Mostree: Malicious Secure Private Decision Tree Evaluation with Sublinear Communication

Jianli Bai
University of Auckland
Auckland, New Zealand
jbai795@aucklanduni.ac.nz

Xiangfu Song*
National University of Singapore
Singapore
songxf@comp.nus.edu.sg

Xiaowu Zhang
CloudWalk Technology
Beijing, China
zhangxiaowu@cloudwalk.com

Qifan Wang
University of Auckland
Auckland, New Zealand
qwan301@aucklanduni.ac.nz

Shujie Cui
Monash University
Melbourne, Australia
shujie.cui@monash.edu

Ee-Chien Chang
National University of Singapore
Singapore
changec@comp.nus.edu.sg

Giovanni Russello
University of Auckland
Auckland, New Zealand
g.russello@auckland.ac.nz

ABSTRACT

A private decision tree evaluation (PDTE) protocol allows a feature vector owner (FO) to classify its data using a tree model from a model owner (MO) and only reveals an inference result to the FO. This paper proposes Mostree, a PDTE protocol secure in the presence of malicious parties with sublinear communication. We design Mostree in the three-party honest-majority setting, where an (untrusted) computing party (CP) assists the FO and MO in the secure computation. We propose two low-communication oblivious selection (OS) protocols by exploiting nice properties of three-party replicated secret sharing (RSS) and distributed point function. Mostree combines OS protocols with a tree encoding method and three-party secure computation to achieve sublinear communication. We observe that most of the protocol components already maintain *privacy* even in the presence of a malicious adversary, and what remains to achieve is *correctness*. To ensure correctness, we propose a set of lightweight consistency checks and seamlessly integrate them into Mostree. As a result, Mostree achieves sublinear communication and malicious security simultaneously. We implement Mostree and compare it with the state-of-the-art. Experimental results demonstrate that Mostree is efficient and comparable to semi-honest PDTE schemes with sublinear communication. For instance, when evaluated on the MNIST dataset in a LAN setting, Mostree achieves an evaluation using approximately 768 ms with communication of around 168 KB.

CCS CONCEPTS

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

KEYWORDS

decision tree, privacy-preserving, sublinear, malicious security

1 INTRODUCTION

Decision trees have found extensive use in various real-world applications, including spam filtering [10], credit risk assessment [26], and disease diagnosis [33]. In many scenarios, the tree model owner and feature owner are distinct parties, and neither of them wants to disclose data to the other due to commercial or privacy concerns.

Private decision tree evaluation (PDTE) protocols allow a feature owner (FO) to learn a classification result evaluated using a decision tree (DT) from a model owner (MO) without revealing anything more. Recently, many PDTE protocols [6, 11, 14, 24, 25, 30, 35, 39] have been proposed with different security and efficiency trade-offs. Ideally, a PDTE protocol should achieve sublinear communication in the tree size. However, as shown in Table 1, most existing PDTE protocols [6, 11, 24, 25, 35, 39] require linear communication, which can be impractical for real-world applications since commercial DTs typically contain thousands or millions of nodes [27]. Moreover, almost all existing PDTE protocols are only secure against semi-honest adversaries, where the adversary honestly follows the protocol specification. Indeed, the adversary could maliciously behave. Achieving private and correct tree evaluation in the presence of a malicious adversary is vital since PDTE protocols are usually used for high-sensitive applications, *e.g.*, healthcare or financial services. Only the works proposed in [14, 30, 35, 39] achieve security against malicious adversaries. In particular, the works [30, 35, 39] only achieve security against a malicious FO, which we call one-side malicious security. Besides, all four works require linear communication costs, limiting their scalability.

In this paper, we propose Mostree, a PDTE protocol that simultaneously achieves security against malicious adversaries and sublinear communication. Mostree considers a three-party honest-majority setting, where a malicious adversary can compromise one party, and the compromised party could be anyone in the system. The main application for Mostree can be found in cloud-assisted privacy-preserving machine learning (PPML) service, as considered

*Corresponding author

Jianli Bai, Xiangfu Song, Xiaowu Zhang, Qifan Wang, Shujie Cui, Ee-Chien Chang, and Giovanni Russello

Table 1: Summary of Some Existing PDTE Protocols

Protocol	Comparison	Communication	Sublinear	Leakage	Security	Corruption	Parties
Bost *et al.* [6]	$\lceil m/2 \rceil$	$O(n+m)$	$\times$	m	$\bigcirc$	1-out-of-2	2PC
Kiss *et al.* [25](GGG)	d	$O(\overline{m}\ell)$	$\times$	$\overline{m}, d$	$\bigcirc$	1-out-of-2	2PC
Kiss *et al.* [25](HHH)	$\lceil m/2 \rceil$	$O((n+m)\ell)$	$\times$	m	$\bigcirc$	1-out-of-2	2PC
Brickell *et al.* [11]	d	$O((n+m)\ell)$	$\times$	m	$\bigcirc$	1-out-of-2	2PC
Joye *et al.* [24]	d	$O(d(\ell+n)+2^d)$	$\times$	d	$\bigcirc$	1-out-of-2	2PC
Tueno *et al.* [36](ORAM)	d	$O(d^4\ell)$	$\sqrt{}$	d	$\bigcirc$	1-out-of-2	2PC
Bai *et al.* [4]	d	$O(dn\ell)$	$\sqrt{}$	m, d	$\bigcirc$	1-out-of-2	2PC
Wu *et al.* [39]	2^d	$O(2^d+(n+m)\ell)$	$\times$	m, d	$\circ\!\!\!\bullet$	1-out-of-2	2PC
Tai *et al.* [35]	$\lceil m/2 \rceil$	$O((n+m)\ell)$	$\times$	m	$\circ\!\!\!\bullet$	1-out-of-2	2PC
Ma *et al.* [30]	d	$O(dn\ell)$	$\sqrt{}$	m, d	$\circ\!\!\!\bullet$	1-out-of-2	2PC
Ji *et al.* [23]	d	$O(d(\log n + \log m + \ell))$	$\sqrt{}$	m, d	$\bigcirc$	1-out-of-3	3PC
Damgård *et al.* [14]	m	$O(2^d n\ell)$	$\times$	$\overline{m}, d$	$\bullet$	2-out-of-3	3PC
Mostree	d	$O(dn\ell \log m)$	$\sqrt{}$	m, d	$\bullet$	1-out-of-3	3PC

Comparison denotes the number of secure comparisons needed; *Parameters*: m: number of tree nodes, $\overline{m}$: number of tree nodes in a depth-padded tree, see [25], n: dimension of a feature vector, d: the longest depth of a tree, ℓ: bit size of feature values. *Symbols*: $\times$: no, $\sqrt{}$: yes, $\sqrt{}$: partially support: linear offline communication, sublinear online communnication; $\bigcirc$: semi-honest, $\circ\!\!\!\bullet$: one-side malicious, $\bullet$: malicious.

by Sharemind (https://sharemind.cyber.ee), TFEncrypted (https://tf-encrypted.io) and SecretFlow (https://www.secretflow.org.cn/docs/secretflow/latest/en-US). Mostree protects both the tree model and the queried features from all parties using replicated secret sharing (RSS). During the tree evaluation, protecting which node is being accessed in each level, *i.e.* the tree access pattern, is also imperative since the classification result is highly relevant to the tree path. In particular, if a model holder knows which path is accessed for each query, it can learn the classification result directly, which should be forbidden to protect the feature owner's privacy. Mostree uses oblivious selection (OS) protocols to hide the access pattern from all parties, which allows the three parties to traverse the decision tree collaboratively and obliviously, without learning which node is being touched. We design two OS protocols. The first OS protocol is purely based on RSS sharings, achieving constant online communication and linear offline communication. We then propose the second OS protocol by applying distributed point function (DPF) [22] over RSS sharings, achieving constant online and sublinear offline communication.

The remaining challenge is how to achieve security against malicious adversaries while ensuring sublinear communication. We exploit the fact that most of our proposed semi-honest protocol components already maintain *privacy* in the presence of a malicious adversary; what we need to ensure is *correctness*.[1] To this end, we propose a set of lightweight consistency check techniques. Notably, all of our check mechanisms are efficiency-oriented by exploiting nice properties of underlying primitives, and they bootstrap existing RSS-based security mechanisms (*e.g.*, low-level RSS-based ideal functionalities) to ensure correctness and introduce low overhead. By integrating these checks, Mostree simultaneously achieves sublinear communication and security against malicious adversaries.

We implement Mostree over different datasets and report efficiency under different network settings. Our results demonstrate that Mostree is highly competitive to the PDTE protocol proposed in [23], which is the latest and most efficient existing solution designed in semi-honest settings. In MNIST testing within a LAN environment, Mostree requires only 4× online communication and 4× online computation compared to [23]. Compared with the malicious security work given in [14], Mostree reduces up to around 311× and about 4× in communication and computation, respectively. Furthermore, our experiments illustrate the scalability of Mostree, particularly for large trees with high dimensions, due to its sublinear communication property.

Contributions. We summarize our contributions as follows:

- We propose two oblivious selection protocols over a three-party setting. The first protocol is based purely on RSS, and the second is on DPF and RSS. Both of them are with low overhead, *e.g.*, sublinear communication. The proposed OS protocol may apply to applications in other areas, *e.g.*, secure database processing.
- We enhance OS protocols with malicious security by proposing lightweight consistency check techniques. We carefully combine the proposed OS protocol, efficient consistency check, and three-party secure computation to design Mostree. Mostree achieves sublinear communication and malicious security in the three-party honest-majority setting. To our best knowledge, Mostree is the first PDTE protocol that simultaneously achieves the above two properties.
- We implement Mostree and measure its performance. The experiment results show Mostree is highly communication-efficient compared with the state-of-the-art.

2 RELATED WORK

We categorize PDTE protocols based on the adversary models: semi-honest PDTE protocols and malicious PDTE protocols. Table 1 provides a comprehensive comparison of some representative PDTE

---

[1]Informally, privacy requires that a protocol reveals nothing except the protocol output and any allowed information. Correctness requires the computation to be done correctly.

schemes, *e.g.*, PDTE in two-party settings [4, 6, 11, 24, 25, 30, 35, 36, 39] and PDTE in three-party settings [14, 23], taking into account their performance and security guarantees.

Semi-honest PDTE Protocols. Most existing PDTE protocols, *e.g.*, [5, 6, 11, 17, 21, 25, 35, 39], are only secure against semi-honest adversaries. Moreover, they come with heavy computation and/or communication overhead. In [22], Ishai and Paskin evaluate DTs via homomorphic encryption (HE), which brings huge computation burden and linear communication cost. Protocols proposed in [11] and [5] require linear communication overhead since the tree transferred can never be re-used due to the access pattern leakage. Later, Bost *et al.* [6] encode the tree into a high-degree polynomial, simplifying the communication process but requiring costly fully HE. Wu *et al.* [39] avoid computing expensive polynomials used in [6] by sending an encrypted permuted tree to FO. Tai *et al.* [35] adopt the same strategies as [39], but rather than transmitting the entire tree, MO computes and sends a value for each path, which they call path cost. By doing so, they save communication costs. Following path costs concept from [35], many subsequent works [17] [25] have been proposed. The work proposed by Kiss *et al.* [25] mainly concentrates on exploring the influence of different combinations of HE and MPC on the performance of PDTE.

Sublinear complexity. As indicated in Table 1, most PDTE protocols have linear complexity in both communication and computation. This limitation renders them impractical for evaluating large decision trees containing millions of nodes [12]. Researchers are keen to explore PDTE protocols with overhead sublinear to the tree size. Tueno *et al.* [36] design the first sublinear protocol with semi-honest security in the two-party setting. Their idea is to represent the tree in an array. The tree construction allows MO to obliviously select the tree node in each tree level if the node index is shared between FO and MO. The selection process is called Oblivious Array Index (OAI). They instantiate the OAI approach by ORAM, which results in $O(d^4)$ communication cost and d^2 rounds for complete trees. Joye and Salehi [24] also work on a semi-honest two-party scenario but employ a different strategy to achieve oblivious selection. They observe there is only one node to be selected from each tree level, and thus their method is obliviously selecting a node from $n = 2^l$ nodes where l is the sitting tree level. For the whole tree evaluation, only d comparisons are required, where d is the depth of the tree. However, the communication is still linear to the tree size. Ma *et al.* [30] follows the idea of [11] to send the encrypted tree to FO. Rather than using a complex garbled circuit, they employ secret sharing to protect the tree. In each level, FO performs Oblivious Transfer (OT) with MO to retrieve the nodes to be evaluated, which optimizes both communication and computation overhead of [11]. However, similar to [11], FO can still learn extra information from the access pattern leakage. The work proposed in [23] achieves sublinear communication by leveraging function secret sharing. However, this work is only designed to defend against semi-honest adversaries.

Malicious Secure PDTE Protocols. Existing works [30, 35, 39] can protect the model from a malicious FO. The idea from [35, 39] is FO additionally sends zero-knowledge proofs (ZKP) to MO to prove the correctness of inputs. Both protocols result in heavy linear computation and communication overhead. In work [30], Ma

et al. replace Garbled Circuit (GC) protocol with its maliciously secure version [2] and employ commitment and ZKP to constrain the parties' behavior. However, this protocol still suffers from linear communication costs because the tree can never be reused after evaluation. All the above three works cannot guarantee correctness when MO is malicious. Damgård *et al.* [14] present a PDTE protocol using SPDZ$_{2^k}$ in the dishonest majority setting (up to $n - 1$ corruption out of n parties). Similar to previous works [17, 35, 39], they have MO and FO to securely perform attribute selection and comparison for each tree node, resulting in $O(mn)$ computation and communication, where m is the tree size, and n is the feature size. Notably, their protocol can protect both privacy and correctness and achieves higher security than ours since they work in the dishonest majority setting.

Recent DPF-over-RSS Techniques. We note several constructions that utilize DPFs over RSS [15, 23, 37, 38] were proposed recently. Waldo [15] use DPFs over RSSs to design a privacy-preserving database query. However, Waldo relies on honest clients to generate and distribute the DPF keys, whereas, in Mostree, the keys are generated and distributed by a possibly malicious party, requiring additional checks to ensure correctness. The DPF-over-RSS technique is also used in [23] and [37], but they only achieve semi-honest security. Pika [38] uses DPFs over ring-based RSS and achieves malicious security. Our scheme operates specifically on boolean-based RSS, which enables the design of more efficient error detection mechanisms. We will show more in the following sections.

3 BACKGROUND

3.1 Decision Trees Evaluation

A decision tree is usually represented as a binary tree where its inner nodes are *decision nodes* and its leaves are *classification nodes*. A decision node consists of a threshold and the index of the corresponding feature attribute. A classification node contains a classification label. Given a feature vector, DTE starts from the root. It compares a feature value with the threshold value and decides which child to visit based on the result (*i.e.*, 1 for the left child and 0 for the other). The evaluation continues until it reaches a leaf, from which the evaluation outputs a label as the classification result.

3.2 Cryptographic Primitives

Notations. We use P_i to denote the ith party, where $i \in \{0, 1, 2\}$ and we write P_{i-1} and P_{i+1} as its "previous" and "subsequent" parties, respectively. Typically, P_{i-1} is P_2 when $i = 0$ and P_{i+1} is P_0 when $i = 2$. We interchangeably use $\mathbb{F}_2^k$ and $\mathbb{F}_{2^k}$ to represent the data in $\{0, 1\}^k$, depending on the context. Addition in $\mathbb{F}_2^k$ and $\mathbb{F}_{2^k}$ corresponds to bit-wise XOR operation. We write $\mathbb{F}_{2^k} \cong \mathbb{F}[X]/f(X)$ for some monic, irreducible polynomial $f(X)$ of degree k. We denote the set $\{0, \cdots, j-1\}$ as $[j]$. Given two vectors $\vec{x}$ and $\vec{y}$, we use $\vec{x} \odot \vec{y}$ to denote inner-product computation between $\vec{x}$ and $\vec{y}$.

Secret Sharing. We use secret sharing for secure computation.

- $\binom{n}{n}$**-sharing** $[\![x]\!]$. We use $[\![x]\!]$ to denote $x \in \mathbb{F}$ is shared in n parties by $\binom{n}{n}$-sharing, where P_i holds a share $[\![x]\!]_i \in \mathbb{F}$ satisfying $x = \sum_{i \in [n]} [\![x]\!]_i$. We use $n = 2$ and 3 in this paper.
- $\binom{3}{2}$**-sharing** $\langle x \rangle$. We use $\langle x \rangle$ to denote x is shared by $\binom{3}{2}$-sharing, also known as *replicated secret-sharing* (RSS). In RSS sharing, we

denote $\langle x \rangle = (x_0, x_1, x_2)$, where each party P_i ($i \in \{0, 1, 2\}$) holds two shares ($[\![x]\!]_i, [\![x]\!]_{i-1}$) such that $[\![x]\!]_0 + [\![x]\!]_1 + [\![x]\!]_2 = x$. Naturally, given a public value v, it can be shared as $\langle v \rangle = (0, 0, v)$. Different from $\binom{3}{3}$-sharing, any two parties in $\binom{3}{2}$-sharing can reconstruct the secret.

We extend the above definition to vectors. We use $\vec{x} \in \mathbb{F}^m$ to denote an m-dimensional vector. Accordingly, we use $[\![\vec{x}]\!]$ and $\langle \vec{x} \rangle$ to denote a $\binom{n}{n}$-sharing and $\binom{3}{2}$-sharing of a vector $\vec{x}$, respectively.

Secure Computation over RSS Sharing. RSS sharing supports the following (semi-honest) addition and multiplication operations:

- $\langle z \rangle \leftarrow \langle x \rangle + \langle y \rangle$: For $i \in [3]$, P_i computes ($[\![z]\!]_i = [\![x]\!]_i + [\![y]\!]_i, [\![z]\!]_{i-1} = [\![x]\!]_{i-1} + [\![y]\!]_{i-1}$).
- $\langle z \rangle \leftarrow \langle x \rangle + c$: P_0 computes ($[\![z]\!]_0, [\![z]\!]_2$) = ($[\![x]\!]_0 + c, [\![x]\!]_2$); P_1 computes ($[\![z]\!]_1, [\![z]\!]_0$) = ($[\![x]\!]_1, [\![x]\!]_0 + c$); and P_2 computes ($[\![z]\!]_2, [\![z]\!]_1$) = ($[\![x]\!]_2, [\![x]\!]_1$).
- $\langle z \rangle \leftarrow c \cdot \langle x \rangle$: For $i \in [3]$, P_i computes ($[\![z]\!]_i, [\![z]\!]_{i-1}$) = ($c \cdot [\![x]\!]_i, c \cdot [\![x]\!]_{i-1}$).
- $\langle z \rangle \leftarrow \langle x \rangle \cdot \langle y \rangle$: P_i computes $[\![t]\!]_i \leftarrow [\![x]\!]_i \cdot [\![y]\!]_i + [\![x]\!]_{i-1} \cdot [\![y]\!]_i + [\![x]\!]_i \cdot [\![y]\!]_{i-1}$ for $i \in [3]$. ($[\![t]\!]_0, [\![t]\!]_1, [\![t]\!]_2$) forms a $\binom{3}{3}$-sharing $[\![t]\!]$. The parties additionally generate a $\binom{3}{3}$-sharing of zero, i.e., $r = [\![r]\!]_0 + [\![r]\!]_1 + [\![r]\!]_2 = 0$. P_i computes and sends $[\![z]\!]_i \leftarrow [\![t]\!]_i + [\![r]\!]_i$ to P_{i+1}, meanwhile receives $[\![z]\!]_{i-1}$ from P_{i-1}. P_i sets $\langle z \rangle_i \leftarrow ([\![z]\!]_i, [\![z]\!]_{i-1})$. A $\binom{3}{3}$-sharing $[\![r]\!]$ of zero can be generated non-interactively using a *PRF-based trick* [18]: each pair of parties (P_i, P_{i-1}) share a common key k_i^{prf} for a PRF $F : \mathcal{K} \times \mathcal{D} \rightarrow \mathbb{F}$. Given a session identifier $id \in \mathcal{D}$, P_i computes $[\![r]\!]_i \leftarrow F(k_i^{\mathsf{prf}}, id) - F(k_{i+1}^{\mathsf{prf}}, id)$. Clearly, $[\![r]\!]_0 + [\![r]\!]_1 + [\![r]\!]_2 = 0$.

Malicious Security Mechanisms for RSS. Mostree relies on some malicious security mechanisms and functionalities for RSS. To start with, we show a *consistent* property for RSS defined as below:

DEFINITION 1 (CONSISTENT RSS SHARING [29]). *Let* (a_1, b_1), (a_2, b_2), (a_3, b_3) *be the RSS shares held by* P_0, P_1, *and* P_2, *and* P_i *be corrupted. Then the shares are consistent if and only if* $a_{i+1} = b_{i+2}$.

One can check that consistency preserves for addition and scalar multiplication. Consistency also holds for secret-shared multiplication (i.e., $\langle x \rangle \cdot \langle y \rangle$), despite the fact that a malicious party can add an error (independent of the shared secrets) to resulting RSS sharing;[2] such an attack is known as an *additive attack* in secure computation. This is captured by Lemma 1 (from [29]).

LEMMA 1. *If* $\langle x \rangle$ *and* $\langle y \rangle$ *are two consistent RSS sharings and* $\langle z \rangle$ *is generated by executing the multiplication protocol on* $\langle x \rangle$ *and* $\langle y \rangle$ *in the presence of one malicious party, then* $\langle z \rangle$ *is a consistent sharing of either* $x \cdot y$ *or of some element* $z^* \in \mathbb{F}$.

Existing RSS-based 3PC relies on a *triple verification* [18, 31] to check the correctness of multiplication. Given a triple of RSS sharing ($\langle a \rangle, \langle b \rangle, \langle c \rangle$) with $a \cdot b = c$, the parties first open $\langle e \rangle = \langle x \rangle - \langle a \rangle$ and $\langle f \rangle = \langle y \rangle - \langle b \rangle$. Then the parties compute $\langle w \rangle \leftarrow e \cdot f + f \cdot \langle a \rangle + e \cdot \langle b \rangle + \langle c \rangle - \langle z \rangle$ and securely open $\langle w \rangle$ to check if $w = 0$.

Mostree uses some assumed ideal functionalities to achieve malicious security, including $\mathcal{F}_{\mathsf{rand}}$, $\mathcal{F}_{\mathsf{open}}$, $\mathcal{F}_{\mathsf{coin}}$, $\mathcal{F}_{\mathsf{recon}}$, $\mathcal{F}_{\mathsf{share}}$, $\mathcal{F}_{\mathsf{mul}}^{\mathbb{F}}$, and $\mathcal{F}_{\mathsf{CheckZero}}$ from [13, 18, 29]. All these functionalities can be securely computed with malicious security using well-established

protocols [13, 18, 29]. We also provide the corresponding protocols in Appendix B for completeness.

- $\mathcal{F}_{\mathsf{rand}}(\mathbb{F})$: sample a random $r \xleftarrow{\$} \mathbb{F}$ and share $\langle r \rangle$ between three parties.
- $\mathcal{F}_{\mathsf{open}}(\langle x \rangle)$: on inputting a *consistent* RSS-sharing $\langle x \rangle$, reveal x to all the parties.
- $\mathcal{F}_{\mathsf{coin}}(\mathbb{F})$: sample a random $r \xleftarrow{\$} \mathbb{F}$ and output r to three parties.
- $\mathcal{F}_{\mathsf{recon}}(\langle x \rangle, i)$: on inputting a *consistent* RSS-sharing $\langle x \rangle$ and a party index i, send x to P_i.
- $\mathcal{F}_{\mathsf{share}}(x, i)$: on inputting a secret x held by P_i, share $\langle x \rangle$ between the parites.
- $\mathcal{F}_{\mathsf{mul}}^{\mathbb{F}}(\langle x \rangle, \langle y \rangle, e)$: take two RSS-sharing $\langle x \rangle$ and $\langle y \rangle$ for $x, y \in \mathbb{F}$ and an additive error $e \in \mathbb{F}$ specified by the adversary $\mathcal{A}$, share $\langle x \cdot y + e \rangle$ between three parties.
- $\mathcal{F}_{\mathsf{CheckZero}}(\langle x \rangle)$: take $\langle x \rangle$ as input and output True if $x = 0$ and False otherwise.

In addition, whenever three-party RSS-based secure computation (3PC for short) is used in a black-box manner (e.g., secure comparison and secure MUX in Mostree), we will use a 3PC ideal functionality $\mathcal{F}_{\mathsf{3pc}}^{\mathbb{F}}$ directly for simplicity, which ensures privacy and correctness for secure computation over $\mathbb{F}$ against a malicious adversary.

Distributed Point Function. A point function $f_{\alpha, \beta} : \mathcal{D} \rightarrow \mathcal{R}$ outputs β only if $x = \alpha$ and outputs 0 for all $x \in \mathcal{R} \setminus \{\alpha\}$. A two-party distributed point function (DPF) scheme [7, 8, 19] can share a point function using two succinct correlated keys (with size sublinear in $|\mathcal{D}|$). Def. 2 shows the formal definition.

DEFINITION 2 (DISTRIBUTED POINT FUNCTION). *A two-party DPF scheme* $\Pi_{\mathsf{dpf}} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{BatchEval})$ *consists of three algorithms:*

- $(k_0^{\mathsf{dpf}}, k_1^{\mathsf{dpf}}) \leftarrow \mathsf{Gen}(1^\kappa, f_{\alpha, \beta})$. *Given a security parameter* 1^κ *and a point function* $f_{\alpha, \beta} : \mathcal{D} \rightarrow \mathcal{R}$, *outputs a two keys* $(k_0^{\mathsf{dpf}}, k_1^{\mathsf{dpf}})$, *each for one party.*
- $[\![y]\!]_i \leftarrow \mathsf{Eval}(k_i^{\mathsf{dpf}}, x)$. *Given a key* k_i^{dpf} *for party* P_i ($i \in \{0, 1\}$), *and an evaluation point* $x \in \mathcal{D}$, *outputs a group element* $[\![y]\!]_i \in \mathcal{R}$ *as the share of* $f(x)$ *for* P_i.
- $\{[\![y_j]\!]_i\}_{j \in [L]} \leftarrow \mathsf{BatchEval}(k_i^{\mathsf{dpf}}, \{x_j\}_{j \in [L]})$. *This algorithm performs evaluation over a batch of* L *inputs* $\{x_j\}_{j \in [L]}$, *outputs a set of shares* $\{[\![y_j]\!]_i\}_{j \in [L]}$, *where* $[\![y_j]\!]_i \leftarrow \mathsf{Eval}(k_i^{\mathsf{dpf}}, x_j)$.

A DPF scheme should ensure secrecy and correctness properties. Roughly, secrecy requires that one party cannot learn any more information from its DPF key. Correctness requires $\mathsf{Eval}(k_0^{\mathsf{dpf}}, x) + \mathsf{Eval}(k_1^{\mathsf{dpf}}, x) = f_{\alpha, \beta}(x)$ always holds. We refer to Appendix A for the formal definition.

Verifiable Distributed Point Function. When the party responsible for generating and distributing DPF keys is malicious, it may generate incorrect keys. To prevent this, verifiable DPFs (VDPFs) [16] additionally provide a verifiable property, defined as follows:

DEFINITION 3 (VERIFIABLE DPF). *A verifiable distributed point function scheme* $\mathsf{VDPF} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{BatchEval}, \mathsf{Verify})$ *contains four algorithms. :*

- Gen *and* Eval: *Same as the definition in DPF.*

---

[2]Suppose P_i is the corrupted party. P_i can instead send $z_i \leftarrow t_i + r_i + e$ to P_{i+1}.The parties will share $\langle x \cdot y + e \rangle$ instead of $\langle x \cdot y \rangle$.

- $(\llbracket y_j \rrbracket_i, \pi_i) \leftarrow \mathsf{BatchEval}(\mathsf{k}_i^{\mathsf{dpf}}, \{x_j\}_{j \in [L]})$. *This algorithm performs batch evaluation with an additional output π_i which is used to verify the correctness of the output.*
- Accept/Reject $\leftarrow \mathsf{Verify}(\pi_0, \pi_1)$. *This is a protocol run between the DPF evaluators, which takes the proofs π_0 and π_1 as the inputs and outputs either* Accept *or* Reject.

Security Definition. We follow the simulation-based security model in the three-party honest-majority setting [1, 18]. We refer to Appendix A for a formal definition.

4 OVERVIEW OF MOSTREE

This section shows the threat model and overview of Mostree.

4.1 Threat Model

System Model. Mostree contains three parties: a model owner (MO), a feature owner (FO), and one assistant computing party (CP). Mostree consists of a one-time setup protocol and an evaluation protocol. In the setup protocol, MO uses RSS sharing to share a tree model among the three parties. Whenever FO wants to perform a PDTE query, it shares its feature vector among the three parties. The parties jointly run the evaluation and reconstruct the classification result to FO, completing the PDTE task.

Mostree works in the three-party honest-majority setting [1, 18, 29, 34] where at most one party is malicious and other two parties are honest. The same assumption is also accepted by many recent privacy-preserving works [15, 28, 32, 34] in order to trade a better efficiency that cannot be obtained by two-party protocols. Mostree ensures privacy and correctness with abort in the presence of a malicious adversary under this model.

Remark. A secure computation protocol ensures private and correct computation once the inputs are fed into the protocol. We do not consider attacks from manipulated inputs or leakage from PDTE protocol output (*e.g.*, inference attacks and model stealing attacks).

4.2 Approach Overview

Mostree first encodes a DT as an array, then transforms DTE to a traversal algorithm over arrays. Mostree focuses on designing protocols to securely evaluate the DT algorithm over the encoded tree and feature arrays.

Evaluation over Encoded Tree Array. Mostree encodes a DT as a multi-dimensional array $\vec{\mathrm{T}}$. Fig. 1 shows the tree encoding method. Each array element corresponding to a tree node contains five values: left child index l, right child index r, threshold value t, feature ID v, and classification label c (only valid for leaf nodes). One can perform DTE over the encoded tree array and a feature vector. The evaluation runs at most d iterations, where d is the maximal DT depth. The evaluation starts from $\vec{\mathrm{T}}[0]$. In each iteration, the algorithm fetches $\vec{\mathrm{X}}[v]$ and compares it with the current threshold value t, and decides to go left or right child according to the comparison result. The algorithm terminates and outputs its label c as the classification result when reaching a leaf.

Private and Correct Tree Evaluation. Mostree aims to evaluate the tree *privately* and *correctly* in the presence of a malicious adversary.

We address the following security issues with efficient solutions. ❶ *Hiding secret values*: a PDTE protocol should hide each element of
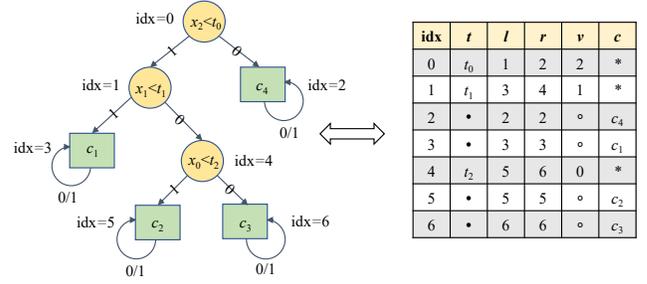

Figure 1: Encoding Tree as an Tree Array $\vec{\mathrm{T}}$ by Breadth-First Search (BFS): $\bullet \in [2^k]$, $\circ \in [n]$ **and** $* \in [2^k]$ **where** k **is the bit length of single value and** n **represents feature dimension.**

$\vec{\mathrm{T}}$ and $\vec{\mathrm{X}}$ as well as all intermediate states/values. ❷ *Hiding running time*: a PDTE protocol should hide the decision path length. If MO learns the number of evaluation rounds, it can infer the accessed path. ❸ *Hiding access pattern*: a PDTE protocol should hide access pattern over $\vec{\mathrm{T}}$ and $\vec{\mathrm{X}}$. Specifically, no party should learn which child and feature are taken during each evaluation round. ❹ *Ensuring correct classification*: the FO must receive a correct classification result if the protocol completes.

To ensure ❶, we choose RSS-based boolean sharing to share all data among three parties, since it matches well with bit-wise computation. To ensure ❷, we use the encoding trick from Bai *et al.* [4]. As shown in Fig 1, the idea is to encode two circles for each leaf node by setting children indexes as the leaf itself, thus the evaluation will be redirected back to itself once reached. The parties always run $d_{\mathsf{pad}} \geq d$ iterations for any query. This hides running time meanwhile ensures the correctness of classification.

Achieving ❸ under the constraint of sublinear communication and malicious security is challenging. What we want is a sublinear-communication oblivious selection (OS) functionality that allows the parties to obliviously select and share a desired tree node in a secret-shared fashion. We propose two efficient OS protocols for RSS sharing. However, both protocols are not totally maliciously secure. They are all vulnerable to additive attacks, which compromise the correctness property. To address this issue, we design a set of lightweight consistency checks, exploiting some nice properties of the proposed primitives and reusing existing RSS-based malicious secure mechanisms and functionalities. Combining them together, Mostree ensures ❹.

5 THE MOSTREE PROTOCOL

This section details techniques in Mostree. We first propose two oblivious selection (OS) protocols both with constant online communication. Then we combine OS protocols with our tree encoding method and existing 3PC ideal functionalities to design Mostree.

5.1 Oblivious Selection from Pure RSS

Mostree uses Oblivious Selection (OS) protocols to perform oblivious node selection. Our OS protocols aim to compute functionality $\mathcal{F}_{\mathsf{os}}$ securely: On inputting an RSS-shared vector $\langle \vec{\mathrm{T}} \rangle$ and an RSS-shared index $\langle idx \rangle$, receive an error $e \in \mathbb{F}$ from the adversary, share $\langle \vec{\mathrm{T}}[idx] + e \rangle$ (with rerandomization) between the parties. Here $\mathcal{F}_{\mathsf{os}}$ is

up to additive attacks. Looking ahead, this imperfect $\mathcal{F}_{os}$ definition suffices for our purpose and achieves our efficiency goals.

Oblivious Selection Using Inner-product. We use inner product computation for oblivious selection. Specifically, given an RSS-sharing $\langle idx \rangle$ and an RSS-sharing vector $\langle \vec{T} \rangle$ and assume the parties can somehow share a unit vector $\langle \vec{u} \rangle$ such that $\vec{u}$ comprises all 0s except for a single 1 at idx, selection can be easily made by computing $\langle \vec{T}[idx] \rangle \leftarrow \langle \vec{u} \odot \vec{T} \rangle$.

Achieve constant resharing communication. Inner-product computation requires m multiplications. Multiplication between two RSS sharings involves a resharing phase that reshares a $\binom{3}{3}$-sharing back to a $\binom{3}{2}$-sharing. Resharing requires communication; thus, trivially invoking m multiplications would incur linear communication. We use an optimization trick to reduce the overhead: the parties first sum all intermediate $\binom{3}{3}$-sharings of m multiplication and then perform resharing only once. Now we can achieve constant communication for inner-product computation. However, generating $\langle \vec{u} \rangle$ from $\langle idx \rangle$ requires linear communication: the parties compute $\langle \vec{u} \rangle$ such that $\vec{u}[j] \leftarrow (j == idx)$ for $j \in [m]$, which requires m invocations of secure equality comparison with linear communication.

Achieve constant online communication. We use a derandomization technique [3, 4, 9, 23] to further reduce online communication. In particular, suppose the parties have already shared a unit vector $\vec{v}$ whose non-zero element appears at a random position rdx. When obtained idx, the parties compute $\langle \Delta \rangle \leftarrow \langle rdx \oplus idx \rangle$ and open Δ by revealing shares to other parties. Note that the vector length must be power-of-2 to enable this derandomization. Now the parties can define $\langle \vec{u} \rangle$ such that $\langle \vec{u}[j] \rangle \leftarrow \langle \vec{v}[j \oplus \Delta] \rangle$ for $j \in [m]$. Clearly, $\vec{u}[j] = 1$ only for $j = idx$. Then they can use $\langle \vec{u} \rangle$ and $\langle \vec{T} \rangle$ to perform selection, as mentioned above. Since generating $\langle \vec{v} \rangle$ can be moved to the offline phase, the online communication is reduced to constant.

Malicious Security with up to Additive Attacks. To achieve malicious security for the offline phase, we rely on existing malicious secure equality comparison protocol to generate the unit vector $\langle \vec{v} \rangle$. This part is not our focus and we use the existing ideal functionality $\mathcal{F}_{3pc}^{\mathbb{F}_2}$ directly. We also provide the concrete secure equality comparison protocol Π_{eq} in Appendix B for completeness.

The online phase requires opening $\langle \Delta \rangle = \langle rdx \oplus idx \rangle$, adjusting $\langle v \rangle$, and performing an inner product computation. However, a malicious party can add errors in the online phase to break correctness. First, we assume both $\langle idx \rangle$ and $\langle rdx \rangle$ are *consistent* RSS sharing; we will show this assumption holds when using OS in Mostree. Thus opening $\langle \Delta \rangle$ can be done *correctly* using $\mathcal{F}_{open}$ with malicious security. The challenge is to check the correctness of m multiplications. Note that resharing for multiplication is subject to additive attacks, allowing the adversary to inject an error into the multiplication sharing. Existing malicious secure RSS-based 3PC protocols [18, 31] perform a triple-based correctness check for *each* multiplication (refer to section 3.2), which incurs linear communication for inner-product.

Our observation is that we can omit the correctness check for multiplication at this stage, which explains why we formally capture this attack in the definition of $\mathcal{F}_{os}$; we rely on a communication-efficient mechanism to detect additive errors at a later stage. As a

---

Parameters: An RSS-sharing array $\langle \vec{T} \rangle$ where each element $\vec{T}[j] \in \mathbb{F}_{2^\ell}$ for $j \in [m]$; $m = 2^{\ell_m}$ where ℓ_m denotes the number of bits of m; an RSS-sharing index $\langle idx \rangle$ for $idx \in [m]$; a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}_{2^\ell}$. A common session identifier sid, and each pair of two parties (P_{i-1}, P_i) hold a common PRF key $k_i^{prf} \in \mathcal{K}$ for $i \in [3]$.

[Preprocess] The parties generate a random unit vector sharing $\langle \vec{v} \rangle$ with a non-zero index sharing $\langle rdx \rangle$.

1. The parties call $\langle rdx \rangle \leftarrow \mathcal{F}_{rand}(\mathbb{F}_{2^{\ell_m}})$.
2. The parties call $\mathcal{F}_{3pc}^{\mathbb{F}_2}$ to compute an RSS-shared unit vector $\langle \vec{v} \rangle$, where $\vec{v}[rdx] = 1$ and $\vec{v}[j] = 0$ for all $j \neq rdx$.
3. The parties store $(\langle rdx \rangle, \langle \vec{v} \rangle)$ for online computation.

[Selection] Upon input (sid, $\langle \vec{T} \rangle, \langle idx \rangle$), do the following:

1. Fetch a preprocessed random unit vector RSS-sharing $(\langle rdx \rangle, \langle \vec{v} \rangle)$.
2. Compute $\langle \Delta \rangle \leftarrow \langle rdx \rangle \oplus \langle idx \rangle$ and open $\Delta \leftarrow \mathcal{F}_{open}(\langle \Delta \rangle)$. If the open fails, abort.
3. The parties define $\langle \vec{u} \rangle$ where $\langle \vec{u}[j] \rangle \leftarrow \langle \vec{v}[j \oplus \Delta] \rangle$.
4. Compute oblivious selection using inner-product as follows:
   1) The parties compute a $\binom{3}{3}$-sharing $[\![t]\!] \leftarrow \sum_{j \in [m]} [\![\vec{T}[j] \cdot \vec{u}[j]]\!]$ using $\langle \vec{T} \rangle$ and $\langle \vec{u} \rangle$.
   2) For $i \in [3]$: P_i computes $[\![r]\!]_i \leftarrow F(k_i^{prf}, sid) - F(k_{i-1}^{prf}, sid)$. P_i defines $[\![s]\!]_i = [\![t]\!]_i + [\![r]\!]_i$ and sends $[\![s]\!]_i$ to P_{i+1}.
   3) For $i \in [3]$, party P_i defines $\langle s \rangle$ such that $\langle s \rangle_i \leftarrow ([\![s]\!]_i, [\![s]\!]_{i-1})$.
5. Output $\langle s \rangle$.

Figure 2: Protocol $\Pi_{rss\text{-}os}$ with Additive Attacks

benefit, the resulting OS protocol keeps constant online communication.

The RSS-based OS Protocol. The RSS-based OS protocol $\Pi_{rss\text{-}os}$ is formally described in Fig. 2. Security of $\Pi_{rss\text{-}os}$ is from Theorem 1, with a proof from Appendix C.1.

THEOREM 1. *$\Pi_{rss\text{-}os}$ securely computes $\mathcal{F}_{os}$ in the $(\mathcal{F}_{rand}, \mathcal{F}_{3pc}^{\mathbb{F}_2}, \mathcal{F}_{open})$-hybrid model in the presence of a malicious adversary in the three-party honest-majority setting, assuming F is a secure PRF.*

5.2 Oblivious Selection from DPF and RSS

Protocol $\Pi_{rss\text{-}os}$ enjoys constant online communication but linear offline communication. We propose another OS protocol with constant online communication and sublinear offline communication.

A Semi-honest DPF-based OS Protocol. The idea is similar to $\Pi_{rss\text{-}os}$. The difference is we apply DPFs over RSS sharings to perform inner-product computation. We first review how to construct a semi-honest OS protocol, then securely enhance it to realize $\mathcal{F}_{os}$.

Apply DPF over RSS sharings. Recall that for an RSS-shared tree array $\langle \vec{T} \rangle$, P_i and P_{i+1} hold a common share vector $[\![\vec{T}]\!]_i$. P_i and P_{i+1} can apply a DPF over the common $[\![\vec{T}]\!]_i$ to perform OS operations. Specifically, a trusted dealer generates a pair of DPF keys $(k_0^{dpf}, k_1^{dpf}) \leftarrow Gen(1^\kappa, f_{idx,1})$ and sends k_b^{dpf} to P_{i+b} for $b \in \{0,1\}$, respectively. Each party locally expands its DPF key over the domain $[m]$. By doing this, P_i and P_{i+1} share a vector $[\![\vec{u}]\!]$ in $\binom{2}{2}$-sharing, where $\vec{u}$ comprises all 0s except for a single 1 appearing at coordinate idx. Each party locally computes inner-product $[\![[\![\vec{T}]\!]_i[idx]]\!] \leftarrow \sum_{j \in [m]} [\![\vec{u}[j]]\!] \cdot [\![\vec{T}]\!]_i[j]$, a $\binom{2}{2}$-sharing of $[\![\vec{T}]\!]_i[idx]$ between P_i and P_{i+1}. The above process is repeatedly run for any two parties out of three. At the end of the three-round execution,

Note:

1). We use ▢ to represent a DPF batch evaluation process with additional k_0^{dpf} and k_1^{dpf} from two different parties as inputs where $(k_0^{dpf}, k_1^{dpf}) \leftarrow$ Gen($1^\kappa, f_{idx,1}$).

2). Any two parties out of three jointly perform the DPF batch evaluation process.

3). P_i computes $\llbracket [\vec{T}]_i[idx] \rrbracket_0 + \llbracket [\vec{T}]_{i+2}[idx] \rrbracket_0$ and

P_{i+1} computes $\llbracket [\vec{T}]_{i+1}[idx] \rrbracket_0 + \llbracket [\vec{T}]_i[idx] \rrbracket_1$ and

P_{i+2} computes $\llbracket [\vec{T}]_{i+1}[idx] \rrbracket_1 + \llbracket [\vec{T}]_{i+2}[idx] \rrbracket_1$ to get a (3,3)-sharing of $\vec{T}[idx]$.

4). Three parties reshare (3,3)-sharing $\vec{T}[idx]$ to its RSS sharing.

Figure 3: OS from DPF and RSS

each party will hold two shares of $\binom{2}{2}$-sharing (since each party will run oblivious selection twice with the other two parties). After that, the parties locally sum up two local shares of $\binom{2}{2}$-sharing. In this manner, three parties jointly produce a $\binom{3}{3}$-sharing of $\vec{T}[idx]$, which can be reshared back to an RSS-sharing $\langle \vec{T} \rangle$ using the PRF-based re-sharing trick [18] (also refer to Section 3.2 and Fig. 2). Fig. 3 sketches the DPF-based OS protocol.

To remove the trusted party, we let the non-involved party P_{i+2} generate the DPF keys. However, we cannot let P_{i+2} know idx as this will violate privacy. We apply the same derandomization technique used in $\Pi_{rss\text{-}os}$ to resolve the issue. Specifically, in the offline phase, P_{i+2} generates a pair of DPF keys for a point function $f_{rdx,1}$ with a randomly chosen index rdx, and P_{i+2} shares the DPF keys as well as rdx (using $\binom{2}{2}$-sharing) between P_i and P_{i+1}. In the online phase, P_i and P_{i+1} each expands its DPF key locally to share a $\binom{2}{2}$-sharing of a unit vector $\llbracket \vec{v} \rrbracket$. P_i and P_{i+1} then open $\Delta = rdx \oplus idx$ and compute $\langle \vec{u} \rangle$ such that $\vec{u}[j] \leftarrow \vec{v}[j \oplus \Delta]$ for $j \in [m]$. With $\llbracket \vec{u} \rrbracket$ and $\llbracket \vec{T} \rrbracket_i$, P_i and P_{i+1} can locally compute and share $\llbracket \vec{T}_i[idx] \rrbracket$ in $\binom{2}{2}$-sharing. Note that the PDF keys are of size $O(\kappa \log m)$ where κ is the security parameter, and the key generation can be moved to the offline phase.[3] Efficiency-wise, this requires constant online communication and sublinear offline communication.

Attacks. The DPF-based OS protocol involves three interactive parts: 1) DPF key generation and distribution; 2) opening Δ; and 3) DPF-based local evaluation and resharing. Now we discuss malicious attacks for each part.

Firstly, we discuss attacks from a corrupted key generator P_{i+2}. There are two possible attacks: 1) *Incorrect DPF keys*. Instead of generating a pair of well-formed DPF keys, the corrupted party may generate the keys for a point function $f_{rdx,s}$ with $s \neq 1$; Indeed, the corrupted party may take arbitrary key generation strategies. 2) *Incorrect index sharing*. Even the DPF keys for $f_{rdx,1}$ are correctly generated and shared, the malicious P_{i+2} may share an inconsistent index $rdx^* \neq rdx$ between P_i and P_{i+1}. As a consequence, this attack results in an incorrect opening of $\Delta \leftarrow rdx \oplus idx \oplus e$ where the error $e = rdx \oplus rdx^*$; looking ahead, this corresponds to an incorrect node being selected in Mostree. Secondly, we notice that

even if the DPF keys are correctly generated and the index rdx is correctly shared, how to correctly reconstruct Δ to P_i and P_{i+1} is still a question. If only P_i and P_{i+1} are involved in the reconstruction of Δ, it is impossible to ensure correctness because an adversary who corrupts either P_i or P_{i+1} can always add errors during the reconstruction process. Lastly, a corrupted party may maliciously add errors before resharing the $\binom{3}{3}$-shared selected secret, resulting in an additive attack. We note that the last attack is allowed by $\mathcal{F}_{os}$.

Defences. The above attacks only compromise correctness, not privacy. We propose a set of consistency checks to ensure correctness in the above protocol. Now we present intuitions, and the formal description is shown by protocol $\Pi_{dpf\text{-}os}$ in Fig. 4.

Parameters: An RSS-sharing for array $\langle \vec{T} \rangle$ where each element $\vec{T}[j] \in \mathbb{F}_{2^\ell}$ for $j \in [m]$; $m = 2^{\ell_m}$ where ℓ_m denotes the number of bits of m; an RSS-sharing index $\langle idx \rangle$ for $idx \in [m]$; point function $f_{\alpha,\beta} : [m] \to \mathbb{F}_{2^\ell}$; each pair of two parties (P_i, P_{i+1}) hold a common key $k_i^{prf} \in \{0,1\}^\kappa$ for $i \in [3]$ for a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}_{2^\ell}$. A common session identifier sid.

[Preprocess] The parties run the following protocol to generate sufficiently many DPF keys.

1. P_{i+2} samples a random value $rdx \xleftarrow{\$} \mathbb{Z}_m$ and locally computes a pair of DPF keys $(k_0^{dpf}, k_1^{dpf}) \leftarrow$ VDPF.Gen($1^\kappa, f_{rdx,1}$). P_{i+2} samples $\llbracket rdx \rrbracket_0$ and $\llbracket rdx \rrbracket_1$ such that $\llbracket rdx \rrbracket_0 + \llbracket rdx \rrbracket_1 = rdx$ over $\mathbb{Z}_m$. P_{i+2} sends $(k_0^{dpf}, \llbracket rdx \rrbracket_0)$ and $(k_1^{dpf}, \llbracket rdx \rrbracket_1)$ to P_i and P_{i+1}, respectively.
2. P_i and P_{i+1} run the DPF key verification protocol from [16] to check the well-formness of DPF keys. If the check fails, abort.
3. P_i and P_{i+1} each expand its DPF key over domain $[m]$ to jointly produce a shared vector $\llbracket \vec{v} \rrbracket$ in the $\binom{2}{2}$-sharing. For $b \in \{0,1\}$, P_{i+b} locally computes:

$$\llbracket t \rrbracket_b \leftarrow \sum_{j \in [m]} \llbracket \vec{v}[j] \rrbracket_b, \llbracket s \rrbracket_b \leftarrow \llbracket rdx \rrbracket_b - \sum_{j \in [m]} j \cdot \llbracket \vec{v}[j] \rrbracket_b.$$

P_i and P_{i+1} open $\llbracket t \rrbracket$ and $\llbracket s \rrbracket$ and check if $t = 1$ and $s = 0$. If the check fails, abort.
4. P_{i+2} stores $(rdx, \llbracket rdx \rrbracket_0, \llbracket rdx \rrbracket_1)$, P_i stores $(k_0^{dpf}, \llbracket rdx \rrbracket_0)$, and P_{i+1} stores $(k_1^{dpf}, \llbracket rdx \rrbracket_1)$.

[Selection] Upon input $(sid, \langle \vec{T} \rangle, \langle idx \rangle)$, do:

1. For $i \in [3]$:
   1) For $b \in \{0,1\}$, P_{i+b} fetches a DPF key with the index share $(k_b^{dpf}, \llbracket rdx \rrbracket_b)$ distributed by P_{i+2} in the offline phase. Then P_i sets $\langle rdx \rangle_i = (0, \llbracket rdx \rrbracket_0)$, P_{i+1} sets $\langle rdx \rangle_{i+1} = (\llbracket rdx \rrbracket_1, 0)$, and P_{i+2} sets $\langle rdx \rangle_{i+2} = (\llbracket rdx \rrbracket_0, \llbracket rdx \rrbracket_1)$.
   2) Three parties compute $\langle \Delta \rangle \leftarrow \langle rdx \rangle \oplus \langle idx \rangle$ and reconstruct Δ *only* to P_{i+1} and P_{i+2} using $\mathcal{F}_{recon}$. If reconstruction fails, abort.
   3) P_i and P_{i+1} share a $\binom{2}{2}$-sharing $\llbracket [\vec{T}]_i[idx] \rrbracket \leftarrow \sum_{j \in [m]} \llbracket \vec{T} \rrbracket_i[j] \cdot \llbracket \vec{v}[j \oplus \Delta] \rrbracket$; P_{i+b} holds $\llbracket [\vec{T}]_i[idx] \rrbracket_b$ for $b \in \{0,1\}$.
2. Now $\vec{T}_i[idx]$ is shared between (P_i, P_{i+1}) in $\binom{2}{2}$-sharing for $i \in [3]$(each party will hold two $\binom{2}{2}$-shares after the DPF-based evaluation). Each party locally sums up its two shares of $\binom{2}{2}$-sharing to generate a $\binom{3}{3}$-sharing $\llbracket \vec{T}[idx] \rrbracket$.
3. For $i \in [3]$, P_i computes $\llbracket z \rrbracket_i \leftarrow \llbracket \vec{T}[idx] \rrbracket_i + F(k_i^{prf}, sid) - F(k_{i-1}^{prf}, sid)$ and sends $\llbracket z \rrbracket_i$ to P_{i+1}. P_i receives $\llbracket z \rrbracket_{i-1}$ from P_{i-1}.
4. For $i \in [3]$, P_i defines $\langle z \rangle_i \leftarrow (\llbracket z \rrbracket_i, \llbracket z \rrbracket_{i-1})$.

Figure 4: Protocol $\Pi_{dpf\text{-}os}$ from DPF and RSS

---

[3]The concrete offline efficiency is poor for small trees. In this case, P_{i+2} directly shares the unit vector $\vec{v}$ between P_i and P_{i+1} instead of distributing a pair of DPF keys.

Detect malicious DPF keys. We use a VDPF scheme to prevent a corrupted DPF key generation party from distributing incorrect DPF keys. A VDPF scheme additionally allows two key DPF receivers to run an efficient check protocol to test whether the DPF keys are correctly correlated without leaking any information other than the validity of the keys. We use an existing VDPF construction [16] to perform the check for DPF keys.

Unfortunately, the check method from [16] only ensures the DPF keys correspond to a point function $f_{\alpha,\beta}$ for arbitrary β; it does not check $\beta = 1$. Besides, for our purpose, P_i and P_{i+1} must check whether the $\binom{2}{2}$-sharing $[\![rdx]\!]$ is consistent with the point function $f_{\alpha,\beta}$ shared by the DPF keys (*i.e.*, ensuring $rdx = \alpha$). To this, we additionally propose the following lightweight checks. First, to check $\beta = 1$, P_i and P_{i+1} locally expand their DPF keys to produce a shared a $\binom{2}{2}$-sharing vector $[\![\vec{v}]\!]$, and they check the sum of all the entries is equal to 1. Second, to check the shared index rdx is equal to α, the checking parties jointly compute $s = rdx - \sum_{j \in [m]} j \cdot \vec{v}[j]$ in a shared fashion, open s, and check if $s = 0$.[4] With these additional checks, we can ensure the correctness of DPF keys and the consistency of the shared index. The detailed description can be found from $\Pi_{\text{dpf-os}}$ (Step 2 - 3, preprocess protocol).

Reconstruct Δ correctly. To open Δ correctly, we observe that any $\binom{2}{2}$-sharing $[\![x]\!]$ can be converted to an RSS sharing $\langle x \rangle$ *without interaction*, where $[\![x]\!]$ is distributed by the dealer P_{i+2} and is shared between P_i and P_{i+1}. Our insight is that the dealer P_{i+2} knows x as well as the shares $[\![x]\!]_0$ and $[\![x]\!]_1$. Therefore, the parties can define an RSS sharing $\langle x \rangle$ from $[\![x]\!] = \{[\![x]\!]_0, [\![x]\!]_1\}$ *non-interactively* as:

$$\langle x \rangle_i = (0, [\![x]\!]_0), \ \langle x \rangle_{i+1} = ([\![x]\!]_1, 0), \ \langle x \rangle_{i+2} = ([\![x]\!]_0, [\![x]\!]_1).$$

Using this trick, the parties can define a *consistent* RSS-sharing $\langle rdx \rangle$ from $[\![rdx]\!]$. Note $\langle idx \rangle$ is a consistent RSS sharing and $\langle \Delta \rangle = \langle rdx \rangle \oplus \langle idx \rangle$, then $\langle \Delta \rangle$ is also consistent (see Def. 1 and its following explaination). Since Δ is a consistent RSS sharing, the parties can correctly reconstruct Δ to P_i and P_{i+1} using $\mathcal{F}_{\text{recon}}$. Note that Δ is only reconstructed to P_i and P_{i+1}, P_{i+2} cannot learn it; otherwise P_{i+2} can learn $idx = \Delta \oplus rdx$.

After applying the above consistency checks, a corrupted party is only limited to adding errors in the resharing phase.
DPF-based OS Protocol. The protocol is formally described in Fig. 4. We note that $\Pi_{\text{dpf-os}}$ is subject to an additive attack in the resharing phase; we will show how to handle additive errors at a later stage. Theorem 2 shows the security of $\Pi_{\text{dpf-os}}$, with the proof from Appendix C.2.

THEOREM 2. *$\Pi_{\text{dpf-os}}$ securely computes $\mathcal{F}_{\text{os}}$ in the $\mathcal{F}_{\text{recon}}$-hybrid model in the presence of a malicious adversary in the three-party honest-majority setting, assuming F is a secure PRF.*

5.3 The Mostree PDTE Protocol

Detecting Errors using MACs. Mostree uses $\mathcal{F}_{\text{os}}$ for oblivious node selection, but $\mathcal{F}_{\text{os}}$ is subject to additive errors. We use SPDZ-like MACs to detect errors.

SPDZ-like MACs. An SPDZ-like MAC $\sigma(\alpha, x)$ is usually defined as $\sigma(\alpha, x) = \alpha \cdot x$ over a finite field $\mathbb{F}$, where x is the value to be authenticated and α is the MAC key. These MACs are additively homomorphic: $\alpha \cdot (x + y) = \alpha \cdot x + \alpha \cdot y$. In the following, we will drop α and instead use $\sigma(x)$ or σ_x to denote the MAC for x when the context is clear. A MAC $\sigma(x)$ is also shared along with its authenticated secret x. Secure computation is performed both for x and $\sigma(x)$, and the protocol aborts if the output x and the MAC tag σ_x not satisfying $\alpha \cdot x = \sigma_x$. In order to achieve overwhelming detection probability, the field $\mathbb{F}$ must be large enough, which is vital to ensure overwhelming detection probability.

MACs over $\mathbb{F}_{2^\ell}$ with low overhead. We use MACs over $\mathbb{F}_{2^\ell}$ to authenticate ℓ-bit secrets as a whole. We exploit the fact that $\mathbb{Z}_2^\ell$ is compatible with $\mathbb{F}_{2^\ell}$ over addition (*i.e.*, bit-wise XOR). Therefore, a secret shared over $\mathbb{Z}_2^\ell$ can be converted to a secret over $\mathbb{F}_{2^\ell}$ for free. We note that $\mathbb{F}_{2^\ell}$ is incompatible with bit-wise multiplication. Nevertheless, these MACs are only used for detecting errors from oblivious selection rather than the whole computation. Another benefit is that $\ell > \lambda$ for real-world decision-tree applications where λ is a statistical security parameter. Thus $\mathbb{F}_{2^\ell}$ is large enough for error detection. This means the MACs are at the same size as the authenticated secrets, incurring only constant overhead. Notably, our method does not require any expensive share conversion.

Parameters: Three parties denoted as P_0, P_1 and P_2; statistical security parameter λ; a finite field $\mathbb{F}_{2^\ell}$ where $\ell \gg \lambda$; number $m \in \mathbb{Z}$ denotes the number of RSS sharings to be checked.

[Check] On inputting m RSS sharings $\{x_j\}_{j \in [m]}$, MAC values $\{\sigma(x_j)\}_{j \in [m]}$ and MAC key sharing $\langle \alpha \rangle$, outputs True if MAC check passed and False otherwise.

1. The parties call $\langle r \rangle \leftarrow \mathcal{F}_{\text{rand}}(\mathbb{F}_{2^\ell})$ and $\langle \sigma(r) \rangle \leftarrow \mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}(\langle r \rangle, \langle \alpha \rangle)$.
2. The parties call $\mathcal{F}_{\text{coin}}(\mathbb{F}_{2^\ell}^m)$ to receive random elements $\rho_1, \rho_2, \cdots, \rho_m \in \mathbb{F}_{2^\ell} \setminus \{0\}$.
3. The parties locally compute $\langle v \rangle \leftarrow \langle r \rangle + \sum_{j \in [m]} \rho_j \cdot \langle x_j \rangle$ and $\langle w \rangle \leftarrow \langle \sigma(r) \rangle + \sum_{j \in [m]} \rho_j \cdot \langle \sigma(x_j) \rangle$.
4. Securely open $\langle v \rangle$ via $\mathcal{F}_{\text{open}}$. Abort if the open fails.
5. Call $\mathcal{F}_{\text{CheckZero}}(\langle w \rangle - v \cdot \langle \alpha \rangle)$ and abort if receives False.

Figure 5: Protocol for Batch MAC Check Π_{MacCheck}

Mostree authenticates the shared tree array $\langle \vec{T} \rangle$ by computing their MACs (also shared) in the setup phase. In the online phase, the parties run $\mathcal{F}_{\text{os}}$ over both the shared tree array and its MAC array. The intuition is that any introduced errors will break the relationship between the shared data and its MAC, which can be detected by a MAC check protocol Π_{MacCheck} in Fig. 5. Such a batch MAC check technique is previously used in [13, 20, 29], which detects additive errors with probability $1 - O(1/|\mathbb{F}|)$, which is closed to 1 except with negligible probability, for large enough field $\mathbb{F}$.
The Mostree Protocol. Combining tree encoding, $\mathcal{F}_{\text{os}}$, MAC over $\mathbb{F}_{2^\ell}$ and RSS-based secure computation, we propose Mostree formally described in Fig. 6. Mostree intends to compute a PDTE functionality $\mathcal{F}_{\text{pdte}}$: On inputting a feature vector $\vec{X}$ from MO and a decision tree $\vec{T}$ from MO, outputs a classification result to FO. Mostree contains a one-time offline setup protocol and an online evaluation protocol.

Parameters: Three parties denoted as P_0, P_1 and P_2; statistical security parameter λ; $\mathbb{F}_2$ denotes the binary field for boolean sharing; m denotes the number of tree nodes; ℓ_m denotes the minimal value such that $m \leq 2^{\ell_m}$; n denotes the dimension of feature vectors; k denotes the bit-length of values (*e.g.*, left and right children index, threshold value, and classification result); ℓ denotes the bit-length of tree nodes, *i.e.*, $\ell = 4k + n$; a finite field $\mathbb{F}_{2^\ell}$ where $\ell \gg \lambda$.

[Setup] Upon receiving a DT $\vec{\mathcal{T}}$, the setup protocol outputs $(\langle \vec{T} \rangle, \langle \vec{M} \rangle)$ where $\langle \vec{T} \rangle$ is the RSS-sharing array for the tree and $\langle \vec{M} \rangle$ is the MAC array for $\langle \vec{T} \rangle$ such that $\vec{M} = \sigma(\vec{T})$.

1. MO encodes $\vec{\mathcal{T}}$ as an array $\vec{T}$. The parties call $\mathcal{F}_{\text{share}}$ in which MO inputs $\vec{T}$. As a result, $\langle \vec{T} \rangle$ is shared between the parties.
2. The parties call $\langle \alpha \rangle \leftarrow \mathcal{F}_{\text{rand}}(\mathbb{F}_{2^\ell})$ to share a MAC key $\alpha \in \mathbb{F}_{2^\ell}$.
3. Compute MACs: $\langle \vec{M}[j] \rangle \leftarrow \mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}(\langle \alpha \rangle, \langle \vec{T}[j] \rangle)$ for $j \in [m]$.
4. The parties run $\Pi_{\text{MacCheck}}(\langle \vec{T} \rangle, \langle \vec{M} \rangle)$. If the check fails, abort.
5. Output $(\langle \vec{T} \rangle, \langle \vec{M} \rangle)$.

[Evaluation] Upon receiving $\vec{X}$ from FO, do:

1. FO shares $\langle \vec{X} \rangle$ via calling $\mathcal{F}_{\text{share}}$.
2. Initialize $\langle result \rangle \leftarrow \perp$.
3. Parse $\langle t \rangle || \langle l \rangle || \langle r \rangle || \langle \vec{v} \rangle || \langle c \rangle \leftarrow \langle \vec{T}[0] \rangle$.
4. For $j \in [d_{\text{pad}}]$: // step 1) to 3) are executed by calling $\mathcal{F}_{\text{3pc}}^{\mathbb{F}_2}$.
   1) $\langle x \rangle \leftarrow \sum_{j \in [n]} \langle \vec{X}[j] \rangle \cdot \langle \vec{v}[j] \rangle$.
   2) $\langle b \rangle \leftarrow \langle x \rangle < \langle t \rangle$.
   3) $\langle idx \rangle \leftarrow \langle r \rangle \oplus \langle b \rangle \cdot (\langle l \rangle \oplus \langle r \rangle)$.
   4) $\langle \vec{T}[idx] \rangle || \langle \vec{M}[idx] \rangle \leftarrow \mathcal{F}_{\text{os}}(\langle idx \rangle, \langle \vec{T} \rangle || \langle \vec{M} \rangle)$.
   5) Parse $\langle t \rangle || \langle l \rangle || \langle r \rangle || \langle \vec{v} \rangle || \langle c \rangle \leftarrow \langle \vec{T}[idx] \rangle$.
   6) Update $\langle result \rangle \leftarrow \langle c \rangle$.
5. Use Π_{MacCheck} to check d_{pad} pairs of RSS sharings from $\mathcal{F}_{\text{os}}$, abort if the check fails.
6. Call $\mathcal{F}_{\text{recon}}(\langle result \rangle)$ to reconstruct $result$ to FO.

Figure 6: The Mostree Protocol Π_{pdte}

Setup phase. The setup protocol requires linear communication in the tree size. Setup is only run once thus the overhead can be amortized across subsequent queries.

As a requirement of $\mathcal{F}_{\text{os}}$, we need the dimension of $\vec{T}$ to be a power of two (*i.e.*, $m = 2^{\ell_m}$ for some ℓ_m). If this is not the case, the tree holder can pad the array before sharing the encoded tree. In particular, the holder computes the minimal ℓ_m such that $m \leq 2^{\ell_m}$ and allocates an array of size 2^{ℓ_m}. The holder randomly assigns m nodes within the power-of-two-length array and modifies each node's left and right child correspondingly. Overall, this padding, at most, doubles the storage overhead of the non-padded version.

Mostree relies on existing ideal functionality $\mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}$ for MAC generation. Concretely, the parties first invoke $\mathcal{F}_{\text{rand}}$ to obtain an RSS-sharing $\langle \alpha \rangle$ for a random MAC key $\alpha \xleftarrow{\$} \mathbb{F}_{2^\ell}$ (Step 2, Setup). Then they call $\mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}$ m times to compute an array sharings $\langle \vec{M} \rangle$ for MACs of $\langle \vec{T} \rangle$, where $\vec{M}[j] = \alpha \cdot \vec{T}[j]$ is defined over $\mathbb{F}_{2^\ell}$ for $j \in [m]$ (Step 3, Setup). Note that $\mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}$ also suffers from additive attacks. To check whether these MACs are correctly generated, the parties can run batch MAC check over $(\langle \vec{T} \rangle, \langle \vec{M} \rangle)$ at the end of the setup protocol (Step 4, Setup); this means we can use Π_{MacCheck} to uniformly handle leakage from the setup protocol and $\mathcal{F}_{\text{os}}$.

Evaluation phase. The evaluation protocol, on inputting $\langle \vec{T} \rangle$, $\langle \vec{M} \rangle$ and an RSS-sharing feature vector $\langle \vec{X} \rangle$, outputs an RSS-sharing $\langle result \rangle$, where $result$ denotes the classification result and will be reconstructed to FO. We stress that feature selection must also be done obliviously. Certainly, we can resort to $\mathcal{F}_{\text{os}}$ again, but it turns out that this approach is overkill in most cases because feature vectors are usually with low-dimension. We instead use a much simpler approach. Specifically, we modify the tree encoding method by replacing each feature index $v \in \mathbb{Z}_n$ to a length-n unit bit vector $\vec{v} \in \mathbb{Z}_2^n$, where $\vec{v}$ comprises all 0s except for a single 1 appearing at coordinate v. In this manner, the parties simply run RSS-based inner-product computation for oblivious feature selection. Here the triple-verification method from [18] is used to ensure the correctness of each multiplication, incurring $O(n)$ communication.

The evaluation protocol contains $d_{\text{pad}} > d$ iterations. In each iteration, the parties obliviously select the desired feature value, make a secure comparison, and decide on the next node for evaluation. All non-RAM computation (*e.g.*, secure comparison) is performed in a secret-shared fashion using 3PC ideal functionality $\mathcal{F}_{\text{3pc}}^{\mathbb{F}_2}$, in which privacy and correctness are guaranteed in the presence of a malicious adversary. After that, the parties call $\mathcal{F}_{\text{os}}(\langle \vec{T} \rangle || \langle \vec{M} \rangle, \langle idx \rangle)$; here we abuse the notation by calling $\mathcal{F}_{\text{os}}$ over $\langle \vec{T} \rangle$ and $\langle \vec{M} \rangle$ simoustiously. Note that $\mathcal{F}_{\text{os}}$ is subject to additive attacks, the parties must run MAC check to detect any error for each selection. Mostree delays all these MAC checks in a batch, before reconstructing $\langle result \rangle$ to the FO.

Complexity. The online communication complexity of Π_{pdte} is sublinear in m because either $\Pi_{\text{rss-os}}$ or $\Pi_{\text{dpf-os}}$ only requires constant online communication, while the computation complexity is linear since each party needs to perform a linear scan over its local share. The linear scan is extremely cheap (*i.e.*, bit-wise computation). Thus, existing hardware acceleration can be applied to optimize its performance. As for offline communication, the setup protocol Π_{setup} requires linear communication but is only invoked once in the initialization phase, which means all initialized RSS sharings can be reused to perform subsequent evaluation queries, among which the linear communication can be amortized. For OS setup protocols, $\Pi_{\text{rss-os}}$ still requires linear communication while $\Pi_{\text{dpf-os}}$ enjoys sublinear offline communication.

Security. We design Mostree in a modular manner, making it easier to analyze security in the hybrid model. Formally, we show the security of Π_{pdte} in Theorem 3 and prove it in Appendix C.3.

THEOREM 3. *Protocol Π_{pdte} securely computes $\mathcal{F}_{\text{pdte}}$ in the $(\mathcal{F}_{\text{3pc}}^{\mathbb{F}_2}$, $\mathcal{F}_{\text{os}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{open}} \mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}, \mathcal{F}_{\text{CheckZero}})$-hybrid model for $\ell > \lambda$ in the present of the malicious adversary in the 3PC honest-majority setting.*

6 EXPERIMENT

This section reports the implementation and performance of Mostree.

6.1 Implementation and Experiment Details

We evaluate the performance of Mostree using ABY³ [34] in C++. Our implementation is available at https://github.com/Jbai795/Mostree-

Table 2: Parameters of Datasets

Dataset	Depth d	Features n	#(Nodes) m	#(Padded nodes) m'
wine	5	7	23	32
breast	7	12	43	64
digits	15	47	337	512
spambase	17	57	171	256
diabetes	28	10	787	1024
Boston	30	13	851	1024
MNIST	20	784	4179	8192

pub. We test on 7 representative datasets from the UCI repository (https://archive.ics.uci.edu/ml) as listed in Table 2. The trees we trained vary in depth and size. Among them, *wine, breast* are small example trees while *Boston* is a deep-but-sparse tree and *MNIST* is a density tree with a high-dimensional feature vector.

We run benchmarks on a desktop PC equipped with Intel(R) Core i9-9900 CPU at 3.10 GHz × 16 running Ubuntu 20.04 LTS with 32 GB memory. We use Linux tc tool to simulate local-area network (LAN, RTT: 0.1 ms, 1 Gbps), metropolitan-area network (MAN, RTT: 6 ms, 100 Mbps) and wide-area network (WAN, RTT: 80 ms, 40 Mbps). We set the computational security parameter $\kappa = 128$, which determines the key length of a pseudorandom function, and statistical security parameter $\lambda = 40$, with element size $k = 64$. Note we do not provide accuracy evaluation for Mostree as DTE only involves comparison, and there is no accuracy loss if the comparison is computed bit-wise in secure multiparty computation.

6.2 Comparison with Three-party Works

Since Mostree is the first three-party PDTE that considers honest majority security settings, we compare it with two latest three-party PDTE schemes [14, 23] with different security settings. The former [14] works in a dishonest majority setting whose security is a bit stronger than Mostree and the latter [23] works in a semi-honest setting. We re-run protocols in [14] and [23], which we name as SPDZ-tree and UCDT, respectively. We can build two kinds of Mostree based on different OS protocols. However, the online phases of them are identical. To this, we give two lines (RSS-tree and Mostree) in Fig. 7(a) for offline communication while we only give one line (Mostree) for online communication in Fig. 7(b). We additionally give the concrete performance of two OS protocols in Appendix C.4. Since the DPF and RSS-based OS protocol outperforms the pure RSS-based OS, Mostree uses the DPF and RSS-based OS protocol if there is no explicit statement in the following.

Communication Evaluation. Fig. 7 reports the communication of Mostree, SPDZ-tree and UCDT. We obtain the communication for all listed datasets under different protocols except SPDZ-tree as we fail to compile the protocol over large trees. Its linear complexity makes it run out of memory during execution.

As we can see, over online communication, Mostree significantly outperforms SPDZ-tree. Specifically, SPDZ-tree requires the most communication cost because it performs oblivious attribute selection and node comparison for each inner tree node. It is worth noticing that Mostree only shows an acceptable increase in online and offline communication overhead compared with UCDT (*e.g.*, Mostree requires ~4× online communication for MNIST). In the offline phase, it is clear that RSS-tree (employs pure RSS-based OS) requires more communication resources than Mostree.
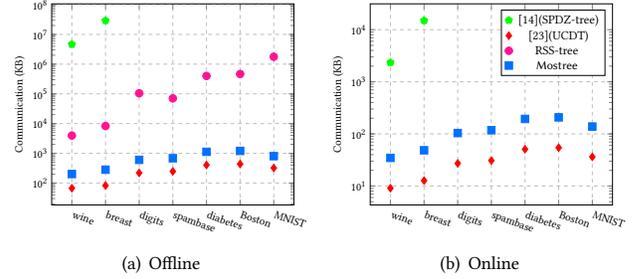

(a) Offline   (b) Online

Figure 7: Online and Offline Communication Cost. y-**axis is in the logarithm scale. RSS-tree (Mostree) means we use pure RSS-based (DPF and RSS-based) OS protocol.**

Running Time. We evaluate Mostree, SPDZ-tree and UCDT under LAN, MAN, and WAN network settings in Fig. 8. Mostree shows a lower running time in the LAN setting than SPDZ-tree since Mostree only requires to perform oblivious node selection once for each tree level. The significant difference between Mostree and SPDZ-tree continues in the MAN and WAN network settings. Compared with UCDT, Mostree reports a slightly more running time (*e.g.*, ~4×) for all listed trees in all network settings.

Table 4 illustrates the total evaluation time of Mostree for different trees. Although the total running time of Mostree is linear to the tree size, the results in Table 4 shows it is more related to the tree depth. We acknowledge the running time is very sensitive to the latency. It can be highly improved by using batching techniques to send independent data together. We leave this as our future work.

Scalability. We evaluate the scalability of Mostree in a LAN setting. Following the approach in [36], we set the tree node number $m = 25d$ and vary d from 20 to 50. Table 5 shows the evaluation result. We can see that while the tree depth increases, the communication grows slowly and linearly to the tree size, showing the scalability of Mostree. As for the running time, since the tree node number is linear to the tree depth in our setting, it also rises with the depth increase.

6.3 Comparison with Two-party Works

Most of the existing PDTE protocols focus on two-party settings. In particular, we compare Mostree with a comprehensive work [25] and two latest efficient protocols [4, 30]. In particular, both communication and computation in work [25] are linear to the tree size, while the communication in [4, 30] is sublinear. Table 3 reports the online results. Since work [25] constructs PDTE modularly either using GC or HE, we compare Mostree with its two protocols: computation-efficient GGG and communication-efficient HHH. The table shows that GGG in [25] enjoys the best running time, and the work in [30] shows the best communication for small trees like wine. While it goes to big trees like MNIST, Mostree outruns others in communication, including HHH. In particular, Mostree saves ~3× communication cost than the latest sublinear work [4]. The main reason is that work [4] requires a two-party secure PRF evaluation for each node selection, which incurs significant communication overhead. In contrast, Mostree performs node selection mainly
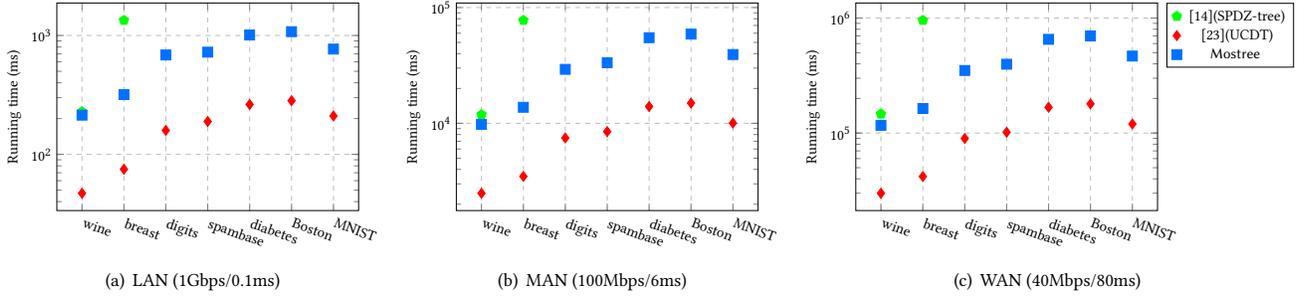
(a) LAN (1Gbps/0.1ms)

(b) MAN (100Mbps/6ms)

(c) WAN (40Mbps/80ms)

Figure 8: Online Runtime in LAN/MAN/WAN Setting. y-axis is in the logarithm scale.

Table 3: Comparison with Two-party Protocols over Different Datasets and Networks

Protocols	Security	wine			digits			MNIST		
		LAN.T	WAN.T	Comm.	LAN.T	WAN.T	Comm.	LAN.T	WAN.T	Comm.
Kiss *et al.* [25][GGG]	○	**10.1**	**257.6**	44.5	144.9	**820.7**	1499.2	2320.1	7088.4	23431.4
Kiss *et al.* [25][HHH]	○	263.7	2083.5	90.2	1306.8	23554.6	990.7	13570.2	311726.9	-
Ma *et al.* [30]	○	10.3	1300.4	**1**	**28.2**	4211.3	110.8	**35.4**	**6493.5**	270.3
Bai *et al.* [4]	○	54.8	6295.8	336.6	152.1	18755.8	109.4	203.9	24997.3	369.4
Mostree	●	213.7	116868	34.6	685.5	350294	**103.8**	768.1	467116	**138.4**

LAN.T, WAN.T, and Commu. represent online running time under LAN, online running time under WAN, and online communication, respectively. ○: semi-honest, ●: malicious. All running times are reported in milliseconds (ms), and communication in KB. The minimum value in each column is bolded.

Table 4: Total Runtime (s) of Mostree under Different Network Conditions

	wine	breast	digits	spambase	diabetes	Boston	MNIST
LAN	0.37	0.56	1.21	1.27	1.74	1.87	1.29
MAN	16.90	23.70	50.81	57.79	94.87	101.61	67.95
WAN	202.65	283.56	607.77	688.81	1133.29	1213.21	934.52

Table 5: Mostree Running on Trees with Varies Depth

Tree Depth	20	25	30	35	40	45	50
TRT (ms)	1397.9	1745.9	2046.3	2537.7	2923.1	2829.4	3270.6
TC (KB)	362.0	446.3	530.6	614.9	699.1	783.4	594.3

TRT and TC represent total running time and total communication, respectively.

from cheap local evaluation, thus achieving better performance. We do acknowledge that this is partially due to the three-party honest majority setting, where Mostree exploits to design a more efficient PDTE.

As for the running time, Mostree shows worse in some cases, but please note that Mostree achieves a much stronger security guarantee than other protocols. In some cases, GGG in [25] shows the best running time. This is because it is evaluated purely based on the Garbled Circuit with the high communication price. As the tree size grows, work [30] outperforms others, yet it leaks access patterns. In Table 3, we only compare Mostree with semi-honest two-party because the existing two-party work [35] [39] only supporting malicious FO can be seen as the malicious version of HHH in [25] (with additional ZKP and COT), with worse running time. We acknowledge that high computation in Mostree also means

high monetary costs for cloud-assisted applications. Reducing the computation overhead further is our future work.

7 CONCLUSION

This paper presents Mostree, a communication-efficient PDTE protocol. Mostree proposes two OS protocols with low communication by using RSS sharings and distributed point function. Mostree carefully combines oblivious selection protocols with data structure and lightweight consistency check techniques, achieving malicious security and sublinear communication. Our experiment results show that Mostree is efficient and practical.

Limitations and Future Works. Mostree achieves sublinear online communication but still requires (super) linear computation cost. Achieving low computation overhead is also important, which we leave as our future work. In addition, we will extend malicious OS to other privacy-preserving applications, *e.g.*, privacy-preserving inference using graph-based models.

REFERENCES

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS*. 805–817.

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS*. 535–548.

[3] Nuttapong Attrapadung, Goichiro Hanaoka, Takahiro Matsuda, Hiraku Morita, Kazuma Ohara, Jacob CN Schuldt, Tadanori Teruya, and Kazunari Tozawa. 2021. Oblivious Linear Group Actions and Applications. In *ACM CCS*. 630–650.

[4] Jianli Bai, Xiangfu Song, Shujie Cui, Ee-Chien Chang, and Giovanni Russello. 2022. Scalable Private Decision Tree Evaluation with Sublinear Communication. In *AsiaCCS*. 843–857.

[5] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Secure evaluation of private linear branching programs with medical applications. In *ESORICS*. Springer, 424–439.

[6] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine learning classification over encrypted data. In *NDSS*, Vol. 4324. 4325.

[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function secret sharing. In *EUROCRYPTO*. Springer, 337–367.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function secret sharing: Improvements and extensions. In *AMC CCS*. 1292–1303.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure computation with preprocessing via function secret sharing. In *TCC*. Springer, 341–371.

[10] Andrej Bratko, Bogdan Filipič, Gordon V Cormack, Thomas R Lynam, and Blaž Zupan. 2006. Spam filtering using statistical data compression models. *The Journal of Machine Learning Research* 7 (2006), 2673–2698.

[11] Justin Brickell, Donald E Porter, Vitaly Shmatikov, and Emmett Witchel. 2007. Privacy-preserving remote diagnostics. In *ACM CCS*. 498–507.

[12] Jason Catlett. 1991. Overprvning Large Decision Trees.. In *IJCAI*. Citeseer, 764–769.

[13] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO*. Springer, 34–64.

[14] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE S&P*. IEEE, 1102–1120.

[15] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A private time-series database from function secret sharing. In *IEEE S&P*. IEEE, 2450–2468.

[16] Leo de Castro and Anitgoni Polychroniadou. 2022. Lightweight, Maliciously Secure Verifiable Function Secret Sharing. In *EUROCRYPT*. Springer, 150–179.

[17] Martine De Cock, Rafael Dowsley, Caleb Horst, Raj Katti, Anderson CA Nascimento, Wing-Sea Poon, and Stacey Truex. 2017. Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. *IEEE TDSC* 16, 2 (2017), 217–230.

[18] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*. Springer, 225–255.

[19] Niv Gilboa and Yuval Ishai. 2014. Distributed point functions and their applications. In *EUROCRYPT*. Springer, 640–658.

[20] Thang Hoang, Jorge Guajardo, and Attila A Yavuz. 2020. MACAO: A maliciously-secure and client-efficient active ORAM framework. *Cryptology ePrint Archive* (2020).

[21] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *CRYPTO*. Springer, 145–161.

[22] Yuval Ishai and Anat Paskin. 2007. Evaluating branching programs on encrypted data. In *TCC*. Springer, 575–594.

[23] Keyu Ji, Bingsheng Zhang, Tianpei Lu, Lichun Li, and Kui Ren. 2021. UC Secure Private Branching Program and Decision Tree Evaluation. *Cryptology ePrint Archive* (2021).

[24] Marc Joye and Fariborz Salehi. 2018. Private yet efficient decision tree evaluation. In *IFIP DBSec*. Springer, 243–259.

[25] Ágnes Kiss, Masoud Naderpour, Jian Liu, N Asokan, and Thomas Schneider. 2019. Sok: Modular and efficient private decision tree evaluation. *PoPETs* 2019, 2 (2019), 187–208.

[26] Hian Chye Koh, Wei Chin Tan, and Chwee Peng Goh. 2006. A two-step method to construct credit scoring models with data mining techniques. *International Journal of Business and Information* 1, 1 (2006), 96–118.

[27] Sotiris B Kotsiantis. 2013. Decision trees: a recent overview. *Artificial Intelligence Review* 39, 4 (2013), 261–283.

[28] Phi Hung Le, Samuel Ranellucci, and S Dov Gordon. 2019. Two-party private set intersection with an untrusted third party. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2403–2420.

[29] Yehuda Lindell and Ariel Nof. 2017. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS*. 259–276.

[30] Jack PK Ma, Raymond KH Tai, Yongjun Zhao, and Sherman SM Chow. 2021. Let's stride blindfolded in a forest: Sublinear multi-client decision trees evaluation. In *NDSS*.

[31] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *ACM CCS*. 35–52.

[32] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. 2023. PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries with Full Security. *Cryptology ePrint Archive* (2023).

[33] Vili Podgorelec, Peter Kokol, Bruno Stiglic, and Ivan Rozman. 2002. Decision trees: an overview and their use in medicine. *Journal of medical systems* 26, 5 (2002), 445–463.

[34] Peter Rindal. [n. d.]. The ABY3 Framework for Machine Learning and Database Operations. https://github.com/ladnir/aby3.

[35] Raymond KH Tai, Jack PK Ma, Yongjun Zhao, and Sherman SM Chow. 2017. Privacy-preserving decision trees evaluation via linear functions. In *ESORICS*. Springer, 494–512.

[36] Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. 2019. Private Evaluation of Decision Trees using Sublinear Cost. *PoPETs* 2019, 1 (2019), 266–286.

[37] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. 2022. Duoram: A Bandwidth-Efficient Distributed ORAM for 2-and 3-Party Computation. *Cryptology ePrint Archive* (2022).

[38] Sameer Wagh. 2022. Pika: Secure Computation using Function Secret Sharing over Rings. *Cryptology ePrint Archive* (2022).

[39] David J Wu, Tony Feng, Michael Naehrig, and Kristin E Lauter. 2016. Privately Evaluating Decision Trees and Random Forests. *PoPETs* 2016, 4 (2016), 335–355.

A SECURITY DEFINITION

DEFINITION 4 (DPF SECURITY). *A two-party DPF satisfies the following requirements:*

- Correctness: *for any point function* $f : \mathcal{D} \rightarrow \mathcal{R}$ *and every* $x \in \mathcal{D}$, *if* $(k_0^{dpf}, k_1^{dpf}) \leftarrow \mathrm{Gen}(1^\kappa, f)$ *then* $\Pr[\mathrm{Eval}(k_0^{dpf}, x) + \mathrm{Eval}(k_1^{dpf}, x) = f(x)] = 1$.

- Secrecy: *For any two point functions* f, f^*, *it holds that* $\{k_b^{dpf} : (k_0^{dpf}, k_1^{dpf}) \leftarrow \mathrm{Gen}(1^\kappa, f)\} \stackrel{c}{\equiv} \{k_b^{dpf^*} : (k_0^{dpf^*}, k_1^{dpf^*}) \leftarrow \mathrm{Gen}(1^\kappa, f^*)\}$ *for* $b \in \{0, 1\}$.

DEFINITION 5 (THREE-PARTY SECURE COMPUTATION). *Let* $\mathcal{F}$ *be a three-party functionality. A protocol* Π *securely computes* $\mathcal{F}$ *with abort in the presence of one malicious party, if for every party* P_i *corrupted by a probabilistic polynomial time (PPT) adversary* $\mathcal{A}$ *in the real world, there exists a PPT simulator* $\mathcal{S}$ *in the ideal world with* $\mathcal{F}$, *such that,*

$$\{\mathrm{IDEAL}_{\mathcal{F}, \mathcal{S}(z), i}(x_0, x_1, x_2, n)\} \stackrel{c}{\equiv} \{\mathrm{REAL}_{\Pi, \mathcal{A}(z), i}(x_0, x_1, x_2, n)\}.$$

where $x_i \in \{0, 1\}^*$ *is the input provided by* P_i *for* $i \in [3]$, *and* $z \in \{0, 1\}^*$ *is the auxiliary information that includes the public input length information* $\{|x_i|\}_{j \in [3]}$. *The protocol* Π *securely computes* $\mathcal{F}$ *with abort in the presence of one malicious party with statistical error* $2^{-\lambda}$ *if there exists a negligible function* $\mu(\cdot)$ *such that the distinguishing probability of the adversary is less than* $2^{-\lambda} + \mu(\kappa)$.

B SUBPROTOCOLS USED IN MOSTREE

This section shows protocols used in Mostree, including protocols that securely compute assumed RSS-based functionalities in section 3 and others used in Mostree, *e.g.*, secure RSS-based equality comparison.

Protocol Π_{rand} **for generating a random RSS sharing**. Fig. 9 shows protocol Π_{rand} for generating an RSS sharing $\langle r \rangle$ for a random value $r \in \mathbb{F}$. The protocol can be done non-interactively using PRF F after a one-time setup for sharing PRF keys.

Parameters: Three parties $\{P_0, P_1, P_2\}$; a field $\mathbb{F}$ over which the RSS sharing works; a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}$.
[Setup] Upon input (setup), do:
1. For $i \in [3]$, P_i samples a PRF key $\mathsf{k}_i^{\mathsf{prf}} \in \mathcal{K}$ and sends $\mathsf{k}_i^{\mathsf{prf}}$ to P_{i-1}. P_i also receives $\mathsf{k}_{i+1}^{\mathsf{prf}}$ from P_{i+1}.
[Rand] Upon input (**rand**, sid), do:
1. for $i \in [3]$, P_i defines $[\![r]\!]_i \leftarrow F(\mathsf{k}_i^{\mathsf{prf}}, \mathsf{sid})$ and $[\![r]\!]_{i+1} \leftarrow F(\mathsf{k}_{i+1}^{\mathsf{prf}}, \mathsf{sid})$. P_i defines $\langle r \rangle_i \leftarrow ([\![r]\!]_i, [\![r]\!]_{i+1})$.
2. The parties output $\langle r \rangle$.

Figure 9: Protocol Π_{rand} for Securely Compute $\mathcal{F}_{\mathsf{rand}}$

Parameters: Three parties $\{P_0, P_1, P_2\}$; a field $\mathbb{F}$ over which the RSS sharing works; a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}$.
[Open] Upon input (sid, $\langle x \rangle$), do:
1. For $i \in [3]$, P_i sends $[\![x]\!]_i$ to P_{i-1} and $[\![x]\!]_{i-1}$ P_{i+1}.
2. For $i \in [3]$, P_i receives $[\![x]\!]_{i+1}$ from P_{i+1} and P_{i+2}. If P_{i+1} and P_{i+2} send different values of $[\![x]\!]_{i+1}$, send abort to all other parties.
3. If the protocol does not abort, for $i \in [3]$, P_i computes $x = \sum_{i \in [3]} [\![x]\!]_i$ and outputs x.

Figure 10: Protocol Π_{open} for Securely Compute $\mathcal{F}_{\mathsf{open}}$

Parameters: Three parties $\{P_0, P_1, P_2\}$; a field $\mathbb{F}$ over which the RSS sharing works; a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}$.
[Coin] Upon input (sid), do:
1. The parties call $\mathcal{F}_{\mathsf{rand}}$ to generate a random RSS sharing $\langle r \rangle$.
2. The parties call $\mathcal{F}_{\mathsf{open}}$ to open $\langle r \rangle$. If the open does not abort, each party outputs r.

Figure 11: Protocol Π_{coin} for Securely Compute $\mathcal{F}_{\mathsf{coin}}$

Parameters: Three parties $\{P_0, P_1, P_2\}$; a field $\mathbb{F}$ over which the RSS sharing works; a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}$.
[Recon] Upon input (sid, $\langle x \rangle$, i), do:
1. P_i receives $[\![x]\!]_{i+1}$ from P_{i+1} and P_{i-1}. If $[\![x]\!]_{i+1}$ from P_{i+1} and P_{i-1} do not match, abort.
2. Compute $x = \sum_{j \in [3]} [\![x]\!]_j$.

Figure 12: Protocol Π_{recon} for Securely Compute $\mathcal{F}_{\mathsf{recon}}$

Parameters: Three parties $\{P_0, P_1, P_2\}$; a field $\mathbb{F}$ over which the RSS sharing works; a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}$.
[Share] Upon input (sid, x, i), do:
1. The parties call $\mathcal{F}_{\mathsf{rand}}$ to generate a random RSS sharing $\langle r \rangle$.
2. The parties call $\mathcal{F}_{\mathsf{recon}}$ over $\langle r \rangle$ and reconstruct r to P_i. P_i broadcasts $\delta = x - r$ to the other parties.
3. P_{i+1} and P_{i+2} check if they receive the same δ. If not, abort.
4. The parties output $\langle x \rangle \leftarrow \langle r \rangle + \delta$.

Figure 13: Protocol Π_{share} for Securely Compute $\mathcal{F}_{\mathsf{share}}$

Parameters: Three parties $\{P_0, P_1, P_2\}$; a field $\mathbb{F}$ over which the RSS sharing works; a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}$.
[Mul] Upon input (sid, $\langle x \rangle$, $\langle y \rangle$), do:
1. For $i \in [3]$, P_i computes $[\![t]\!]_i \leftarrow [\![x]\!]_i \cdot [\![y]\!]_i + [\![x]\!]_{i-1} \cdot [\![y]\!]_i + [\![x]\!]_i \cdot [\![y]\!]_{i-1}$. $([\![t]\!]_0, [\![t]\!]_1, [\![t]\!]_2)$ forms a $\binom{3}{3}$-sharing $[\![t]\!]$.
2. The parties call $\mathcal{F}_{\mathsf{rand}}$ to generate a random RSS sharing $\langle r \rangle$.
3. For $i \in [3]$, P_i computes $[\![z]\!]_i \leftarrow [\![t]\!]_i + [\![r]\!]_i - [\![r]\!]_{i-1}$. P_i sends $[\![z]\!]_i$ to P_{i+1} and receives $[\![z]\!]_{i-1}$ from P_{i-1}.
4. For $i \in [3]$, P_i defines $\langle z \rangle_i = ([\![z]\!]_i, [\![z]\!]_{i-1})$.

Figure 14: Protocol Π_{mut} for Securely Compute $\mathcal{F}_{\mathsf{mut}}^{\mathbb{F}}$

sharing $\langle r \rangle$ between the parties. Then the parties securely reconstruct r to the party P_i. P_i broadcasts $\delta = x - r$ to the other two parties. The other two parties cross-check whether they receive the same δ. If yes, the parties jointly compute $\langle x \rangle \leftarrow \langle r \rangle + \delta$.

Protocol Π_{mut} for multiplication. Fig. 14 shows how to perform multiplication with up to an additive attack. This protocol is essentially a semi-honest multiplication protocol. We note a malicious party can add an error e in the resharing phase to produce an incorrect RSS sharing $\langle x \cdot y + e \rangle$.

Protocol $\Pi_{\mathsf{CheckZero}}$. In Fig. 15, we show a protocol $\Pi_{\mathsf{CheckZero}}$. It can check whether an RSS sharing $\langle x \rangle$ is a sharing of 0. In particular, the protocol first calls $\mathcal{F}_{\mathsf{rand}}$ to generate a random RSS sharing $\langle r \rangle$. The parties call $\mathcal{F}_{\mathsf{mut}}^{\mathbb{F}}$ to compute $\langle w \rangle \leftarrow \langle x \rangle \cdot \langle r \rangle$. The parties then open $\langle w \rangle$ via $\mathcal{F}_{\mathsf{open}}$ and check if $w = 0$ and abort if not the case.

Protocol Π_{eq} for secure equality test. In Fig. 16, we use an equality test protocol Π_{eq} for securely comparing whether two shared secrets are equal or not, and share the equality test result between the parties. To achieve malicious security, the parties need to check

Protocol Π_{open} for securely opening an RSS sharing. We show Π_{open} in Fig. 10 for securely opening an RSS sharing. The intuition is that each party P_i will receive the share $[\![x]\!]_{i+1}$ from both P_{i-1} and P_{i+1}, which allows P_i to do cross-check and abort if the values are unequal.

Protocol Π_{coin} for securely generating random coins. Protocol Π_{coin} in Fig. 11 outputs a random value $r \in \mathbb{F}$ to all the parties. The idea is that the parties first call $\mathcal{F}_{\mathsf{rand}}$ to generate a random $\langle r \rangle$ and then open $\langle r \rangle$ using $\mathcal{F}_{\mathsf{open}}$.

Protocol Π_{recon} for secure secret reconstruction. Protocol Π_{recon} in Fig. 12 reconstructs a consistent RSS sharing $\langle x \rangle$ to P_i. To check whether the reconstruction is correct or not, P_i receives $\langle x \rangle_{i+1}$ from both P_{i+1} and P_{i-1}, and abort if the two shares are unequal.

Protocol Π_{share} for sharing a secret from P_i. Fig. 13 shows the protocol Π_{share}. It shares a secret x from P_i among three parties. In particular, Π_{share} first invokes $\mathcal{F}_{\mathsf{rand}}$ to generate a random RSS

Parameters: Three parties $\{P_0, P_1, P_2\}$; a field $\mathbb{F}$ over which the RSS sharing works; a PRF $F : \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{R}$ where $\mathcal{K} = \mathcal{D} = \{0,1\}^\kappa$ and $\mathcal{R} = \mathbb{F}$.

[CheckZero] Upon input (sid, $\langle x \rangle$), do:

1. The parties call $\mathcal{F}_{\mathsf{rand}}$ to generate a random RSS sharing $\langle r \rangle$.
2. The parties compute $\langle w \rangle \leftarrow \langle x \rangle \cdot \langle r \rangle$ by calling $\mathcal{F}_{\mathsf{mul}}^{mul}$.
3. The parties call $\mathcal{F}_{\mathsf{open}}$ to open $\langle w \rangle$. If the open aborts or $w \neq 0$, return Flase; otherwise return True.

Figure 15: Protocol $\Pi_{\mathsf{CheckZero}}$ for Securely Compute $\mathcal{F}_{\mathsf{CheckZero}}$

Input: An RSS-sharing index $\langle idx \rangle$ for $idx \in [n]$ and a public value j.
Output: An RSS-sharing $\langle u[j] \rangle$ where $u[j]$ is 1 if $idx == j$, otherwise $u[j]$ is 0.

1. The parties naturally share the public value j as $\langle j \rangle = (0, 0, j)$ and parse $\langle idx \rangle = \langle idx[l-1] \rangle \cdots \langle idx[0] \rangle$ and $\langle j \rangle = \langle j[l-1] \rangle \cdots \langle j[0] \rangle$.
2. The parties perform $\langle h[q] \rangle \leftarrow \langle idx[q] \rangle \oplus \langle j[q] \rangle$ for $q \in [l]$.
3. Set $\langle u[j] \rangle \leftarrow \langle h[0] \rangle$.
4. For $q \in \{1, \cdots, l-1\}$,
  1) The parties perform $\langle u[j] \rangle \leftarrow \langle u[j] \rangle \cdot \langle h[q] \rangle$.

Figure 16: Equality Test Π_{eq}

---

the correctness of multiplications. This can be done by the triple-based multiplication verification trick, where the triples used for verification are generated by a cut-and-choose method from [18].

C SECURITY PROOF

C.1 Proof of Theorem 1

PROOF. For any PPT adversary $\mathcal{A}$, we construct a PPT simulator $\mathcal{S}$ that can simulate the adversary's view with accessing the functionalities $\mathcal{F}_{\mathsf{rand}}$, $\mathcal{F}_{3pc}^{\mathbb{F}_2}$ and $\mathcal{F}_{\mathsf{open}}$. In the cases where $\mathcal{S}$ aborts or terminates the simulation, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Simulating preprocess phase. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{rand}}$ and uses the simulator of $\mathcal{F}_{\mathsf{rand}}$ for simultation. For each $j \in [m]$, $\mathcal{S}$ simulates $\mathcal{F}_{3pc}^{\mathbb{F}_2}$ in Π_{eq} and receives an error from the adversary. $\mathcal{S}$ aborts if $\mathcal{A}$ sends any non-zero error.

Simulating selection phase. We assume $\langle rdx \rangle$ and $\langle \vec{v} \rangle$ are correctly shared/simulated in the first stage. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{open}}$ and uses the simulator of $\mathcal{F}_{\mathsf{open}}$ for simulation and receives an error from the adversary. $\mathcal{S}$ aborts if $\mathcal{A}$ sends any non-zero error. For the local shifting and local multiplication computation, $\mathcal{S}$ is easy to simulate. Towards the re-sharing phase, the simulation randomly samples P_{i+2}'s share and sends it to $\mathcal{A}$. In the real protocol execution, this is generated by a PRF F. Therefore, the simulation is computationally indistinguishable from real protocol execution due to the security of PRF. Overall, the simulation is indistinguishable from real-world execution. □

C.2 Proof of Theorem 2

PROOF. For any PPT adversary $\mathcal{A}$, we construct a PPT simulator $\mathcal{S}$ that can simulate the adversary's view with accessing the functionality $\mathcal{F}_{\mathsf{os}}$. In the cases where $\mathcal{S}$ aborts or terminates the simulation, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Simulating preprocess phase. When the corrupted party P_i plays the role of DPF key generator, $\mathcal{S}$ plays the role of honest parties P_{i+1} and P_{i+2}, and receives a pair of DPF keys from $\mathcal{A}$. $\mathcal{S}$ can check whether the keys are correct and abort if not. When an honest party takes the turn for DPF key generation, $\mathcal{S}$ generates a pair of DPF keys and samples a random share $r_i \xleftarrow{\$} \mathbb{Z}_m$. $\mathcal{S}$ sends the key $(k_i^{\mathsf{dpf}}, r_i)$ to $\mathcal{A}$. $\mathcal{S}$ runs VDPF verification protocol with $\mathcal{A}$, and aborts if $\mathcal{A}$ aborts in the verification protocol. For all other messages sent from honest parties to the corrupted one, the simulator samples them randomly.

The above simulation is indistinguishable from real-world execution. First, if $\mathcal{A}$ sends incorrect DPF keys, then the real-world execution would have aborted due to VDPF check. In the ideal world, $\mathcal{S}$ can check keys directly, thus the ideal world aborts with indistinguishable probability. When the corrupted party plays the role of DPF evaluator, the DPF keys are honestly generated by $\mathcal{S}$. If $\mathcal{A}$ follows the VDPF key verification protocol, the check will always pass, otherwise the check protocol will abort. Therefore, $\mathcal{S}$ just runs the check protocol with $\mathcal{A}$ as in the real-world protocol, and it can always abort with indistinguishable probability.

Simulating selection phase. If the protocol does not abort after the preprocessing stage, then all DPF keys generated by $\mathcal{A}$ in $\mathcal{S}$ are correct. The only issue in the selection phase is whether $\mathcal{A}$ follows the protocol honestly. Also, $\mathcal{S}$ has to successfully extract the error term in the simulation. We assume $\langle \vec{T} \rangle$ and $\langle idx \rangle$ are already correctly shared/simulated in the first place, which means the honest parties hold correct and consistent RSS shares.

For all local computations, $\mathcal{S}$ is easy to simulate. $\mathcal{S}$ only needs to simulate the view between honest parties and the corrupted party and to abort with indistinguishable probability. Note that only a few parts in the selection phase require interaction. The first part is on securely reconstructing Δ. Since $\langle \Delta \rangle = \langle rdx \rangle \oplus \langle idx \rangle$ and both $\langle rdx \rangle$ and $\langle idx \rangle$ are correct RSS sharings, $\langle \Delta \rangle$ is also a consistent sharing by definition. This means that $\mathcal{S}$ can cross-check whether $\mathcal{A}$ sends the correct value using the share of honest party P_{i+2}: 1) if $\mathcal{A}$ sends incorrect share, $\mathcal{S}$ just aborts; 2) if $\mathcal{A}$ sends the correct share, $\mathcal{S}$ samples a random $\Delta \in \mathbb{Z}_m$ and computes the shares of P_{i+1} and P_{i+2} according to Δ and the RSS shares of P_i, and sends to $\mathcal{A}$. In this case, the parties open the random Δ to $\mathcal{A}$. Another part is from re-sharing the selection result from $\vec{A}[idx]$ from $\binom{3}{3}$-sharing back to RSS sharing. Here $\mathcal{A}$ can add an error and $\mathcal{S}$ extracts the error as follows: $\mathcal{S}$ receives the share z_i^* from the corrupted party to an honest party and updates P_{i+1}'s local share correspondingly. $\mathcal{S}$ also locally computes value z_i, which is the correct value that the corrupted party should send ($\mathcal{S}$ can do this since he knows all necessary shares to compute z_i). Then, $\mathcal{S}$ computes $d \leftarrow z_i - z_i^*$ to $\mathcal{F}_{\mathsf{os}}$. $\mathcal{S}$ sends d to $\mathcal{F}_{\mathsf{os}}$, completing simulation.

Note that in simulating the open step of Δ, $\mathcal{S}$ only uses shares of honest parties. These shares are either randomly simulated or computed from available local data. Also, they are consistent with the shares/data held by the corrupted party. Therefore, the simulation is perfectly indistinguishable from real protocol execution. Towards re-sharing phase, the simulation randomly samples P_{i+2}'s
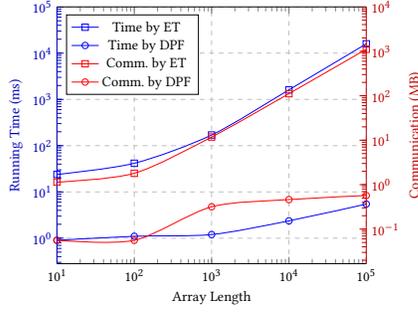
Figure 17: Unit Vector Generation. y-axis is in the logarithm scale. ET and Comm. represent equality test and communication cost, respectively.

share and sends it to $\mathcal{A}$. In the real protocol execution, this is generated by a PRF F. Therefore, the simulation is computationally indistinguishable from real protocol execution due to the security of PRF. Overall, the simulation is indistinguishable from real-world execution. $\square$

C.3 Proof of Theorem 3

PROOF. For any PPT adversary $\mathcal{A}$ who corrupts party P_i, we construct a PPT simulator $\mathcal{S}$ who can simulate the adversary's view with accessing the functionality $\mathcal{F}_{\text{os}}$. In the cases where $\mathcal{S}$ aborts or terminates the simulation, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Simulating setup. $\mathcal{S}$ samples random shares for $\vec{\mathbf{T}}$ and hands the shares to $\mathcal{A}$. $\mathcal{S}$ also randomly samples shares for the honest parties in order to perform subsequent simulations. $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{rand}}$ and uses the simulator of $\mathcal{F}_{\text{rand}}$ for simulation. For $j \in [m]$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}$ and receives d_j from the adversary. From d_j, $\mathcal{S}$ adds the error into the share of MAC value $\langle \sigma(\vec{\mathbf{T}}[j]) \rangle$ for honest party P_{i+1}. $\mathcal{S}$ simulates $\mathcal{F}_{\text{rand}}$ and $\mathcal{F}_{\text{CheckZero}}$ in Π_{MacCheck}. If there exists any $j \in [m]$ such that $d_j \neq 0$ or the simulation aborts from $\mathcal{F}_{\text{rand}}$ and $\mathcal{F}_{\text{CheckZero}}$ in Π_{MacCheck} ($\mathcal{S}$ uses existing simulation strategy for $\mathcal{F}_{\text{rand}}$ and $\mathcal{F}_{\text{CheckZero}}$), $\mathcal{S}$ aborts.

The above simulation is indistinguishable from real protocol execution. Since the protocol is designed in a hybrid model, existing simulation strategy for $\mathcal{F}_{\text{rand}}$, $\mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}$, and $\mathcal{F}_{\text{CheckZero}}$ are available, thus $\mathcal{S}$ uses them directly. $\mathcal{S}$ can also extract the strategy of $\mathcal{A}$ by receiving the error term in $\mathcal{F}_{\text{mul}}^{\mathbb{F}_{2^\ell}}$ thus $\mathcal{S}$ can abort correspondingly. Note that the MAC we use is over $\mathbb{F}_{2^\ell}$. In real protocol execution, any additive error will incur MAC check fail except with probability $\frac{1}{2^\ell - 1}$. Overall, the above simulation is statistically indistinguishable from the real-world execution, with statistical error $\frac{1}{2^\ell - 1}$.

Simulating evaluation. For all bit-wise secure computation (including inner product, comparison and MUX), $\mathcal{S}$ directly uses existing simulation strategy for $\mathcal{F}_{\text{3pc}}^{\mathbb{F}_2}$ directly. $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{os}}$ for each oblivious selection and receives an error term from $\mathcal{A}$. If $\mathcal{S}$ receives any non-zero error in simulating $\mathcal{F}_{\text{os}}$, $\mathcal{S}$ aborts at the end of the protocol. The simulation for bit-wise computation using $\mathcal{F}_{\text{3pc}}^{\mathbb{F}_2}$ is well-studied thus $\mathcal{S}$ can directly use existing simulation strategy (see [18]). This part is indistinguishable from real-world execution. The only issue is from $\mathcal{F}_{\text{os}}$. Since $\mathcal{S}$ receives the error terms from $\mathcal{A}$ per selection, $\mathcal{S}$ just aborts if $\mathcal{A}$ sends any non-zero error. Overall, the above simulation is statistically indistinguishable from real-protocol execution, with statistical error $\frac{1}{2^\ell - 1}$. $\square$

C.4 Evaluation of Two OS Protocols

We show the performance of unit vector generation of two proposed OS protocols in Fig. 17. The RSS-based OS employs equality test to generate the unit vector, which requires three parties to jointly compare each index with a given value, resulting in linear computation and communication. The DPF-based OS needs to share the keys. Due to the sublinear property of the DPF scheme, the required communication is sublinear to the array length. However, to expand the keys to the unit vector, each party needs to perform local computation for each element, which also requires linear computation. Note both of these two unit vector generations can be moved offline.