. In our experiments, we use a 2048-bit key length for $\mathbb{Z}_p$ and SHA-256 as random oracle for the OT extension. For our OT_n^1 implementation, we instantiated the PRF with AES-128. Finally, *ObliVM-GC* provides a large set of built-in Boolean circuits and a GC implementation of Circuit ORAM [62]. The ORAM is secret-shared between the two parties and for each read and write operation the client and the server execute a GC protocol to scan the corresponding path of the ORAM, then they execute another GC protocol to perform the eviction operation.

We stress that we did not use the ObliVM compiler. Using the compiler to transform the plaintext tree evaluation program in a secure one results in a program whose size is proportional to the tree size. A similar idea was already considered in the related work [4, 12] and was outperformed by Wu et. al.'s paper [66]. Our memory accesses are not only to variable locations but they also depend on conditions involving secret variables. The ObliVM Compiler or any framework implementing the whole program will generate for each condition a corresponding oblivious computation whose number is proportional to the number of decision nodes. We then use only ObliVM-GC to run our manually created GC which are independent of the branching result.

As mentioned above, we compare our protocol to the scheme of Wu et. al. [66] in the semi-honest model at the same security level 128, because they perform an extensive comparison to the other protocols and have the best performance in the computational two-party setting. Hence, we choose a similar test environment. We run both parties on two machines with Intel(R) Xeon(R) CPU E7-4880 v2 at 2.50GHz connected via a

shared LAN and running Windows 10. The server machine has 4 CPUs and 4GB of RAM. The client machine has 4 CPUs and 8GB RAM. However, Wu et. al. implemented their protocol in C++. As HE they implemented exponential ElGamal using the 256-bit elliptic curve *numsp256d1*, which is one of the main source of their performance improvement comparing to previous protocols. Our implementation uses Oracle's Java 1.8 and all experiments were run on the Java SE 64-Bit Server virtual machine. We also compare our scheme to the scheme of Tai et. al. [60]. They implemented their protocol using ElGamal over elliptic curve *secp256k1*. However, we notice that, in their experiments [60], both the client and the server are run on one desktop computer equipped with Intel Core i7-6700 CPU (3.40 GHz).

Our scheme uses the OT extension protocol of [36] and requires therefore a setup phase which consist of an initialization of the OT extension protocol by running the basic OT ([53] in our implementation). The setup phase is executed only once to exchange symmetric keys which will be used in the OT extension [2]. It takes about 1 second and consumes about 20.31 KB (from sender to receiver) and 3.96 KB (from receiver to sender) communication. We note that [66] uses the very efficient OT extension of [2], which is not implemented in ObliVM yet. Additionally, if we want to index the tree with ORAM, we populate it in this setup phase. However, this is executed only once for each client. In the experiments below, we evaluate and report the costs for the online tree evaluation. In the following figures, we use *Server* (*Client*) *Time* to denotes the running time of the server (client) and *Server* (*Client*) *upload* to denote the number of bits sent by the server (client).

8.2 Performance on UCI Datasets

As Wu et. al., we evaluate our protocol on seven real datasets from the UCI repository (http://archive.ics.uci.edu/ml/) in the semi-honest model and at the security level 128. We first transform each tree in an array as explained above. Then, we perform 100 tree evaluations, measure runtime and bandwidth costs and compute the mean.

Our results are summarized in Table 3, where n, d, m, l are as defined in Table 1 and $l = 64$ as in [66]. Moreover, Tables 4 and 5 show how each sub-protocols contribute to the costs of Table 3. Recall that we can instantiate our scheme with four different array indexing methods. The results in Tables 3, 4 and 5 were achieved using OT indexing on both side.

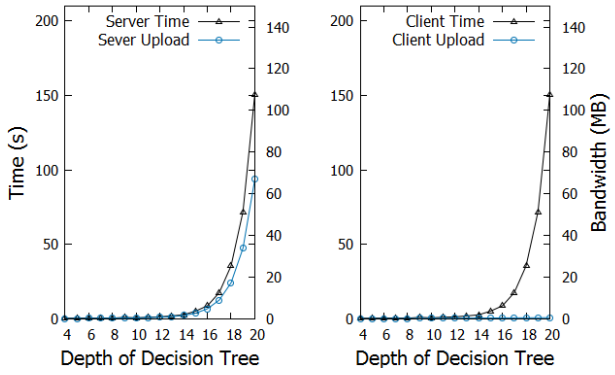
For small size trees, [66] has better bandwidth costs compared to the current implementation of our scheme. However, for applications that are willing to compromise on bandwidth, our scheme is more suitable as it is faster. Moreover, it has other advantages. First, it can be further optimized, e.g., by using the efficient OT extension [2, 3, 41] and fast garbling (e.g., JustGarble [6]), which are not yet implemented in ObliVM. In [49], it is estimated that combining ObliVM with JustGarble may reduce the time to compare 16384 bit Integers from 26 ms (using the original ObliVM as in our experiments) to 1.96 ms. The scheme of [2, 3] significantly reduces the runtime and bandwidth costs of the OT extension protocol to 41% and 50% respectively. OAI with OT can be implemented with the OT_n^1 of [41] which improves upon [54] by a factor ≈ 5.39. It is therefore clear that these optimizations will significantly improve the performance of our scheme in both runtime and bandwidth. Additionally, while [66] is based on public-key primitives (i.e., discrete logarithm on elliptic curve), our scheme relies only on symmetric cryptography. We require public-key primitives only for a one time initialization of the OT extension protocol. Since our scheme is based on secret array indexing, it naturally benefits from optimizations on oblivious data structures [40, 64]. Finally, the optimization described in Section 7.2 reduces the bandwidth costs of IndexVector to few kilobytes.

For mid-size to large trees (e.g., "housing" and "Spambase" datasets) our protocol outperforms previous protocols. Our protocol reduces the runtime of the "housing" dataset from 4 seconds (by Wu et. al.) to 0.5 second, while the communication cost is reduced from 2 MB to 1 MB. For the "Spambase" dataset our protocol is even 17 times better. Their takes 17 seconds consuming 18 MB bandwidth, while we run within less than 1 second and require only 1.2 MB communication.

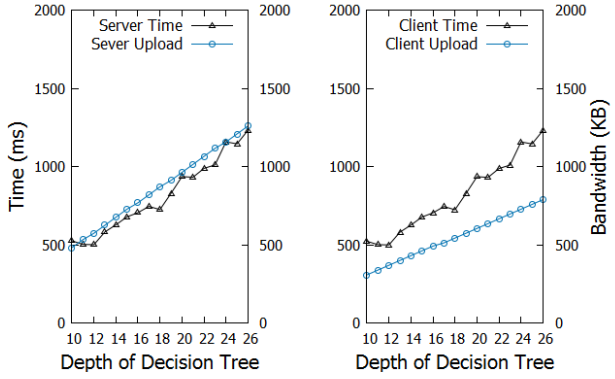
Tai et. al. [60] also compare their work to [66], however, running both the client and the server on one desktop computer equipped with Intel Core i7-6700 CPU (3.40 GHz). Their reported time consists therefore only of the local computation time without network traffic. For small trees, they reported similar performances to [66] in both bandwidth and runtime. For large trees such as "housing" and "Spambase", their protocols run in about 2s (1.984 and 1.804 resp.) on one machine consuming slightly less than 1MB (0.854 and 0.920 resp.). As a result, our scheme is already faster and can be further optimized as explained above. This shows that flattening the tree increases performance.

Dataset	Time (ms)								
	IndexVector			IndexTree			GC		
	F	M	S	F	M	S	F	M	S
ECG	16.52	56.95	139.15	11.92	34.36	59.11	108.76	136.35	314.99
Nursery	15.31	67.12	126.31	12.05	39.12	46.70	85.40	160.51	323.17
Breast-cancer	33.51	104.31	211.95	21.25	54.10	108.11	179.10	210.65	626.59
Heart-disease	9.95	37.25	128.21	7.84	18.13	69.12	73.52	75.83	291.33
Housing	50.92	156.0	265.88	37.56	76.86	165.04	334.80	310.52	1067.25
Credit-screening	17.53	65.87	127.53	10.14	46.24	44.90	127.88	187.80	321.87
Spambase	71.83	213.33	501.18	44.64	104.61	220.44	427.85	443.98	1266.41

Table 5. Detailed Time Costs on UCI datasets using OT/OT. The columns F, M, S have the same meaning as in Table 3.



(a) Complete Trees



(b) Sparse Trees

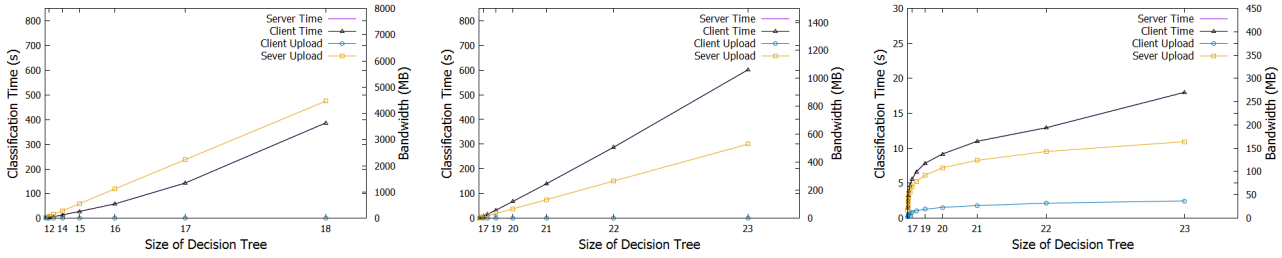
Fig. 11. Scalability experiment with OT/OT indexing

8.3 Scalability

We also evaluate the scalability of our scheme by experimenting with synthetic trees of different depths and densities and using similar parameters as previous work. We vary the depth of the tree between 4 and 26 and use 64-bit precision and $n = 16$. We also ran both experiments of this section on a LAN, while Wu et. al. reported costs excluding network traffic.

First, we consider complete decision trees, which are the worst-case. We evaluate the scheme with OT indexing for the tree and the attribute vector and measure computation and communication costs of both parties. The result is depicted in Figure 11a. Our results show that even for deeper trees with a depth around 20, the scheme takes less than 3 minutes and less than 70 MB bandwidth. For complete trees of depth 14, the maximal depth reported by [66], their protocol runs in about 5 minutes, excluding network communication, and requires about 120 MB bandwidth. Our protocol evaluates a complete tree of depth 14 in less than 3 seconds via LAN, while consuming only about 2 MB of the bandwidth, which is 60 times better.

Using a complete tree even in case when the decision tree has only a few nodes is totally inefficient, because for deeper trees the difference between $2^{d+1} - 1$ and the real size of the tree can be very huge. For their experiments on sparse trees, Wu et. al. assume that the number of decision nodes is linear in the depth of the tree, e.g., $m = 25d$. We experiment with sparse trees using the optimization mentioned in Section 7.3. We vary the depth of the tree between 10 and 26 and set the number of decision nodes to $m = 25d$. Then we generate random trees with defined parameters d and $m = 25d$, run our protocol and measure the results. Figure 11b shows that our costs grow linearly, rather than exponentially in the depth of the tree as in [66]. For sparse trees of depth up to 26, our protocol takes less than 1.5 seconds and less than 1.5 MB bandwidth. For sparse trees of depth 20, the maximal depth reported by [66], their protocol runs in about 2 minutes (excluding network communication) and requires about 140 MB bandwidth. The scheme of [60] runs in about 10 seconds (excluding network communication) and requires about 4.1 MB bandwidth. Our protocol evaluates a sparse tree of depth 20 with 500 decision nodes in less than 1 second via LAN, while consuming only about 1.5 MB bandwidth.



(a) Indexing with GC/OT **(b)** Indexing with OT/OT **(c)** Indexing with ORAM/OT

Fig. 12. Costs for very large trees: For readability only the depth (i.e., $\log(M) - 1$) is displayed on the x-axis, which ends at 18 for GC and 23 for OT and ORAM because we ran out of memory.

8.4 Very Large Trees

In our last experiment, we consider very large complete trees with depth larger than 20, containing millions of decision nodes. In this setting, we expect the ORAM solution to outperform the other approaches, since it yields to a sub-linear complexity. We ran this experiment on a single machine (via a loopback interface) with Intel(R) Core(TM) i7-4770 CPU at 3.40GHz, 16GB of RAM and Windows 10. We then run three experiments using either GC, OT or ORAM to index the decision tree and OT indexing for the attribute vector. For each experiment, we set $n = 64$ and vary the depth from 10 to 24. As mentioned above, indexing with ORAM requires populating the ORAM in the setup phase, which is very tedious for large trees and may take up to 20 days to compute [31]. However, notice that the costs for an ORAM access depend on the capacity of the ORAM, but not on the actual number of elements stored in it. For this reason, we avoid a long ORAM initialization by populating the ORAM for larger trees with just enough elements to evaluate the decision tree. The results of the experiment are summarized in Figure 12. For small trees with depth smaller than 12, GC and OT indexing are better than ORAM. The computation cost for OT remains better than ORAM up to depth 16. However, the costs for GC and OT double with the depth and are linear in the size of the tree, while the costs for ORAM are sublinear in the size of the tree as shown in Figure 12. For trees of depth larger than 21, ORAM outperforms both GC and OT in computation and communication costs. For example, for depth 22, ORAM takes about 13 seconds and 175 MB of Bandwidth, while OT takes 287 seconds and 266 MB.

9 Conclusion

In this paper, we presented a protocol for evaluating private decision trees using sublinear communication. The idea of our novel solution is to represent the tree as an array. Then we execute a number of comparisons that is equal to the depth of the tree. We get the inputs to the comparison by obliviously indexing the tree and the attribute vector. Each comparison outputs secret shares of the index of the next node to be evaluated. We implement oblivious array indexing using either GC, OT or ORAM. Using ORAM this results in the first protocol with sub-linear communication cost in the size of the tree. We implemented and evaluated our scheme on real datasets and synthetic decision trees. Our results show that, we are not only able to provide the first sublinear communication cost for large trees, but also reduce the computation and communication costs for mid-size to large real-world data set compared to the best related work. In future work, we intend to evaluate our scheme with other frameworks such as JustGarble [6], SCAPI [25], SPDZ [21], MASCOT [39] and efficient OT extension protocols [2, 3, 38, 41, 56].

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 644579 of ESCUDO-CLOUD project.

References

[1] A. Aly and M. V. Vyve. Securely solving classical network flow problems. In *ICISC*, pages 205–221, 2014.

[2] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, pages 535–548, New York, NY, USA, 2013. ACM.

[3] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer extensions. *J. Cryptology*, 30(3):805–858, 2017.

[4] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. Secure evaluation of private linear branching programs with medical applications. In *ESORICS*, pages 424–439, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] D. Beaver. Commodity-based cryptography (extended abstract). In *STOC*, pages 446–455, New York, NY, USA, 1997. ACM.

[6] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *SP*, pages 478–492, Washington, DC, USA, 2013. IEEE Computer Society.

[7] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: A system for secure multi-party computation. In *CCS*, pages 257–266, New York, NY, USA, 2008. ACM.

[8] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, 1988.

[9] M. Blanton, A. Steele, and M. Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS*, pages 207–218, 2013.

[10] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[11] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[12] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In *CCS*, pages 498–507, New York, NY, USA, 2007. ACM.

[13] S. S. Burra, E. Larraia, J. B. Nielsen, P. S. Nordholt, C. Orlandi, E. Orsini, P. Scholl, and N. P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. *IACR Cryptology ePrint Archive*, 2015:472, 2015.

[14] J. Catlett. Overpruning large decision trees. In *IJCAI*, pages 764–769, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[15] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *STOC*, pages 11–19, 1988.

[16] M. D. Cock, R. Dowsley, C. Horst, R. Katti, A. C. A. Nascimento, S. C. Newman, and W. Poon. Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. *IACR Cryptology ePrint Archive*, 2016:736, 2016.

[17] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT*, pages 280–299, 2001.

[18] I. Damgård, M. Geisler, and M. Krøigaard. Efficient and secure comparison for on-line auctions. In *ACISP*, pages 416–430, 2007.

[19] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *PKC*, pages 160–179, 2009.

[20] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, pages 1–18, 2013.

[21] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[22] I. Damgård and R. Thorbek. Efficient conversion of secret-shared values between different fields. *IACR Cryptology ePrint Archive*, 2008:221, 2008.

[23] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[24] J. Doerner and A. Shelat. Scaling oram for secure computation. In *CCS*, pages 523–535, 2017.

[25] Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell. SCAPI: the secure computation application programming interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012.

[26] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith. CBMC-GC: an ANSI C compiler for secure two-party computations. In *CC '14*, pages 244–249, 2014.

[27] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS*, pages 1322–1333, 2015.

[28] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, New York, NY, USA, 1987. ACM.

[29] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[30] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[31] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, pages 513–524, 2012.

[32] T. Graepel, K. Lauter, and M. Naehrig. Ml confidential: Machine learning on encrypted data. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, ICISC'12, pages 1–21, Berlin, Heidelberg, 2013. Springer-Verlag.

[33] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[34] W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*, pages 451–462, 2010.

[35] E. Hesamifard, H. Takabi, M. Ghasemi, and C. Jones. Privacy-preserving machine learning in cloud. In *Proceedings of the 2017 on Cloud Computing Security Workshop*, CCSW '17, pages 39–43, New York, NY, USA, 2017. ACM.

[36] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, pages 145–161, 2003.

[37] A. Jarrous and B. Pinkas. Secure hamming distance based computation and its applications. In *ACNS*, pages 107–124, 2009.

[38] M. Keller, E. Orsini, and P. Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO*, pages 724–741, 2015.

[39] M. Keller, E. Orsini, and P. Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, pages 830–842, 2016.

[40] M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In *ASIACRYPT*, pages 506–525, 2014.

[41] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO*, pages 54–70. Springer, 2013.

[42] V. Kolesnikov, A. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, pages 1–20, 2009.

[43] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, pages 486–498, 2008.

[44] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *FC*, pages 83–97, 2008.

[45] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *CRYPTO*, volume 1880, pages 36–54, Berlin and New York, 2000. Springer.

[46] Y. Lindell and B. Pinkas. Privacy preserving data mining. *Journal of Cryptology*, 15(3):177–206, 2002.

[47] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *IACR Cryptology ePrint Archive*, 2008:197, 2008.

[48] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptol.*, 22(2):161–188, Apr. 2009.

[49] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *SP*, pages 359–376, 2015.

[50] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *SSYM*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[51] P. Mohassel, S. S. Sadeghian, and N. P. Smart. Actively secure private function evaluation. In *ASIACRYPT*, pages 486–505, 2014.

[52] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38, 2017.

[53] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA*, pages 448–457, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[54] M. Naor and B. Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18:1–35, Jan 2005.

[55] J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *TCC*, pages 368–386, 2009.

[56] A. Patra, P. Sarkar, and A. Suresh. Fast actively secure OT extension for short secrets. In *NDSS*. The Internet Society, 2017.

[57] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. *IACR Cryptology ePrint Archive*, 2009:314, 2009.

[58] P. Pullonen, D. Bogdanov, and T. Schneider. The design and implementation of a two-party protocol suite for sharemind 3, Sept. 2012.

[59] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[60] R. K. H. Tai, J. P. K. Ma, Y. Zhao, and S. S. M. Chow. Privacy-preserving decision trees evaluation via linear functions. In *ESORICS*, pages 494–512, 2017.

[61] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *USENIX*, pages 601–618, 2016.

[62] X. Wang, T. H. Chan, and E. Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *CCS*, pages 850–861, 2015.

[63] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *CCS*, pages 191–202, 2014.

[64] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *CCS*, pages 215–226, New York, NY, USA, 2014. ACM.

[65] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

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

[67] X. Wu, M. Fredrikson, S. Jha, and J. F. Naughton. A methodology for formalizing model-inversion attacks. In *CSF*, pages 355–370, 2016.

[68] A. C. Yao. Protocols for secure computations. In *SFCS*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[69] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, pages 220–250, 2015.

A Complexity Analysis

This section presents the complexity analysis of our scheme and compares it to previous work [60, 66]. We focus on the level indexing and sparse trees optimizations of Sections 7.1 and 7.3.

A.1 Asymptotic Analysis

Asymptotic Analysis of [60, 66]. We begin by recalling the cost provided by [60, 66]. In [66] the client performs $O((n+m)l)$ asymmetric and $O(d)$ symmetric operations, while the server performs $O(ml)$ asymmetric and $O(2^d)$ symmetric operations. In [60] the client performs $O((n+m)l)$ asymmetric operations, and the

server $O(ml)$ asymmetric operations.

Asymptotic Analysis of our Scheme. Our cost consists of the cost for GCs Traverse and Move (C_{GC}), the cost for IndexVector (C_{IV}) and the cost for IndexTree (C_{IT}). If $C_{IT}^{(j)}$ denotes the cost for IndexTree at level j (Section 7.1), then the total cost is $d \cdot C_{GC} + d \cdot C_{IV} + \sum_{j=1}^{d} C_{IT}^{(j)}$. All computations require only symmetric operations. Each GC consists of $6l$ ciphertexts resulting in a total of $O(6ld) = O(ld)$ operations for each party.

OAI With OT. In this case, the cost of IndexVector is $O(dn)$ for the client, and $O(d\log(n))$ for the server. Each level j of the tree has 2^j elements, resulting in $O(d^2)$ and $O(2^d)$ operations in IndexTree for the client and the server respectively. Overall, the client and server perform $O(ld + dn + d^2) = O(d^2)$ and $O(ld + d\log(n) + 2^d) = O(2^d)$ symmetric operations.

OAI With ORAM. The asymptotic cost of recursive circuit ORAM for an array of size $O(2^d)$ is $O(d^3)$. Hence with ORAM the costs of the client and server are respectively $O(ld + dn + d^4)$ and $O(ld + dlog(n) + d^4)$. For ODS and FLORAM, it suffices to replace $O(d^3)$ by $O(d^2)$ and $O(d)$ respectively. Assuming constant l and n (as in the experiment with complete trees, Section 8.3), the complexity is indeed sublinear in the size $O(2^d)$ of the tree.

Asymptotic Analysis Summary. The above cryptographic operations are in fact encryption/decryption operations such that we can assimilate their number to the number of ciphertexts sent. In the worst case (complete tree), $m \approx 2^d$ is the dominant factor. This results in an overall asymptotic cost of $O(m)$ as claimed in Table 2 for [60, 66] and our scheme.

A.2 Concrete Analysis

We now compute the estimated concrete communication cost depending on the protocols parameters.

Concrete Analysis of [60, 66]. We first provide our own analysis of the communication of previous work. Let λ denote the bit length of the asymmetric ciphertext. For ElGamal using 256-bit elliptic curve, $\lambda = 1024$ (a ciphertext consists of two group elements, which on the elliptic curve are encoded as points with coordinates 256-bit long). We use κ to denote the symmetric security parameter, hence $\kappa = 128$. Wu et. al.'s scheme uses the OT extension protocol of [2] in which for sending μ strings of length κ in parallel, the receiver sends $\mu\kappa$ and the sender $2\mu\kappa$ bits.

In [66] each party sends two messages before engaging in an 1-out-of-2^d-OT (with the server as sender) which is implemented using Naor and Pinkas OT [54] by running d times the OT extensions of [2] and transferring 2^d symmetric ciphertexts of length 2κ each. This results in the client sending $\lambda ln + \lambda m + 2d\kappa$ and the server sending $\lambda lm + \lambda 2^d + 4d\kappa + 2^d 2\kappa$ bits.

The analysis of [60] is much simpler. Also in this case, each party sends two messages, which results in the client sending $\lambda ln + \lambda m$ and the server sending $\lambda lm + 2\lambda(m+1)$ bits.

Concrete Analysis of our Scheme. We implemented our scheme with the OT extensions of [36] which has concrete cost of $2\kappa(\kappa + \mu)$ bits for both sender and receiver to transfer μ strings of length κ. In each GC the cost of the client (as evaluator) consists of its cost in OT, while the cost of the server (as generator) consists of the ciphertexts of the garbled tables (GT), the garbled input (GI) of the server and the OT cost. For each node DS (w, vr, vl, i), the threshold w has the same bit length l as the attribute values, while the length of vl, vr, i might be much more smaller and will be denoted by l'. The length of i depends on the size of the attribute vector, while vl, vr depend on the number of nodes at the corresponding level in the tree (Section 7.1). In our evaluation for UCI Datasets, $l' = 20$ for spambase and $l' = 16$ for the other datasets.

The input length of the client in GC Traverse is $2(l + l')$ hence the client sends $2\kappa(\kappa + 2l + 2l')$ bits. Similarly, in GC Move the client sends $2\kappa(\kappa + l + 2l')$ bits resulting in a total of $2\kappa d(2\kappa + 3l + 4l')$. In GC Traverse, the server sends $2\kappa(l + 2l')$ bits as GT cost, $\kappa(2l + 3l')$ bits as GI cost and $2\kappa(\kappa + 2l + 2l')$ bits in OT. In GC Move, the server sends $2\kappa(l + 2l')$ bits as GT cost, $\kappa(l + l')$ bits as GI cost and $2\kappa(\kappa + l + 2l')$ bits in OT. Overall the server sends $\kappa d(4\kappa + 13l + 20\kappa l')$ bits as GC cost.

Recall that the array indexing with OT is implemented with Naor and Pinkas OT_n^1 that requires running $\log(n)$ times OT_2^1 and transferring n symmetric ciphertexts. In IndexVector, the client as sender sends $\kappa d(2\kappa + 2\log(n) + n)$ and the server sends $2\kappa d(\kappa + \log(n))$ bits.

In IndexTree, the parties run $\text{OT}_{2^j}^1$ at each level j in $[1, d]$, where the client and the server send $2\kappa(\kappa + j)$ and $2\kappa(\kappa + j) + \kappa 2^i$ bits respectively. As a result, the client sends $2\kappa(\kappa d + \frac{(1+d)d}{2})$ and the server $2\kappa(\kappa d +$

$\frac{(1+d)d}{2}) + 2\kappa(2m+1)$ bits (a tree with m decision nodes, has $2m+1$ nodes in total).

In summary, the client sends

$$2\kappa d(2\kappa + 3l + 4l') + \kappa d(4\kappa + 2\log n + n + d + 1)$$

bits and the server sends

$$\kappa d(4\kappa + 13l + 20l') + \kappa d(4\kappa + 2\log(n) + d + 1) + 2\kappa(2m+1)$$

bits in our protocol.

Concrete Analysis Summary. To summarize this section (see Table 6), the protocols in [60, 66] perform m comparisons using asymmetric operations, while we perform only $d = \log(m)$ comparisons using symmetric operations. Using GC or OT, we still have linear cost as previous work, while requiring only symmetric operations with small concrete constants. Using ORAM (or ODS, FLORAM), C_{IT} is computed accordingly and OAI is sublinear while requiring only symmetric operations. Our implementation uses a concurrent queue implementation offered by ObliVM, which uses only 2 threads to manage network input/output. While running the protocol, the main thread inserts new messages in the queue (it waits until there is enough place). The second thread is only responsible for sending the content of the queue over the network. Previous work can also use multithreading to improve the execution time, but not the communication. Their major advantage is the use of ECC which allows smaller ciphertexts than when using a group $\mathbb{Z}_p$ with prime p. Our scheme, however, can still be optimized by using more efficient garbling [6], OT extensions [2, 3, 38] and ORAM (ODS [40, 64], FLORAM [24]). Using more efficient OT extensions [2, 3] that reduces the runtime and bandwidth to 41% and 50% respectively, will significantly improve the performance of our scheme.

A.3 Round Complexity

The main cryptographic primitives used in our scheme are OT extension, OT^1_N and GC, which are all one round protocol. The scheme itself consists of d iterations. Each iteration consists of an OAI on x, a GC Traverse, an OAI on the decision tree and a GC Move. When instantiated with OT or GC, OAI has one round. Hence, our scheme has $4d$ rounds. Recursive Circuit ORAM with size N has $log(N)$ rounds, resulting in $(d+3)d = d^2+3d$ rounds for our scheme, when instantiated with ORAM. ODS and FLORAM have one round, resulting in $4d$ rounds for our scheme.

B Correctness and Security

This section discusses the correctness and security of our scheme. Our security proofs follow the idea of [48]. We construct simulators as in [33, 48] and refer to the same references for the indistinguishability part.

B.1 Sub-protocols

Lemma B.1. *The GC protocol in Fig. 3 is correct and secure in the semi-honest model.*

Proof. Depending on the comparison result between the attribute value xi and the threshold, Fig. 3 returns the correct index of the next node. Security follows from Yao's garbled circuit protocol. □

Lemma B.2. *The GC protocol in Fig. 4 is correct and secure in the semi-honest model.*

Proof. If the execution of the protocol reaches a leaf, then i must be Null which is correctly checked in Step 2. As a result, the correct classification label is computed in Step 3. Security follows from Yao's garbled circuit protocol. □

Lemma B.3. *The OAI using the GC in Fig. 5 is correct and secure in the semi-honest model.*

Proof. The equality check in step 4 returns 1 for exactly one index between $0, \ldots, n-1$. The selection step 5 returns 0 for all $j < i$ and A_i for all $j \geq i$. The output A_i is secret-shared in step 7. Security follows from Yao's garbled circuit protocol. □

Lemma B.4. *The OAI as described in Fig. 7 is correct and secure in the semi-honest model.*

Proof. Let $i_S = r$ and $i_C = i+r$. From the correctness of the OT^1_n protocol, C receives the keys corresponding to $i+r$ and can decrypt $A'_{i+r \bmod n} = A_i \oplus s$. Security also follows from the OT^1_n protocol, which guarantees that C can decrypt only one message. Finally, this message is blinded by a random value s.

Let $view^{OT}_S, view^{OT}_C$ be the view of the sender and the receiver in the OT^1_n protocol. Then the respective views $view^{OTI}_S, view^{OTI}_C$ in the OT indexing are $(A', view^{OT}_S, A_{iS})$ and $view^{OT}_C$. Moreover, let SIM^{OT}_S, SIM^{OT}_C be the simulator of the OT^1_n for sender

Dataset	[66]			[60]			This Work			This Work
	↑	↓	**Total**	↑	↓	**Total**	↑	↓	**Total**	**With OT [2]**
ECG	48.88	50.75	**99.63**	48.75	49.75	**98.5**	97.06	137.09	**234.16**	**105.78**
Nursery	64.62	34.75	**99.37**	64.5	33.25	**97.75**	97.31	137.09	**234.40**	**105.90**
Breast-cancer	73.75	136.5	**210.25**	73.5	99.25	**172.75**	195.25	274.90	**470.15**	**213.15**
Heart-disease	104.71	41.43	**146.15**	104.62	41.5	**146.12**	73.17	102.90	**176.07**	**79.70**
Housing	115.90	2016.81	**2132.71**	115.5	759.25	**874.75**	319.10	452.25	**771.35**	**353.73**
Credit-screening	120.75	42.75	**163.5**	120.62	41.5	**162.12**	97.75	137.15	**234.90**	**106.40**
Spambase	463.78	20945.06	**21408.84**	463.25	478.75	**942.0**	439.60	610.87	**1050.48**	**494.79**

Table 6. Comparison of estimated bandwidth costs (in KB) on UCI datasets. The symbols ↑ and ↓ stand for upload to server and download to client. Columns **Total** are the total costs. The last column is our estimated total cost when using OT extension of [2].

and receiver. We construct the simulators of the OT indexing as follows:

- Sender: the simulator $SIM_S^{OTI}((A, i_S), A_{iS})$ receives as input the array A, a share i_S of the index and a share A_{iS} of the indexed element. The simulator computes $A'_{j+r \bmod n} \leftarrow A_j \oplus A_{iS}, j = 0, \cdots, n-1$ and outputs $(A', SIM_S^{OT}(A', \emptyset), A_{iS})$.
- Receiver: the simulator $SIM_C^{OTI}(i_C, A_{iC})$ receives as input a share i_C of the index and a share A_{iC} of the indexed element. It just outputs $SIM_C^{OT}(i_C, A_{iC})$

The output of both simulators SIM_S^{OTI} and SIM_C^{OTI} are clearly indistinguishable from the corresponding party's view in the real protocol. □

B.2 Main Protocol

Theorem B.5 (Correctness)**.** *The protocol described in Fig. 2 is correct.*

Proof. We prove by induction on the level of the tree that the protocol correctly traverses the tree. At level 0 there is only the root node. At level $d' < d$ the correctness of the sub-protocols guarantees the computation of the correct attribute value xi at Step 5, the correct node index v' at Step 7 and the correct node $\mathcal{N}'$ at Step 8. Finally, at level d Fig. 4 returns the correct classification label. □

Theorem B.6 (Security)**.** *The protocol described in Fig. 2 is secure in the semi-honest model.*

Proof. Given a DT model $\mathcal{M}$, the simulator SIM_S generates random elements to simulate the sharing of the root (Step 1). Then it generates a random attribute vector $\tilde{x}$ and invokes d times the simulators of the sub-protocols for the server. Analogously, given d and x the simulator SIM_C generates a random model $\mathcal{M}'$, simulates the sharing of the root as above and invokes d times the simulators of the sub-protocols for the client.

Let $view_S^{IV}, view_S^{TR}, view_S^{IT}, view_S^{MV}$ denote the real views of the server in the sub-protocols IndexVector, Traverse, IndexTree, Move respectively. Moreover, let $SIM_S^{IV}(\widetilde{i_S}, \widetilde{y_S})$, $SIM_S^{TR}(\widetilde{y_S}, \widetilde{v_S})$, $SIM_S^{IT}((N, \widetilde{v_S}), \widetilde{\mathcal{N}_S})$, $SIM_S^{MV}((\widetilde{w_S}, \widetilde{i_S}), (0, \widetilde{i_S}))$ be the respective simulators. Then the view $view_S^{PDTE}$ of the server in Fig. 2 is:

$$\mathcal{N}_S, \underbrace{view_S^{IV}, view_S^{TR}, view_S^{IT}, view_S^{CL}, \cdots}_{d \text{ times}}$$

The simulator for the server $SIM_S^{PDTE}(\mathcal{M}, \emptyset)$ receives as input the decision tree, generates a random $\widetilde{\mathcal{N}_S}$, invokes d times the simulators of the sub-protocols and outputs:

$$\widetilde{\mathcal{N}_S}, \underbrace{SIM_S^{IV}(\cdot), SIM_S^{TR}(\cdot), SIM_S^{IT}(\cdot), SIM_S^{MV}(\cdot), \cdots}_{d \text{ times}}$$

The simulation for the client is similar. □