



Secure Parallel Computation on Privately Partitioned Data and Applications

Nuttapong Attrapadung
AIST, Japan

n.attrapadung@aist.go.jp

Jacob C. N. Schuldt
AIST, Japan
jacob.schuldt@aist.go.jp

Hiraku Morita
University of St. Gallen, Switzerland
hiraku.morita@unisg.ch

Tadanori Teruya
AIST, Japan
tadanori.teruya@aist.go.jp

Kazuma Ohara
AIST, Japan
ohara.kazuma@aist.go.jp

Kazunari Tozawa
The University of Tokyo, Japan
tozawa.kazunari@mail.u-tokyo.ac.jp

ABSTRACT

Parallel computation is an important aspect of multi-party computation, not only in terms of improving efficiency, but also in terms of providing privacy for computation involving conditional branching based on private data. While applying multi-party computation in parallel over several sets of input data is straightforward if the partitioning of the input data into sets is publicly known, the problem becomes much more challenging when this partitioning is private. This setting is relevant to broad class of secure computations, in particular to secure graph and database analysis in which the underlying data (graph or database) is private. In this paper, we consider a general class of functions which can be expressed via the iterative evaluation of a binary associative operation, and propose efficient protocols for evaluating such functions in parallel over privately partitioned input data. Our protocols are optimal in terms of the required number of evaluations of the underlying binary operation (i.e. $N - 1$ evaluations for total input size N), while simultaneously achieving a round complexity which is only logarithmic in the total size of the input data (i.e. $O(\log N)$).

Applying our protocols to specific functions result in concrete improvements compared to dedicated protocols from previous works. For example, we improve upon the previously best known protocols for simple functionalities such as (grouped) summation and (grouped) max, as well as the secure graph analysis protocols by Nayak *et al.* (S&P'15), which all requires $O(N \log N)$ evaluations of their respective underlying operations to achieve a $O(\log N)$ round complexity. While our protocols achieve the same asymptotic performance as the shortest path algorithms by Anagreh *et al.* (Cryptography'21), we achieve better concrete performance. Lastly, considering shortest path computations on a weighted graph via the Bellman-Ford algorithm, we reduce the communication complexity by 2.4 ~ 5.4 compared to the recent results by Araki *et al.* (CCS'21) on large-scale graphs of thousand nodes and edges. Besides this, we achieve efficient protocols for functions not considered previously, such as ArgMax, first/last projections, and list concatenation.

CCS CONCEPTS

• Theory of computation → Cryptographic protocols.

KEYWORDS

secure multi-party computation; private parallel computation

ACM Reference Format:

Nuttapong Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, Tadanori Teruya, and Kazunari Tozawa. 2022. Secure Parallel Computation on Privately Partitioned Data and Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, November 7–11, 2022, Los Angeles, CA, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3548606.3560695>

1 INTRODUCTION

Secure multi-party computation (MPC) allows mutually distrusting parties to jointly evaluate a function on their private inputs without revealing anything beyond the output of the function. MPC has come a long way since the classic general feasibility results in the eighties [9, 14, 27], and recent results constructs efficient MPC for highly complex tasks such as privacy-preserving machine learning (e.g., [22]).

In this paper, we focus on *parallelism* in MPC. Parallel computation generally improves overall computation time by simultaneously running (possibly the same) computation over multiple data sets. In the context of secure MPC, parallelization is moreover crucial when considering computation involving conditional branching based on private input, as all possible computation branches essentially needs to be executed to hide the branch taken, such as in secure decision tree learning (e.g., [13]).

Parallel computation requires the input data to be partitioned in a way that allows a given computation to be executed on the correct subset of the input data. When this partitioning is public, i.e., the (private) input data can be partitioned and structured in a way that does not leak any sensitive information, parallel MPC is trivial by running MPC on each input data subset / group. However, in many useful privacy-preserving applications, the partitioning itself is *private* (i.e., it is provided in encrypted or secret-shared form). In these cases, constructing parallel execution of MPC protocols becomes much more challenging, as we have no clear boundary for the input data the computation should be executed on. We call this scenario *private parallel computation*, and illustrate this in Figure 1, where we consider a variable-length function $f : M^* \rightarrow M$ for a domain set M . We let $[x]$ denote x in encrypted or secret-shared form. Note that the $[0]$'s in the output vector are placeholder

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9450-5/22/11.

<https://doi.org/10.1145/3548606.3560695>

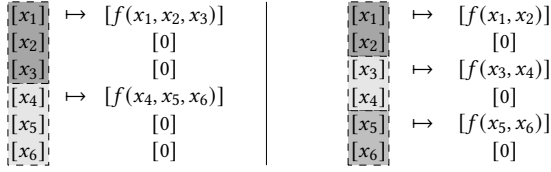


Figure 1: Example of private parallel computation, where the elements in the input vector are *privately* partitioned (illustrated in two cases)

elements that ensure the partitioning information, especially the number and the sizes of the input subsets, remain private. Figure 1 illustrates two cases of private partitioning over the input vector $[x] = ([x_1], \dots, [x_6])$, where each input data group has different colors. The private partitioning information can be given as an encrypted/secret-shared vector of boolean values that indicate the start position of each group, as suggested recently in [15]. For example, it is $[g]^B = ([1], [0], [0], [1], [0], [0])$ for the left case in Figure 1 (note that $[\cdot]^B$ refers to the boolean type).

Private parallel computations as described above was first explicitly studied by Nayak *et al.* [23], who proposed a framework for functions f that are *associative and commutative* (length-preserving) functions. This paradigm has also appeared (often as implicit sub-protocols) in many applications such as ORAM [19], secure graph analysis [5] (which improved upon [23]), and secure decision tree training [1, 15]. Each of these considers specific functions suitable for their intended application. We summarize the achieved properties in Table 1. Private parallel computation is also directly applicable to secure database queries that perform computations based on privately grouped data (*i.e.*, private analogs of plain SQL GROUP BY operations).

Our Focus and Goal. Focusing on versatility, we follow [23] and aim to obtain an improved generic protocol that can deal with a general class of functions f that can be iteratively evaluated on binary inputs to obtain an n -ary functionality. A protocol for private parallel computation is said to be generic if it can be based generically on any secure protocol for evaluating the binary input gate for f , together with some basic protocols that are *independent of f* . As the protocol for the binary-input gates for f will often dominate the cost of the overall protocol, the main design objective will be to minimize the number of calls to these gates. It is not hard to see that the lower bound is exactly that of the binary-gate circuit-size complexity of non-degenerated N -input functions, which is $N - 1$ (see *e.g.*, [26, Chapter 3]).

Our Main Contribution: Generic Protocols with the Optimal Number of OP Gates. We propose generic private parallel computation protocols for any *associative and length-preserving* functions $f : M^* \rightarrow M$ for a set M , namely, its binary operation is associative and naturally inductively defines its n -ary operation:

$$f(f(x_1, x_2), x_3) = f(x_1, f(x_2, x_3)) =: f(x_1, x_2, x_3),$$

and that the length of any element in M is fixed. More formally, (M, f) forms a *semigroup* (which is a more general algebraic structure than a group or a monoid) that is length-preserving. For a binary-input f named OP, we denote this secure functionality as

$$[z] \leftarrow \text{GroupOP}([g]^B, [x])$$

Table 1: Comparison among GroupOP Protocols

Protocol	OP	# Rounds	# OP Gates	# Basic Gates*	#Perm*
Binary gates					
Linear Scan	Generic [†]	$O(rN)$	$N - 1$	$O(N)$	-
[23]	Generic [‡]	$O(r \log N)$	$O(N \log N)$	$O(N \log N)$	-
[15] [§]	Max	$O(r \log N)$	$O(N \log N)$	$O(N \log N)$	-
[5]	Max,OR	$O(r \log N)$	$O(N \log N)$	$O(N \log N)$	-
[2]	Min	$\approx 2r \log N$	$\approx 2N$	$O(N)$	-
Ours (§5)	Generic [†]	$\approx r \log N$	$N - 1$	$O(N)$	1
Up to N -ary					
Ours (RE)*	Generic [†]	$O(r \log \log N)$	$O(N \log \log N)$	$O(N \log \log N)$	1
Ours (CE)*	Generic [†]	$O(r \log \log N)$	$N - 1$	$O(N \log \log N)$	1
Ours (RE)**	Generic [†]	r	$O(N^2)$	$O(N^2)$	1
Ours (CE)**	Generic [†]	$r + 1$	$N - 1$	$O(N^2)$	1

- * N is the input vector length; A constant r is the number of rounds of a base protocol for OP, plus 1; the upper part uses only binary gates for OP, while the below part also allow up to N -ary OP gates.
- † Basic gates are those in the ABB model *e.g.*, OR, If-then; each has $O(1)$ round&communication cost.
- ‡ Perm is for application of secure (inverse) permutation on an N -length vector, which can be done in one online round and $O(N \log N)$ communication in the semi-honest setting [6].
- § [15] also proposed GroupSum with constant rounds; we compare it separately in Table 4.
- † : Semigroups (with right identity), see §2.2. ‡ : Commutative Semigroups.
- * : for the log-log-round construction. ** : for the constant-round construction.

and call it *group-wise aggregation* for OP. The class of semigroups covers various important binary operations: addition and multiplication over the ring of integers, max/min and argmax/argmin over totally-ordered sets, first/last projections, list concatenations, set intersections/unions/differences, function compositions, monadic computations. Our class improves upon [23], which deal with a class that can be considered as the *commutative* restriction of semigroups. We do not assume commutativity.

We construct an *optimal* protocol for GroupOP for any OP in the above associative class, where the optimality refers to that the number of binary gates for OP in the protocol matches the lower bound, $N - 1$, where N is the number of elements of the input x ($N = 6$ in Fig. 1). Achieving $N - 1$ binary OP gates is rather straightforward if one allows *linear* round complexity. Contrastingly, ours has $O(\log N)$ round complexity. All previous works with logarithm-rounds [5, 15, 23] require $O(N \log N)$ OP gates, with the only exception of the recent work by [2] which achieves $O(N)$ OP gates but with a non-optimal constant. Our protocol is the first to achieve a number of OP gates that exactly matches the lower bound. We compare performance to previous work in Table 1 and provide more concrete details in Table 2.

Our protocol permits *reusing* the same $[g]^B$ for multiple $[x]$'s. In this case, GroupOP can be separated into two phases: a preparation phase, which depends only on $[g]^B$, and a execution phase, which use the result of the preparation phase to do computation on arbitrary $[x]$ according to the partitioning $[g]^B$. This situation occurs in many applications and this reusability feature enables high efficiency, *e.g.*, for secure graph analysis (see below).

More Contributions. We obtain more results as follows.

- **Generalizations and instantiations of constant-round or log-log-round GroupOP (§5.5).** We generalize our protocol when an efficient protocol for n -ary OP gates with $n > 2$ are available. This general protocol can then be instantiated in many ways resulting in GroupOP protocols with $O(1)$ rounds or $O(\log(\log(N)))$ rounds. We present two variants: round-efficient (RE) and communication-efficient (CE). See a comparison in Table 1; more details are in Table 3.

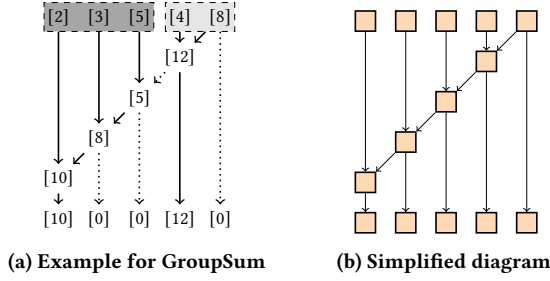


Figure 2: Linear scan method for GroupOP

- **Constant-round GroupSum (§6).** When OP is addition, a protocol for the corresponding binary gate comes for free via local operations only. In this case, we provide a dedicated efficient protocol for GroupSum, which takes only one round in the execution phase (and still a constant number of rounds overall). The best previous GroupSum [15] requires at least four rounds. See Table 4.
- **GroupOP Variants with Propagation (§6).** The basic functionality for GroupOP has the evaluation result in the first slot of the group with the other elements in the group being [0] (cf. Fig. 1). Some applications (such as graph analysis, see below) demand the aggregation result in all slots in the group. We provide an efficient protocol for propagating the evaluation results to enable this variant. The previous protocol [15] takes at least four rounds, while our propagation protocol adds only a single round in the execution phase. See Table 4.
- **Applications to Secure Graph Analysis (§7).** Based on our GroupOP protocols, we construct an efficient secure protocol for computing shortest path distances/routes on weight graphs via the Bellman-Ford Algorithm. We reduce the communication complexity by a factor of $2.4 \sim 5.4$ compared to the very recent and efficient results by Araki *et al.* [5] on a set of graphs with 50 to 3000 nodes and 400 to 50000 edges. See Table 5, 6.

We furthermore provide the following additional results (the details are provided in the full version). In the two-party setting, we construct an optimized GroupMax protocol that improves upon our GroupOP when instantiating OP with Max. We show how to apply our protocols to secure database. Finally, we propose a list concatenation protocol compatible with a length-preserving semi-group (note that a trivial list-concatenation is not length-preserving).

1.1 Previous Works and Our Method Overview

Naive Approach: Linear Scan. A naive approach called *linear scan*, as depicted in Figure 2, can be used to construct any GroupOP. In the diagram, the binary gates represents the OP operation followed by two conditional branches at its output (we opted to not write the if-then gates explicitly). A diagonal branch is turned off (represented as a dotted line) iff it is at a group boundary, using the (private) information parsed via the flag vector $[g]^B$. This method indeed yields the optimal number of binary gates, $N - 1$. However, it incurs a linear depth in N , which reflects the linear number of rounds. This drawback obviously conflicts with our goal of running a *parallel* computation over the input blocks, as the flow indeed runs *sequentially*, one OP gate per each horizontal row in this case.

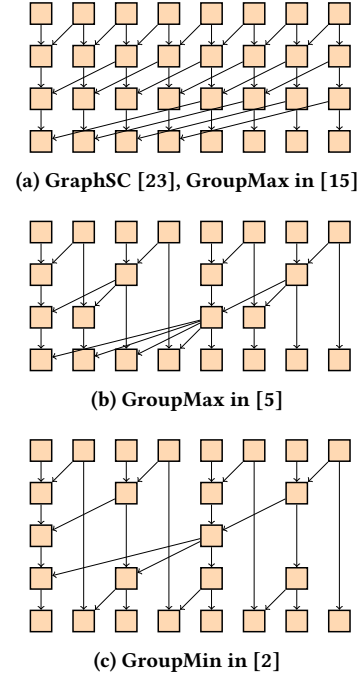


Figure 3: Underlying diagrams in previous works

Previous Works: “Conditional Suffix-OP” based methods. We observe that the linear scan approach actually performs a “suffix-operation” for OP computation, dubbed *Suffix-OP*, meaning that it accumulatively performs OP from the right-most input all the way to the left-most input, and outputs all the accumulative results along the way. The only twist is that it contains conditional branches at each output of the OP gates. All previous works in this area can be seen as computing the same kind of “conditional suffix-OP” but now with more sophisticated circuits that require only logarithmic depth $O(\log N)$. Most of previous works that achieve a logarithmic number of rounds, namely, GraphSC [23], GroupSum in [5], GroupMax in [15], use circuits where all horizontal rows contain N OP gates, and hence incur overall $O(N \log N)$ OP gates. We depict their underlying circuits in Figure 3a,3b. The recent work by [2] is the only logarithm-round scheme that achieve overall $O(N)$ OP gates. This is done via a “sparse” circuit for suffix-OP from the classical Ladner–Fisher parallel prefix computation method [18], as shown in Figure 3c. Intuitively, it can be seen somewhat as a tree-based structure that starts from leaves, consecutively fold one level at a time to the root (or a level near to it), and then unfold back again all the way to the leaves.

Our Method: GroupOP without Computing Full Suffix-OP.

Our main observation is that computing suffix-OP is indeed not necessary for GroupOP. This is in contrast to all previous works above. We instead construct a dedicated circuit that directly computes (plaintext) GroupOP using the optimal number of OP gates, namely, $N - 1$ gates, while can still maintain the least round complexity among all the existing works. We require additional basic gates *independent of* OP, such as if-then, or, permutation; these are shown in Table 2. Assuming MPC protocols for the OP gate and these basic gates, we can then directly instantiate an MPC protocol for GroupOP.

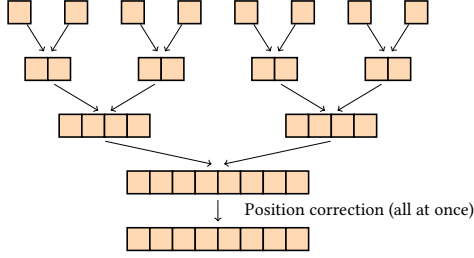


Figure 4: Diagram for our method

We use a complete binary tree *à la* a tournament-based circuit, which is a natural method for computing normal (*i.e.*, ungrouped) multi-input OP products from binary-input OP operations in logarithm depth. Our basic idea to achieve the optimal number of OP gates is to use only one OP gate per each internal (non-input) node in the binary tree, which exactly consists of $N - 1$ nodes. Each node in distance i from the input level store 2^i elements so that the output node has $2^{\log N} = N$ elements, as required for GroupOP.

With this setup, the difficulty then stems from the fact that one OP per each node will be unfortunately insufficient to place the intermediate aggregation results at the right position (namely, the start of the group). Intuitively, this is since each node can only deal with its local position (over the 2^i elements), but not the entire vector *yet*. Our main idea to resolve this problem is to simply “forget” the exact position where the aggregation result should be stored, and instead, to place it *somewhere* as long as it is within its group, while ensuring all the other places in the group be $[0]$. In the main part, we iteratively perform operations at each node until the root node. After that, in the final step, we apply a “position correction” algorithm that privately moves the aggregation results to the start of each group. The position correction can be applied *all at once* in one round in the execution phase (and a constant round overall). We enable all of these mechanisms using some additional labels and a flag per each node together with their proper conditional arguments. We describe our intuition in more details in Section 4.

2 PRELIMINARIES

2.1 Notations

We use bold face for vectors, and let v_j denote the j -th element (starting from 0) of the vector v . Let $v_{[i,j]}$ be the slice $(v_i, v_{i+1}, \dots, v_j)$ of v . We write $v \parallel w$ for the concatenation of v and w . For a vector v , we let $\text{pop}(v)$ indicate the vector obtained by removing the last element from v .

For $b, c \in \mathbb{Z}_2$, we denote the XOR sum, conjunction and disjunction of b and c by $b \oplus c$, bc and $b \vee c$, respectively. For a bit vector $g \in \mathbb{Z}_2^N$, we define the following operations:

- $\text{succ}(g, i)$ returns the address j satisfying $g_j = 1$ and $g_k = 0$ for all $i < k < j$ if exists, and N otherwise.
- $\text{last}(g, i)$ returns the address j satisfying $j \leq i$, $g_j = 1$ and $g_k = 0$ for all $j < k < i$ if exists, and 0 otherwise.

We write $\text{msb}(g)$ and $\text{lsb}(g)$ as syntactic sugar for $\text{succ}(g, -1)$ and $\text{last}(g, N - 1)$, respectively. In other words, $\text{msb}(g)$ and $\text{lsb}(g)$ respectively indicate the positions where the first and last 1 in g occur.

Let $(S_n, \circ)$ be the symmetric group of degree n equipped with the ordinary function composition $(\circ)$. For a vector v of length

N and $\pi \in S_N$, we denote permutation application by $v \cdot \pi$ given by $(v \cdot \pi)_i := v_{\pi(i)}$. Note that $(\cdot)$ is a right group action and thus satisfies $(v \cdot \pi_1) \cdot \pi_2 = v \cdot (\pi_1 \circ \pi_2)$ for all v and $\pi_1, \pi_2 \in S_N$.

2.2 Semigroup with Right Identity Element

The focus of our study is when OP is an associative operation and forms a semigroup. The important property of a semigroup is that the n -fold product can be computed in any order using a binary operation. Remark that we only consider the semigroups that have a right identity element due to technical requirements, and we will simply call them semigroups.

Formally, a *semigroup* M is a tuple (M, op, e) of a set M , a binary operator $op : M \times M \rightarrow M$ on M and an element $e \in M$ satisfying the associativity and identity law axioms as follows:

$$\begin{aligned} op(op(m_1, m_2), m_3) &= op(m_1, op(m_2, m_3)) & \forall m_1, m_2, m_3 \in M \\ m &= op(m, e) & \forall m \in M \end{aligned}$$

The element e is called a *right identity element* of M . Note that we do not assume the uniqueness of e . In general, a semigroup can have several different right identity elements. We write $op_{j=s}^t m_j$ for the op product of $\{m_s, \dots, m_t\}$. If $t < s$, we set the value to a right identity element e .

In this paper, we assume that all elements of M are represented as data of the same size. This assumption is necessary for the privacy of partitioning information, since a secret-sharing scheme cannot handle variable size data while hiding the size. For example, concatenation operation for variable-length-strings forms a semigroup, but we cannot consider it in this study.

The class of semigroups contains various important binary operations. A typical instance of semigroup is addition in the ring of integers. Binary operations used in applications, as in this instance, often have a richer algebraic structure. For example, if e is also a left identity element, *i.e.*, $op(e, m) = m (\forall m \in M)$, then M is called a *monoid*.

Example 1. Multiplication. When $M = \mathbb{Z}_K$, the ordinary multiplication operation $\text{mult} : M \times M \rightarrow M$ forms a monoid with the identity element 1. In addition, M can be extended to the set of fixed-length vectors or the set of fixed-size matrices by considering an appropriate multiplication operation.

Example 2. Max / Min Functions. Let M be a totally ordered set. If M has a minimal element MIN_VALUE , the (binary) maximum function $\text{max} : M \times M \rightarrow M$ forms a monoid $(M, \text{max}, \text{MIN_VALUE})$. Similarly, the (binary) minimal function $\text{min} : M \times M \rightarrow M$ gives a monoid $(M, \text{min}, \text{MAX_VALUE})$ if a maximal element MAX_VALUE of M exists.

Example 3. ArgMax / ArgMin Functions. We can extend max / min functions to calculate arg max / arg min at the same time. Let M be a totally ordered set, and S be a set. Then the following binary operation arg max forms a semigroup structure on $M \times S$:

$$\text{arg max}((v_1, i_1), (v_2, i_2)) = \begin{cases} (v_1, i_1) & \text{if } v_1 \geq v_2 \\ (v_2, i_2) & \text{if } v_1 < v_2 \end{cases}$$

A right identity element is given as $(\text{MIN_VALUE}, i)$ for any i (and thus there is no left identity element). When $i_j = j$, the above function returns the smallest index in the arguments of the maximum $\arg \max_i \{v_i\}$ in the usual sense.

Example 4. First / Last Projections. Let E be the equivalence relation on a set $\mathbb{Z}_2 \times S$ determined by $(0, v) \equiv (0, w)$ for all $v, w \in S$, and let M be the quotient set $(\mathbb{Z}_2 \times S)/E$. We define the binary operator $\text{last} : M \times M \rightarrow M$ as follows:

$$\text{last}((b_1, x_1), (b_2, x_2)) = \begin{cases} (b_1, x_1) & \text{if } b_2 = 0 \\ (b_2, x_2) & \text{if } b_2 = 1 \end{cases}$$

The right identity element is (the equivalent class of) $(0, x)$.

We note that the last projection is an example of embedding a semigroup into monoid. The ordinary definition of the second projection functions $\text{proj}_2 : S \times S \rightarrow S$ only satisfies the axioms of a semigroup, and there is no right identity element. Our last projection operation forms a monoid according to the following fact: Any semigroup can be considered a monoid by adding an additional special element as the identity element. Thus, we can use semigroup operations in our GroupOP scheme by adding the equivalence class of $(0, x)$.

On the other hand, the first projection operation is given as the arg max operation on $\mathbb{Z}_2 \times S$.

Example 5. List Concatenation. Let S be a set of symbols and let $M = S^N \times \mathbb{Z}_K$. We consider $(s, n) \in M$ as a string of length n by assuming that the top n symbols are valid. The binary operator $\text{concat} : M \times M \rightarrow M$ is defined as follows:

$$\text{concat}((s^{(0)}, n^{(0)}), (s^{(1)}, n^{(1)})) := (s, n^{(0)} + n^{(1)})$$

$$s_i := \begin{cases} s_i^{(0)} & \text{if } i < n^{(0)} \\ s_i^{(1)} & \text{if } n^{(0)} \leq i < n^{(0)} + n^{(1)} \\ s_{i-n^{(0)}}^{(1)} & \text{if } n^{(0)} + n^{(1)} \leq i \end{cases}$$

$$n^{(0)} + n^{(1)} := \begin{cases} n^{(0)} + n^{(1)} & \text{if } n^{(0)} + n^{(1)} < N \\ N - 1 & \text{if } N \leq n^{(0)} + n^{(1)} \end{cases}$$

In other words, s is the top N symbols of the concatenation of the valid part of $s^{(0)}$, the valid part of $s^{(1)}$ and the invalid part of $s^{(0)}$. By definition, when $n^{(1)} = 0$, the output coincides with the first argument $(s^{(0)}, n^{(0)})$, so that $(s, 0)$ is a right identity element for any $s \in S^N$.

2.3 Multi-party Computation Model

In the following, we briefly outline the MPC model and related functionalities used in the paper. We say that a protocol securely computes a functionality when the protocol can emulate an ideal version of the functionality in the real world. Furthermore, we say that a protocol securely computes a functionality in a $\mathcal{F}$ -hybrid model when the protocol can securely compute the functionality when given access to an ideal functionality $\mathcal{F}$. For more formal and detailed definitions, see [20].

Arithmetic Black Box. The multi-party computation model we consider is based on an arithmetic black-box functionality $\mathcal{F}_{ABB}$

$\mathcal{F}_{ABB}$

Let $[b]^B$ and $[x]$ denote the identifiers for the values $b \in \mathbb{Z}_2$ and $x \in \mathbb{Z}_K$ stored by the functionality.

Input(b or x): Receive $b \in \mathbb{Z}_2$ or $x \in \mathbb{Z}_K$ from a party and store this.

NOT($[b]^B$): Compute the compliment $\tilde{b} \leftarrow \neg b$ and store $\tilde{b} \in \mathbb{Z}_2$.

AND($[a]^B, [b]^B$): Compute $c = ab$ and store $c \in \mathbb{Z}_2$.

OR($[a]^B, [b]^B$): Compute $c \leftarrow a \text{ OR } b$ and store $c \in \mathbb{Z}_2$.

XOR($[a]^B, [b]^B$): Compute $c \leftarrow a \text{ XOR } b$ and store $c \in \mathbb{Z}_2$.

Add($[x], [y]$): Compute $z = a + b$ and store $z \in \mathbb{Z}_K$.

SMult($x, [y]$): Compute $z = xy$ and store $z \in \mathbb{Z}_K$.

Mult($[x], [y]$): Compute $z = xy$ and store $z \in \mathbb{Z}_K$.

B2A($[b]^B$): Store $b \in \mathbb{Z}_K$.

IfThen($[b]^B, [x]$): Compute $z = bx$ and store $z \in \mathbb{Z}_K$.

Output($[b]^B$ or $[x]$): Output b resp. x to all parties.

Figure 5: Arithmetic black-box functionality

[12]. This captures the secure computation of basic arithmetic operations between a set of participating parties $P_1, \dots, P_n$, and abstracts away the underlying secret sharing scheme and simple protocols. More specifically, we consider a $\mathcal{F}_{ABB}$ operating on binary values $b \in \mathbb{Z}_2$ as well as values $x \in \mathbb{Z}_K$ in a ring $\mathbb{Z}_K$, and when stored by $\mathcal{F}_{ABB}$, we refer to these as $[b]^B$ and $[x]$ respectively (we will also informally refer to $[b]^B$ and $[x]$ as “sharings” of b and x). The basic operations $\mathcal{F}_{ABB}$ is assumed to provide are listed in Figure 5.

For ease of notation, we will for values $a, b \in \mathbb{Z}_2$ use $[-a]^B$ and $[a]^B \oplus [b]^B$ to denote NOT($[a]^B$) and XOR($[a]^B, [b]^B$), respectively. Likewise, for values $x, y \in \mathbb{Z}_K$, we will use $[x] + [y]$ and $x[y]$ to denote Add($[x], [y]$) and SMult($x, [y]$), respectively.

Note that when working with elements of an arbitrary set M , we will implicitly assume an appropriate encoding of elements in M when invoking the $\mathcal{F}_{ABB}$ functionality i.e. for an element $v \in M$, we will let $[v]$ denote an encoding of v in $\mathbb{Z}_K$ stored by $\mathcal{F}_{ABB}$.

Permutation functionalities. In addition to the basic arithmetic functionalities provided by $\mathcal{F}_{ABB}$, our protocols will make use of the following functionalities related to oblivious permutations. More specifically, we consider an extend $\mathcal{F}_{ABB}$ functionality, which we denote $\mathcal{F}_{ABB, \text{Perm}}$ that in addition to the operations provided by $\mathcal{F}_{ABB}$, also provides the following:

- $[z] \leftarrow \text{ApplInvPerm}([v], [\rho])$, where $v \in M^N$, $\rho \in S_N$ and $z = v \cdot \rho^{-1}$.
- $[z] \leftarrow \text{UnApply}([v], [\rho])$, where $v \in M^N$, $\rho \in S_N$ and $z = v \cdot \rho$.

Note that in practice, representing $\rho \in S_N$ as a vector $(\rho(0), \rho(1), \dots, \rho(N-1))$ of length N , and $[\rho]$ as $([\rho(0)], [\rho(1)], \dots, [\rho(N-1)])$ would be more efficient as opposed to encoding ρ as a single ring element (which would require a large ring), but for notational simplicity, we use the above. We additionally assume ApplInvPerm can be applied to both boolean and ring-element vectors i.e., calling ApplInvPerm($[v]^B, [\rho]$) is supported. Finally, we restrict UnApply to be called only after calling ApplInvPerm using the same permutation as input. Section 7 describes the cost of oblivious permutation in the two-party setting.

Binary operation functionality. Lastly, as we consider a generic binary operation op , we require a corresponding functionality for securely computing this. Hence, we consider a further extended functionality, which we denote $\mathcal{F}_{ABB,Perm,OP}$, that in addition to the operations provided by $\mathcal{F}_{ABB,Perm}$, supports the following for a semigroup (M, op, e) .

- $[z] \leftarrow OP([x], [y])$, where $x, y, z \in M$ and $z = op(x, y)$.

Instantiation and cost. We define our protocols and consider the security of these with respect to the above described extended functionality $\mathcal{F}_{ABB,Perm,OP}$. This functionality can be realized by combining protocols from existing works with an appropriate protocol for computing op . In the full version of this paper, we provide concrete OP protocols for each of the instances of the semigroup examples described in Section 2.2.

More specifically, combining additive secret sharing, an appropriate multiplication protocol [8] and the permutation protocols from [6, 10] immediately yield protocols implementing $\mathcal{F}_{ABB,Perm}$ which provides semi-honest security in the two-party setting. Alternatively, the three-party semi-honestly secure multiplication and permutation protocols from [3, 4, 11, 17, 21] could be considered.

In our performance analysis, we assume that the arithmetic operations NOT, XOR, Add, and SMult are local operations which do not incur any communication cost. The cost of the remaining functions in the two-party setting is highlighted in Section 7.

3 DEFINITION FOR GROUPOP

In this section, we give a formal definition of *group-wise aggregation* which we consider in the description of our protocols.

Grouped Vector. A *grouped vector* is a pair $([g]^B, [v])$ of sharings of vectors $g \in \mathbb{Z}_2^N$ and $v \in M^N$ with the same degree. Here, g represents partitioning information. g classifies the entries in v into several groups so that v_i and v_j belong to the same group if and only if $\text{last}(g, i) = \text{last}(g, j)$. We assume that $g_0 = 1$, which guarantees that there is at least one group in a grouped vector.

Functionality for Group-wise Aggregation. Let GroupOP be the functionality for group-wise aggregation defined as follows:

- $[z] \leftarrow \text{GroupOP}([g]^B, [v])$, where $([g]^B, [v])$ is a grouped vector of length N , and $z \in M^N$ satisfies $z_i = 0$ if $g_i = 0$, and $z_i = op_{j=i}^{\text{succ}(g,i)-1} v_j$ otherwise.

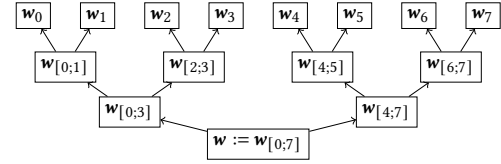
In other words, GroupOP parallelly outputs the OP product of the entries in each group at the starting position of the group. At the other positions, the output value is set to 0.

Let us consider the definition of GroupOP. In previous studies, there are several formulations of group-wise aggregation. For example, the GroupSum protocol proposed in [15] is designed to store the sum of the values in each group at every position in the group. Compared to such aggregation, our scheme is more primitive. Section 6 shows that GroupSum in [15] can be decomposed into two functionalities, and discuss the gap between the two formulations. The idea of storing the output values only at the start positions is based on the aggregation protocol proposed in the GraphSC scheme. We show that the group-wise aggregation in our definition is suitable for the GraphSC paradigm in Section 7.

4 OVERVIEW OF OUR CONSTRUCTION

In this section, we explain the intuition of our GroupOP protocol, which we will formally describe in Section 5. We continue the discussion from the overview in Section 1.1. Recall that our protocol consists of a main part and a position correction part.

Technical Overview of Our Protocol. The goal of our algorithm is to build a *segment tree* for GroupOP for a given grouped vector $w = (g, v)$. A segment tree is a data structure that supports efficient retrieval of information for any segment, by storing only information for some segments as tree node labels: The algorithm assumes a binary tree T as a public parameter, which has the same number (denoted by $|T|$) of leaves as the length of w . Each leaf node is assigned each entry of w from left to right. Then, each internal node is considered to have the concatenation of the slices of the grouped vector assigned to its child nodes. For example, when T is the complete binary tree with height 3, each node is assigned the slice of w as shown below:



A segment tree for GroupOP holds two types of data for each segment in T : One is related to the group-wise aggregation within the segment, using approximated positions to store aggregation results, and the other is related to a permutation on the segment that correct the positions. Our protocol performs two processes in parallel, the main part and the position correction part, each of which computes each type of data.

Data Structure for Group-wise Aggregation with Approximate Storage Positions. In the main part, the algorithm computes a segment tree for GroupOP with approximate storage positions. To do this with a divide-and-conquer strategy that uses only $N - 1$ OP calls, we propose a new data structure. For each node label $w_{[i,j]}$ in T , the parties securely compute a tuple $(f, z, p, s) \in \mathbb{Z}_2 \times M^{j-i+1} \times M \times M$ as follows:

$$w_{[i,j]} = (g_{[i,j]}, v_{[i,j]}) \mapsto \begin{bmatrix} p \\ z \\ s \end{bmatrix}^f$$

The first element of the tuple is

- $f \in \mathbb{Z}_2$, a flag indicating whether $g_{[i,j]}$ contains at least one 1

The values in the other slots in the tuple will depend on the value of f . When $f = 1$,

- $z \in M^{j-i+1}$ is the vector of all zeros except that one of the positions in each inner group of $w_{[i,j]}$ stores the OP product of the values in the group.
- p is the OP product of the values in the left outer group in $w_{[i,j]}$
- s is the OP product of the values in the right outer group in $w_{[i,j]}$

Here, an inner group of w is a group whose range is determined by two occurrences of 1 in g , and the other groups in w are called the left outer group and the right outer group of w , respectively. On the other hand, when $f = 0$, the terms “left outer group” and “right outer group” are ambiguous. Here, we define the value of p and s depending on the tree T . Specifically, if a node is a right child in T ,

- $z \in M^{j-i+1}$ is the all-zero vector
- $p \in M$ is the OP product of the values in $w_{[i:j]}$
- $s \in M$ is 0

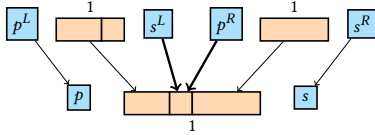
When the node is left child node, p and s are defined in reverse.

Algorithm Overview of the Main Part. Our algorithm proceeds iteratively. For the base case, the algorithm computes the intermediate data for each entry of w , parallelly and independently from each other. Concretely, on input (g, v) , the intermediate data (f, g, p, s) for the node is set as follows:

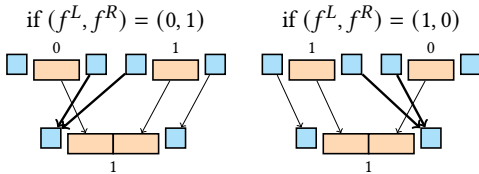
$$\begin{array}{ccc} \text{if } g = 1 & \text{if } g = 0, \text{dir} = L & \text{if } g = 0, \text{dir} = R \\ \begin{array}{|c|c|c|} \hline e & 0 & v \\ \hline \end{array} & \begin{array}{|c|c|c|} \hline 0 & 0 & v \\ \hline \end{array} & \begin{array}{|c|c|c|} \hline v & 0 & 0 \\ \hline \end{array} \end{array}$$

Here, “ $\text{dir} = L$ ” (resp. “ $\text{dir} = R$ ”) means that the node is a left (resp. right) child node of its parent node. When $g = 1$, we regard the left outer group as empty, and set p as a right identity element e . This is the reason why we assume the existence of a right identity element.

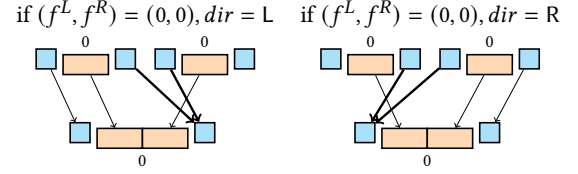
In the iterative step, the algorithm combines the intermediate data of the child nodes. Let (f^L, v^L, p^L, s^L) and (f^R, v^R, p^R, s^R) be the intermediate data for the left and right child nodes, respectively. How the data is combined depends on the values of f^L and f^R . The most critical case is when $(f^L, f^R) = (1, 1)$. In this case, the algorithm computes the OP product of s^L and p^R , stores it at the last position of z^L , then concatenates z^L and z^R . By this concatenation, the right outer group of w^L and the left outer group of w^R are joined and become a new inner group of w of which the OP product has been computed.



In the other cases, the data is combined as above except where the OP product is stored. When $(f^L, f^R) = (0, 1)$, the OP product of s^L and p^R is stored as p . Concatenating z^L and z^R creates a new left outer group. On the other hand, the stored value corresponds to the OP product of the group since s^L is the OP product of w^L . By the same argument, in the case of $(f^L, f^R) = (1, 0)$, the OP product is stored as s .



When $(f^L, f^R) = (0, 0)$, the OP product of s^L and p^R corresponds to the product of w , by definition. The storage position depends on whether the node under consideration is to the left or right of its parent node.



We note that our algorithm requires only one OP call per node, regardless of the value of (f^L, f^R) . Also note that the storage position of each group's OP product depends only on the values of (f^L, f^R) .

Our algorithm only requires OP operations, OR operations, and conditional branching as the components that require communication in the MPC model. In particular, the data storage position depends only on f , so there is no need to perform a complicated computation for an array write operation. Due to these properties, our proposed algorithm is suitable for MPC, and we can easily build a protocol for computing the algorithm, as we will see in this paper.

Data Structure for Position Correction. After executing the main part, it is necessary to correct each OP product's position in z . This is because the main part of our algorithm only guarantees that the OP product of each group is somewhere in the group. Note that each OP product's position in z depends only on g and T . Let $m_T(i)$ be the index at the rightmost leaf of the left subtree under the lowest common ancestor of i and $\text{succ}(g, i)$ (when $\text{succ}(g, i) = |T|$, we set $m_T(i) = |T| - 1$). Then it easily follows that, by definition, $m_T(i)$ corresponds to the position where the main part of our algorithm stores the OP product of the group starting from $g_i = 1$.

We define a new data structure to represent a permutation for position correction on each segment. For a slice of the partitioning $g_{[i:j]}$, the data structure consists of a triple $(f, y, l) \in \mathbb{Z}_2 \times \mathbb{Z}_N^{j-i+1} \times \mathbb{Z}_N$, which holds information about position correction for the output z of the main part.

$$g_{[i:j]} \mapsto \begin{array}{|c|c|} \hline f & l \\ \hline y & \\ \hline \end{array}$$

Here, f is the same as in the main part, i.e., a flag indicating whether there exist at least one 1 in $g_{[i:j]}$. When $f = 0$, we set $l = 0$ and $y = 0$. When $f = 1$, the value of l corresponds to the length of the right outer group of $g_{[i:j]}$. When T is the subtree under the considering node, the value of y_k is defined as follows:

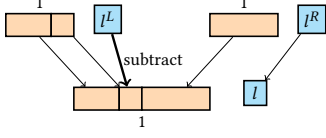
$$y_k = \begin{cases} 1 & q_k \leq k < m_T(q_k), \text{msb}(g_{[i:j]}) \leq k \\ q_k - m_T(q_k) & k = m_T(q_k), \text{msb}(g_{[i:j]}) \leq k \\ 0 & \text{otherwise} \end{cases}$$

where $q_k = \text{last}(g_{[i:j]}, k)$. Each entry of y represents the relative position to which the corresponding elements of z should be moved, to correct the position of the OP products in z . It is easily checked that, after subtracting l from the last entry of y , the rearrangement determined by y is bijective, and moves $m_T(k)$ to k for all $g_k = 1$. Hence, y gives a permutation for the desired position correction.

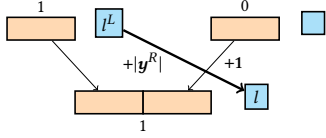
Algorithm Overview of the Position Correction Part. To create the above data for all nodes in T as intended, we define the algorithm iteratively on T . In the base case, on input $g \in \mathbb{Z}_2$, the data for a leaf node is set as follows:

$$\begin{array}{cc} \text{if } g = 1 & \text{if } g = 0 \\ \begin{array}{|c|c|} \hline 1 & 1 \\ \hline \end{array} & \begin{array}{|c|c|} \hline 0 & 0 \\ \hline \end{array} \end{array}$$

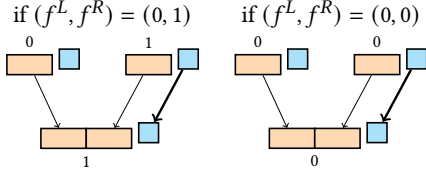
In the iteration step, the behavior changes according to the values of f^L and f^R . In the case of $(f^L, f^R) = (1, 1)$, which determines the storage position of each OP product in the main part, the algorithm subtracts l^L from the last entry of $\mathbf{y}^L$. Since all entries in the right outer group in $\mathbf{y}^L$ store 1, the result corresponds to the relative distance to the start entry of the group.



When $(f^L, f^R) = (1, 0)$, the right outer group is extended, so the algorithm adds 1 to every entry in $\mathbf{y}^R$ and stores $l^L + |\mathbf{y}^R|$ as l , where $|\mathbf{y}^R|$ is the length of $\mathbf{y}^R$. The resulting l corresponds to the length of the extended right outer group.



The other cases are straightforward since the right outer group does not change during the concatenation of $\mathbf{y}^L$ and $\mathbf{y}^R$.



After obtaining the permutation on the main part's output $\mathbf{z}$, we use a permutation application protocol for position correction.

By simplifying the branching structure via flags, the algorithm is easily converted to MPC. The algorithm only requires $O(N)$ invocations of B2A and IfThen, which achieves about the same communication bits as the input size. Since the depth of the algorithm is logarithmic in the length of the grouped input vector, the protocol requires $O(\log N)$ communication rounds in total. However, as long as the protocol is used with the main part of our protocol, most communication is done parallel to the main part, so the number of additional communication rounds required for position correction is 1. The construction of the position correction permutation can also be done in the preparation phase, as it depends only on $\mathbf{g}$ and T . The resulting permutation is reusable for grouped vectors with the same partitioning $\mathbf{g}$.

5 OUR MAIN PROTOCOL FOR GROUPOP

In this section, we propose a protocol for computing GroupOP. To formalize the ideas in the previous section we first give some notations. Our protocol assumes that the parties have a common binary tree as a public parameter of the algorithm. Formally, a binary tree is defined by the following syntax:

$$T ::= \text{node}(T, T) \mid \text{leaf}$$

Let $|T|$ denote the number of occurrences of leaf in T . We assign each leaf node an 0-based index from left to right. We assign either

the constant symbol L or R to each node to indicate the direction of its parent node.

Let T be a binary tree and $\mathbf{x}$ be a vector of length $|T|$. Then, we consider the pair $(T, \mathbf{x})$ as a labeled tree in which each node is assigned the slice of $\mathbf{x}$ whose range is the index of its descendant leaves. To formalize this, we define the split function as follows:

$$\text{split}(T, \mathbf{x}) = (\mathbf{x}^L, \mathbf{x}^R),$$

where $\mathbf{x}^L$ and $\mathbf{x}^R$ satisfy $\mathbf{x} = \mathbf{x}^L \parallel \mathbf{x}^R$ and the lengths of $\mathbf{x}^L$ and $\mathbf{x}^R$ are $|T_1|$ and $|T_2|$ if $T = \text{node}(T_1, T_2)$, respectively.

Let $I_{\text{out}}(\mathbf{g})$, $I_{\text{rout}}(\mathbf{g})$ and $I_{\text{in}}(\mathbf{g})$ be the indices in the left outer group, right outer group and the inner groups of $\mathbf{g}$, respectively. More specifically, $I_{\text{out}}(\mathbf{g}) := \{0, \dots, \text{msb}(\mathbf{g}) - 1\}$, $I_{\text{rout}}(\mathbf{g}) := \{\text{lsb}(\mathbf{g}), \dots, N - 1\}$ and $I_{\text{in}}(\mathbf{g}) := \{\text{msb}(\mathbf{g}), \dots, \text{lsb}(\mathbf{g}) - 1\}$. Note that $I_{\text{out}}(\mathbf{g}) = I_{\text{rout}}(\mathbf{g}) = \{0, \dots, N - 1\}$ when $f = 0$.

5.1 Protocol for Main Part

Functionality. Let Evallabel be the functionality of the main part of our (plain-text) algorithm, which can be defined as follows:

- $([f]^B, [z], [p], [s]) \leftarrow \text{Evallabel}(T, [\mathbf{g}]^B, [\mathbf{v}], \text{dir})$, where T is a binary tree, $\mathbf{g} \in \mathbb{Z}_2^{|T|}$, $\mathbf{v} \in M^{|T|}$, $\text{dir} \in \{L, R\}$ and $(f, z, p, s) \in \mathbb{Z}_2 \times M^{|T|} \times M \times M$ satisfies the following:
 - (1) $f = \bigvee_{j=0}^{|T|-1} \mathbf{g}_j$.
 - (2) $z_i = \begin{cases} \text{op}_{j=\text{last}(\mathbf{g}, i)}^{\text{succ}(\mathbf{g}, i)-1} \mathbf{v}_j & \text{if } i = m_T(\text{last}(\mathbf{g}, i)), i \in I_{\text{in}}(\mathbf{g}) \\ 0 & \text{otherwise} \end{cases}$
 - (3) $p = \begin{cases} f(\text{op}_{j \in I_{\text{out}}(\mathbf{g})} \mathbf{v}_j) & \text{if } \text{dir} = L \\ \text{op}_{j \in I_{\text{out}}(\mathbf{g})} \mathbf{v}_j & \text{if } \text{dir} = R \end{cases}$
 - (4) $s = \begin{cases} \text{op}_{j \in I_{\text{rout}}(\mathbf{g})} \mathbf{v}_j & \text{if } \text{dir} = L \\ f(\text{op}_{j \in I_{\text{rout}}(\mathbf{g})} \mathbf{v}_j) & \text{if } \text{dir} = R \end{cases}$

Note that $(\mathbf{g}, \mathbf{v})$ is not necessarily a grouped vector. In other words, $\mathbf{g}_0 = 1$ may not hold in general. This is since we consider the case where the input is a slice of the input grouped vector.

Protocol. We describe a protocol for computing Evallabel below. The protocol is defined recursively and has two subprotocols, Leaflabel and Nodelabel , for the base and recursion cases, respectively. At the first step, the parties locally decompose the input grouped vector according to T , and then invoke Leaflabel in parallel. At each internal node, the parties invoke Nodelabel with the outputs of Evallabel at its children nodes.

Protocol 1. $\Pi_{\text{Evallabel}}$

Input: $(T, [\mathbf{g}]^B, [\mathbf{v}], \text{dir})$, where $\mathbf{g} \in \mathbb{Z}_2^N$, $\mathbf{v} \in M^{|T|}$ and $\text{dir} \in \{L, R\}$.

Output: $([f]^B, [z], [p], [s])$, where $f \in \mathbb{Z}_2$, $z \in M^{|T|}$ and $p, s \in M$.

- (1) If $T = \text{leaf}$,
 - (a) $([f]^B, [z], [p], [s]) \leftarrow \text{Leaflabel}([\mathbf{g}]^B, [\mathbf{v}], \text{dir})$.
- (2) If $T = \text{node}(T_1, T_2)$,
 - (a) $(([\mathbf{g}^L]^B, [\mathbf{v}^L]), ([\mathbf{g}^R]^B, [\mathbf{v}^R])) := \text{split}(T, ([\mathbf{g}]^B, [\mathbf{v}]))$
 - (b) $([f^L]^B, [z^L], [p^L], [s^L]) \leftarrow \text{Evallabel}(T_1, [\mathbf{g}^L]^B, [\mathbf{v}^L], L)$
 - (c) $([f^R]^B, [z^R], [p^R], [s^R]) \leftarrow \text{Evallabel}(T_2, [\mathbf{g}^R]^B, [\mathbf{v}^R], R)$
 - (d) $([f]^B, [z], [p], [s]) \leftarrow \text{Nodelabel}(([\mathbf{g}^L]^B, [z^L], [p^L], [s^L]), ([\mathbf{g}^R]^B, [z^R], [p^R], [s^R]), \text{dir})$
- (3) Output $([f]^B, [z], [p], [s])$.

Protocol 2. $\Pi_{\text{LeafLabel}}$ **Input:** $([g]^B, [v], \text{dir})$, where $g \in \mathbb{Z}_2$, $v \in M$, $\text{dir} \in \{L, R\}$.**Output:** $([f]^B, [z], [p], [s])$, where $f \in \mathbb{Z}_2$, $z, p, s \in M$.*Preparation phase*

- (1) $[f]^B := [g]^B$.
- (2) $[f] \leftarrow \text{B2A}([f]^B)$.
- (3) $[fe] := e[f]$.

Execution phase

- (1) $[z] := 0$.
- (2) If $\text{dir} = L$,
 - (a) $[p] := [fe], [s] := [v]$.
- (3) If $\text{dir} = R$,
 - (a) $[fv] \leftarrow \text{IfThen}([f]^B, [v])$
 - (b) $[p] := [v] - [fv] + [fe], [s] := [fv]$.
- (4) Output $([f]^B, [z], [p], [s])$.

Protocol 3. $\Pi_{\text{NodeLabel}}$ **Input:** $(([f^L]^B, [z^L], [p^L], [s^L]), ([f^R]^B, [z^R], [p^R], [s^R]), \text{dir})$, where $f^L, f^R \in \mathbb{Z}_2$, $z^L \in M^{n_1}$, $z^R \in M^{n_2}$, $p^L, p^R, s^L, s^R \in M$ and $\text{dir} \in \{L, R\}$.**Output:** $([f]^B, [z], [p], [s])$, where $f \in \mathbb{Z}_2$, $z \in M^{n_1+n_2}$ and $p, s \in M$.*Preparation phase*

- (1) $[f]^B \leftarrow \text{OR}([f^L]^B, [f^R]^B)$.
- (2) $[a]^B := [f^L]^B \oplus [f^R]^B \oplus [f]^B$.

Execution phase

- (1) $[v] \leftarrow \text{OP}([s^L], [p^R])$.
- (2) $[v'] \leftarrow \text{IfThen}([a]^B, [v])$.
- (3) $[z] := \text{pop}([z^L]) \parallel ([v']) \parallel [z^R]$.
- (4) If $\text{dir} = L$,
 - (a) $[fv] \leftarrow \text{IfThen}([f^R]^B, [v])$.
 - (b) $[p] := [p^L] + [fv] - [v']$.
 - (c) $[s] := [s^R] + [v] - [fv]$.
- (5) If $\text{dir} = R$,
 - (a) $[fv] \leftarrow \text{IfThen}([f^L]^B, [v])$.
 - (b) $[p] := [p^L] + [v] - [fv]$.
 - (c) $[s] := [s^R] + [fv] - [v']$.
- (6) Output $([f]^B, [z], [p], [s])$.

5.2 Protocol for Position Correction Part

Functionality. We define the functionality EvalPosCorr of the position correction part of our algorithm as follows:

- $([f]^B, [f], [\mathbf{y}], [l]) \leftarrow \text{EvalPosCorr}(T, [\mathbf{g}]^B)$, where T is a binary tree, $\mathbf{g} \in \mathbb{Z}_2^{[T]}$ and $(f, \mathbf{y}, l) \in \mathbb{Z}_2 \times \mathbb{Z}_N^{[T]} \times \mathbb{Z}_N$ satisfies the following:
 - (1) $f = \bigvee_{j=0}^{|T|-1} g_j$.
 - (2) $\mathbf{y}_k = \begin{cases} f & \text{if } k < m_T(\text{last}(\mathbf{g}, k)) \text{ or } k \in I_{\text{rout}}(\mathbf{g}) \\ \text{last}(\mathbf{g}, k) - k & \text{if } k = m_T(\text{last}(\mathbf{g}, k)) \text{ and } k \in I_{\text{in}}(\mathbf{g}) \\ 0 & \text{if } k > m_T(\text{last}(\mathbf{g}, k)) \text{ or } k \in I_{\text{lout}}(\mathbf{g}) \end{cases}$
 - (3) $l = f(|T| - \text{lsb}(\mathbf{g}))$

In particular, $\mathbf{y} = \mathbf{0}$ and $l = 0$ if $f = 0$. The second entry in the output is an arithmetic sharing of f . The domain size of the output arithmetic sharings must be greater than the length N of the input grouped vector to represent a permutation of the N element set. For simplicity, this paper sets $\mathbb{Z}_N$.

Protocol. We describe a protocol for computing EvalPosCorr below. The protocol invokes protocols for LeafPosCorr and

NodePosCorr, which are respectively the position correction procedures at the base case and the induction step. The protocol only takes $[\mathbf{g}]$ and not the whole grouped vector as input, so that it can be executed in the preparation phase. The round complexity is not a problem as long as it is used for the position correction of the output of the protocol $\Pi_{\text{EvalLabel}}$. We note that, in this case, the OR calls in Step 1 of NodePosCorr and about half of the B2A calls in Step 2 of LeafPosCorr are common to the main part protocol.

Protocol 4. $\Pi_{\text{EvalPosCorr}}$ **Input:** $(T, [\mathbf{g}]^B)$, where $\mathbf{g} \in \mathbb{Z}_2^{[T]}$ **Output:** $([f]^B, [f], [\mathbf{y}], [l])$, where $f \in \mathbb{Z}_2$, $\mathbf{y} \in \mathbb{Z}_N^{[T]}$ and $l \in \mathbb{Z}_N$.

- (1) If $T = \text{leaf}$.
 - (a) $([f]^B, [f], [\mathbf{y}], [l]) \leftarrow \text{LeafPosCorr}([\mathbf{g}]^B)$.
- (2) If $T = \text{node}(T_1, T_2)$,
 - (a) $([\mathbf{g}^L]^B, [\mathbf{g}^R]^B) := \text{split}(T, [\mathbf{g}]^B)$
 - (b) $([f^L]^B, [f^L], [\mathbf{y}^L], [l^L]) \leftarrow \text{EvalPosCorr}(T_1, [\mathbf{g}^L]^B)$
 - (c) $([f^R]^B, [f^R], [\mathbf{y}^R], [l^R]) \leftarrow \text{EvalPosCorr}(T_2, [\mathbf{g}^R]^B)$
 - (d) $([f]^B, [f], [\mathbf{y}], [l]) \leftarrow \text{NodePosCorr}(([\mathbf{g}^L]^B, [f^L], [\mathbf{y}^L], [l^L]), ([\mathbf{g}^R]^B, [f^R], [\mathbf{y}^R], [l^R]))$
- (3) Output $([f]^B, [f], [\mathbf{y}], [l])$.

Protocol 5. $\Pi_{\text{LeafPosCorr}}$ **Input:** $[\mathbf{g}]^B$, where $\mathbf{g} \in \mathbb{Z}_2$.**Output:** $([f]^B, [f], [\mathbf{y}], [l])$, where $f \in \mathbb{Z}_2$, $\mathbf{y}, l \in \mathbb{Z}_N$.

- (1) $[f]^B := [\mathbf{g}]^B$.
- (2) $[f] \leftarrow \text{B2A}([f]^B)$.
- (3) $[\mathbf{y}] := [f], [l] := [f]$.
- (4) Output $([f]^B, [f], [\mathbf{y}], [l])$.

Protocol 6. $\Pi_{\text{NodePosCorr}}$ **Input:** $(([\mathbf{g}^L]^B, [f^L], [\mathbf{y}^L], [l^L]), ([\mathbf{g}^R]^B, [f^R], [\mathbf{y}^R], [l^R]))$, where $f^L, f^R \in \mathbb{Z}_2$, $\mathbf{y}^L \in \mathbb{Z}_N^{n_1}$, $\mathbf{y}^R \in \mathbb{Z}_N^{n_2}$, and $l^L, l^R \in \mathbb{Z}_N$.**Output:** $([f]^B, [f], [\mathbf{y}], [l])$, where $f \in \mathbb{Z}_2$, $\mathbf{y} \in \mathbb{Z}_N^{n_1+n_2}$ and $l \in \mathbb{Z}_N$.

- (1) $[f]^B \leftarrow \text{OR}([f^L]^B, [f^R]^B)$.
- (2) $[f] \leftarrow \text{B2A}([f]^B)$.
- (3) $[v] \leftarrow \text{IfThen}([f^R]^B, [l^L])$.
- (4) $[\mathbf{y}_i] := [\mathbf{y}_i^L]$ for $i \in \{0, \dots, n_1 - 2\}$.
- (5) $[\mathbf{y}_{n_1-1}] := [\mathbf{y}_{n_1-1}^L] - [v]$.
- (6) $[\mathbf{y}_{n_1+i}] := [\mathbf{y}_i^R] + [f] - [f^R]$ for $i \in \{0, \dots, n_2 - 1\}$.
- (7) $[l] := [l^R] + [l^L] - [v] + n_2([f] - [f^R])$.
- (8) Output $([f]^B, [f], [\mathbf{y}], [l])$.

5.3 Protocol for GroupOP

We now describe a protocol for computing GroupOP below.

Protocol 7. Π_{GroupOP} **Input:** $([\mathbf{g}]^B, [\mathbf{v}])$, where $\mathbf{g} \in \mathbb{Z}_2^N$, $\mathbf{v} \in M^N$, $\mathbf{g}_0 = 1$.**Output:** $[\mathbf{w}]$, where $w_i = 0$ if $\mathbf{g}_i = 0$, and $w_i = \text{op}_{j=i}^{\text{succ}(\mathbf{g}, i)-1} v_j$ otherwise.*Preparation phase*

- (1) Build a binary tree T satisfying $|T| = N$ among the parties.
- (2) $([f]^B, [\mathbf{y}], [l]) \leftarrow \text{EvalPosCorr}(T, [\mathbf{g}]^B)$.
- (3) $[\pi_i] := i + [\mathbf{y}_i]$ for $i \in \{0, \dots, N - 2\}$.
- (4) $[\pi_{N-1}] := (N - 1) + [\mathbf{y}_{N-1}] - [l]$.

Execution phase

- (1) $([f]^B, [z], [p], [s]) \leftarrow \text{EvalLabel}(T, [\mathbf{g}]^B, [\mathbf{v}], R)$.
- (2) $[z_{N-1}] := [s]$.
- (3) $[\mathbf{w}] \leftarrow \text{ApplInvPerm}([z], [\pi])$.

Security. The correctness proof is given straightforwardly according to the discussion in the previous section. It is not hard to see that our protocol Π_{GroupOP} securely computes the GroupOP functionality in the $\mathcal{F}_{\text{ABB,Perm,OP}}$ -hybrid model. This is since no value is revealed from the $\mathcal{F}_{\text{ABB,Perm,OP}}$ functionality as all of its algorithms including the sub-protocols here ($\Pi_{\text{Evallabel}}, \Pi_{\text{Leaflabel}}, \Pi_{\text{Nodelabel}}, \Pi_{\text{EvalPosCorr}}, \Pi_{\text{LeafPosCorr}}, \Pi_{\text{NodePosCorr}}$) do not call the Output subroutine of $\mathcal{F}_{\text{ABB,Perm,OP}}$. The details of the security proof are left to the full version.

Some Optimizations. We provide further optimization on our GroupOP protocol. More optimizations are given in the full version.

- The numbers of OR and IfThen calls can be reduced by $\log N$, respectively. Since $g_0 = 1$, the value of f^L at the leftmost node at each level is always 1, so we can set $[f]^B := [f^R]^B$ and $[fv] := [v']$ in the Nodelabel protocol.
- The round complexity of our GroupOP protocol can be reduced by omitting the IfThen calls in Nodelabel at the root node: In this case, we can set $[f^R]^B := 1$ and $[a]^B := 1$, regardless of the original value of f^R . This setting forces the storage position of $op(s^L, p^R)$ to be v' , but is only allowed in the last step. Note that this optimization requires the same setting at the last iteration in EvalPosCorr.

5.4 Performance and Comparison

Complexity Analysis for Our GroupOP. Let $r_{\mathcal{F}}$ (resp. $c_{\mathcal{F}}$) be the number of communication rounds (resp. communication bits) required for computing the functionality $\mathcal{F}$. For simplicity, we consider the case where $N = 2^H$ for some H , and T is the complete binary tree with height H . We assume $r_{\text{IfThen}} = 1$, $r_{\text{B2A}} = 1$ and $r_{\text{OR}} = 1$, as with many existing schemes.

We first measure the round complexity of our GroupOP protocol. Since EvalPosCorr and Evallabel can be done in parallel, the round complexity of our GroupOP protocol is measured as $r_{\text{Evallabel}} + r_{\text{AppInvPerm}}$. Leaflabel takes $\min(r_{\text{B2A}}, r_{\text{IfThen}})$ rounds. In each invocation of Nodelabel, OR can be computed in parallel with other part, so the parties require $r_{\text{OP}} + r_{\text{IfThen}}$ rounds. When we adopt the optimization described above to reduce the round complexity by r_{IfThen} , the total communication rounds for Π_{GroupOP} is $(r_{\text{OP}} + 1) \log N + r_{\text{AppInvPerm}}$.

We next consider the number of communication bits required for our GroupOP protocol. When using the optimizations, EvalPosCorr consists of $(N - \log N - 1)$ OR, $(2N - \log N - 1)$ B2A and $(N - 2)$ IfThen invocations. Evallabel requires $\frac{N}{2}$ IfThen and $(N - 1)$ B2A (if $e \neq 0$) calls for Leaflabel, and 1 OR, 1OP and 2IfThen calls for each invocation of Nodelabel. Since the output of OR and B2A calls are common between Leaflabel and Nodelabel, the total communication bits required for Π_{GroupOP} is counted as follows: $(N - 1)c_{\text{OP}} + (3.5N - \log N - 4)c_{\text{IfThen}} + c_{\text{AppInvPerm}} + (N - \log N - 1)c_{\text{OR}} + (2N - \log N - 1)c_{\text{B2A}}$.

Comparison to Previous Works. Table 2 provides a comparison to previous works. Here, “Ours I” indicates the GroupOP scheme proposed in this section. As in the linear scan scheme, we achieve the optimal number of OP gates, i.e., $N - 1$. In addition, our scheme has almost the same round complexity as the state-of-the-art round-efficient protocols [5, 15, 23] since $r_{\text{AppInvPerm}}$ is 1 in the 2-party

setting [6] or 2 in the 3-party setting [11]. On the other hand, there is a computing overhead of AppInvPerm compared to the other schemes. Nonetheless, our protocol is efficient in many situations because the cost of AppInvPerm has been smaller due to the performance improvements from recent works. To demonstrate, we estimate the complexity of “Ours I” and Gather [5] in the two-party setting with 32-bit arithmetic. We recap efficiency of building blocks in § 7. The communication cost is about $(N - 1)c_{\text{OP}} + 2N \log N + 152N$ bits for ours, and $(N \log N / 2)c_{\text{OP}} + 17N \log N$ bits for [5]. When OP is Min, $c_{\text{OP}} = 234$ bits. Note that ours is always smaller when N is larger than a small constant, e.g., $N \geq 8$.

5.5 Generalized GroupOP Protocol

Assuming constant-round protocols for computing multi-input OR and OP are available, we can further reduce the round complexity. Thanks to the modularity of our divide-and-conquer construction, we can use any rooted tree as the algorithm parameter instead of a binary tree. This extension is a generalization for grouped vectors of Toft’s construction [25] of multi-input Max protocols.

We propose an idea for building generalized protocols here. We defer the details to the full version. Node labels are defined similarly to the binary tree case (We assign R to all nodes except the leftmost child). The only challenge of extending the Evallabel protocol is to define a generalized Nodelabel protocol. This is because we can again use the protocol Leaflabel defined for binary trees. In addition, we employ the GroupSum protocol (§6) for position correction.

We define generalized Nodelabel as follows. On input n labels $\{[f^{(i)}]^B, [z^{(i)}], [p^{(i)}], [s^{(i)}]\}_{0 \leq i \leq n-1}$, we first set $f := \bigvee_{0 \leq i \leq n-1} f^{(i)}$ in the preparation phase. Then the protocol parallelly computes p , s and the value $y^{(i)}$ of the last position of each $z^{(i)}$.

When $f = 1$, we compute the following:

- When $f^{(0)} = 0$, we set $p := op(s^{(0)}, (op_{j=1}^{k_0} p^{(j)}))$ where k_i is the first index after i satisfying $f^{(k)} = 1$. Otherwise, $p := p^{(0)}$.
- When $f^{(i)} = 1$, we set $y^{(i)} := op(s^{(i)}, (op_{j=i+1}^{k_i} p^{(j)}))$. Otherwise, $y^{(i)} := 0$.
- We set $s := op(s^{(l)}, (op_{j=l+1}^{n-1} p^{(j)}))$ where l is the last index satisfying $f^{(l)} = 1$.

When $f = 0$, every $y^{(i)}$ is set to 0, and $op(s^{(0)}, (op_{j=1}^{n-1} p^{(j)}))$ is stored as p or s according to the value of $dir \in \{L, R\}$.

Following the above idea, we propose two protocols $\Pi_{\text{Nodelabel}}^{\text{RE}}$ and $\Pi_{\text{Nodelabel}}^{\text{CE}}$ for computing generalized Nodelabel. There is a trade-off between the OP gate complexity and the round complexity in comparison between the two protocols, as shown in Table 3. By choosing an appropriate tree parameter T , our scheme provides various round optimizations as the technique proposed in [24] for prefix sum. Table 1 shows the complexity in some cases. When the depth of T is 1, the protocol achieves constant communication rounds regardless of N . When T is the doubly logarithmic tree with N leaves, the depth of T is $\log \log N + 1$, so the round complexity is $O(\log \log N)$. In this case, the gate complexity is bound by $O(N \log \log N)$ due to the property of doubly logarithmic trees [16].

Table 2: Performance Comparison among GroupOP Protocols

Protocol	# Rounds	# Gates in Execution Phase			# Gates in Preparation Phase			
		OP	IfThen	others	IfThen	OR	B2A	others
Ours I (§5)	$r \log N + r_{\text{AppInvPerm}}$	$N - 1$	$\frac{5}{2}N - \log N - 2$	$1^\dagger$	$N - 2$	$N - \log N - 1$	$2N - \log N - 1$	-
Ours II (§5, §6)	$r \log N + r_{\text{AppInvPerm}}$	$N - 1$	$\frac{5}{2}N - \log N - 2$	$1^\dagger$	$N - 1$	$N - \log N - 1$	$N - 1$	$2^\ddagger$
[23]	$r \log N + 1$	$N \log N - N + 1$	$N \log N + 1$	-	-	$N \log N - N + 1$	-	-
[15]	$r \log N + 1$	$N \log N - N + 1$	$N \log N + 1$	-	-	$N \log N - N + 1$	-	-
[5]	$r \log N + 1$	$\frac{N \log N}{2}$	$\frac{N \log N}{2} + N$	-	-	$\frac{N \log N}{2}$	-	-
[2]	$2r \log N - 2r + 1$	$2N - \log N - 2$	$3N - \log N - 2$	-	-	$N - \log N - 1$	-	-
Linear Scan	$r(N - 1)$	$N - 1$	$2N - 1$	-	-	-	-	-

N is the input vector length; r is the number of rounds of a base protocol computing OP plus IfThen. $r_{\text{AppInvPerm}}$ is the number of rounds required for AppInvPerm (which is 1 using [6]).

† = AppInvPerm. ‡ = AppInvPerm and UnApply.

Table 3: Comparison among Our (generalized) Nodelabel Protocols

Protocol	# rounds	# gates				IfThen
		2-OP	3-OP	...	N-OP	
$\Pi_{\text{Nodelabel}}^{\text{RE}}$	$r_{\text{If}} + r_{\text{OP}}$	$N - 1$	$N - 2$	...	1	$\frac{N^2 + 3N - 4}{2}^*$
$\Pi_{\text{Nodelabel}}^{\text{CE}}$	$2r_{\text{If}} + r_{\text{OP}}^\dagger$	1	1	...	1	$\frac{N^2 + N - 2}{2}$
$\Pi_{\text{Nodelabel}}$ (§ 5.1)	$r_{\text{If}} + r_{\text{OP}}$	1	-	...	-	2

r_{OP} (resp. r_{If}) denotes the number of communication rounds required for OP (resp. IfThen).

k -OP indicates the OP gate with k inputs.

*: If $N = 2$, we can reduce it by 1. † : If $N = 2$, we can reduce it to $r_{\text{If}} + r_{\text{OP}}$.

6 CONSTANT-ROUND GROUPSUM AND PROPAGATION PROTOCOLS

In this section, we propose constant-round protocols for GroupSum and Propagate. GroupSum is a special case of GroupOP where binary operations are additions, and we give a GroupSum protocol individually from the general scheme in the previous section. On the other hand, Propagate is the functionality for copying values according to the partitioning information, which is often used in combination with group-wise aggregation by GroupOP. We show that these functionalities can be achieved by using each inverse permutation in constant communication rounds. Formally, we consider the following functionalities.

- $[z] \leftarrow \text{GroupSum}([g]^B, [v])$, where $z_i = 0$ if $g_i = 0$, and $z_i = \sum_{j=i}^{\text{succ}(g,i)-1} v_j$ otherwise.
- $[z] \leftarrow \text{Propagate}([g]^B, [v])$, where $z_i = v_{\text{last}(g,i)}$.

Our protocols can be considered as a refinement of the protocol of [15]. Ours inherits the idea of achieving constant communication rounds using oblivious permutation. On the other hand, the original protocol considers the functionality of computing GroupSum and Propagate simultaneously. We consider the two functionalities separately and construct a protocol for each. As a result, we can also construct a round-efficient protocol for the original functionality, shown at the end of this section.

6.1 GenCycle Protocol

The key tool in our GroupSum and Propagate protocols is a cyclic permutation regarding the group information g . As in the EvalPosCorr protocol, the parties generate the cyclic permutation during the preparation phase.

Functionality. Let GenCycle be the functionality for generating the cyclic permutation from g as follows:

- $([\pi], [\pi^{-1}]) \leftarrow \text{GenCycle}([g]^B)$, where $g \in \mathbb{Z}_2^N$ and $\pi \in S_N$ is the permutation on N elements satisfying

$$\pi(i) = \begin{cases} i & \text{if } g_i = 0 \\ \text{succ}(g, i) & \text{if } g_i = 1, i \neq \text{lsb}(g) \\ 0 & \text{if } i = \text{lsb}(g) \end{cases}$$

In other words, π is the cyclic permutation consisting of the indices where g assigns 1 and is in ascending order. For example, on input $g = (11001011)$, then GenCycle outputs a pair of sharings of the cyclic permutation $(01467) \in S_8$ and its inverse.

Protocol. We describe a protocol for computing GenCycle. As Steps 9 and 12 can be run parallelly, the round complexity equals the sum of those for B2A, IfThen, AppInvPerm, and UnApply.

Protocol 8. Π_{GenCycle}

Input: $[g]^B$, where $g \in \mathbb{Z}_2^N, g_0 = 1$.

Output: $([\pi], [\pi^{-1}])$, where $\pi \in S_N$

- (1) $[g_i] \leftarrow \text{B2A}([g]^B)$ for $i \in \{0, \dots, N - 1\}$.
- (2) $[s_i^{(0)}] := \sum_{j=0}^{i-1} (1 - [g_j])$ for $i \in \{0, \dots, N - 1\}$.
- (3) $[s_i^{(1)}] := N - \sum_{j=0}^{N-1} [g_j] + \sum_{j=0}^{i-1} [g_j]$ for $i \in \{0, \dots, N - 1\}$.
- (4) $[\sigma_i] \leftarrow [s_i^{(0)}] + \text{IfThen}([g_i]^B, [s_i^{(1)}] - [s_i^{(0)}])$ for $i \in \{0, \dots, N - 1\}$.
- (5) $[t_i^{(0)}] := i(1 - [g_i])$ for $i \in \{0, \dots, N - 1\}$.
- (6) $[t_i^{(1)}] := i[g_i]$ for $i \in \{0, \dots, N - 1\}$.
- (7) $[u] \leftarrow \text{AppInvPerm}([t^{(1)}], [\sigma])$.
- (8) $[v_{N-1}] := 0, [v_i] := [u_{i+1}]$ for $i \in \{0, \dots, N - 2\}$.
- (9) $[y] \leftarrow \text{UnApply}([v], [\sigma])$.
- (10) $[\pi_i] := [y_i] + [t_i^{(0)}]$ for $i \in \{0, \dots, N - 1\}$.
- (11) $[w_0] := 0, [w_i] := [u_{i-1}]$ for $i \in \{1, \dots, N - 1\}$.
- (12) $[z] \leftarrow \text{UnApply}([w], [\sigma])$.
- (13) $[\pi_0^{-1}] := [u_{N-1}], [\pi_i^{-1}] := [z_i] + [t_i^{(0)}]$ for $i \in \{1, \dots, N - 1\}$.

Correctness. Steps 1-4 give a sharing of the stable sorting σ of g since these steps securely compute the counting sort algorithm. Since $u = t^{(1)} \cdot \sigma^{-1}$, u is the ascending list of the indices satisfying $g_i = 1$ concatenated after consecutive 0 of the same length as the number of 0 in g . By definition, v and w are given by shifting w left and right by 1, respectively. Accordingly, noting that $g_0 = 1$, the following holds:

$$y_i = \begin{cases} 0 & \text{if } g_i = 0 \\ 0 & \text{if } i = \text{lsb}(g) \\ \text{succ}(g, i) & \text{otherwise} \end{cases}, \quad z_i = \begin{cases} 0 & \text{if } g_i = 0 \\ 0 & \text{if } i = 0 \\ \text{last}(g, i - 1) & \text{otherwise} \end{cases}$$

Since adding $t^{(0)}$ results in $y_i = i$ for all i satisfying $g_i = 0$, Step 10 returns the desired permutation. Similarly, since u_{N-1} corresponds to $\text{lsb}(g)$, Step 13 returns the inverse permutation. $\square$

6.2 Our GroupSum Protocol

We describe a protocol for computing GroupSum below. The protocol contains a single invocation of AppInvPerm after the GenCycle invocation. Accordingly, our two GroupOP protocols require the same total communication rounds in many cases.

Protocol 9. Π_{GroupSum}

Input: $([g]^B, [v])$, where $g \in \mathbb{Z}_2^N$, $v \in M^N$, $g_0 = 1$.

Output: $[z]$, where $z_i = 0$ if $g_i = 0$, and $z_i = \sum_{j=i}^{\text{succ}(g,i)-1} v_j$ otherwise.

Preparation phase

(1) $([\pi], [\pi^{-1}]) \leftarrow \text{GenCycle}([g]^B)$.

Execution phase

(1) $[s_i] := \sum_{j=i}^{N-1} [v_j]$ for $i \in \{0, \dots, N-1\}$.

(2) $[t_0] := 0, [t_i] := [s_i]$ for $i \in \{1, \dots, N-1\}$.

(3) $[y] \leftarrow \text{AppInvPerm}([t], [\pi^{-1}])$.

(4) $[z] := [s] - [y]$.

Correctness. By definition, s is the suffix sum of v . Since $y = t \cdot (\pi^{-1})^{-1} = t \cdot \pi$ and $z = s - y$, we have the following:

$$y_i = \begin{cases} s_i & \text{if } g_i = 0 \\ 0 & \text{if } i = \text{lsb}(g) \\ s_{\text{succ}(g,i)} & \text{otherwise} \end{cases}, \quad z_i = \begin{cases} 0 & \text{if } g_i = 0 \\ \sum_{j=i}^{N-1} v_j & \text{if } i = \text{lsb}(g) \\ s_i - s_{\text{succ}(g,i)} & \text{otherwise} \end{cases}$$

Here, the value of $s_i - s_{\text{succ}(g,i)}$ corresponds to the sum from v_i to $v_{\text{succ}(g,i)-1}$, which is our claim. $\square$

Application: Another Method for Position Correction. A pre-defined GroupSum protocol gives another method for position correction in our GroupOP protocol. Let us consider the following:

Protocol 10. $\Pi_{\text{GroupOP}}^{(II)}$

Input: $([g]^B, [v])$, where $g \in \mathbb{Z}_2^N$, $v \in M^N$, $g_0 = 1$.

Output: $[w]$, where $w_i = 0$ if $g_i = 0$, and $w_i = \text{op}_{j=i}^{\text{succ}(g,i)-1} v_j$ otherwise.

Preparation phase

(1) Build a valid binary tree T with $|T| = N$ among the parties.

Execution phase

(1) $([f]^B, [z], [p], [s]) \leftarrow \text{EvalLabel}(T, [g]^B, [v], R)$.

(2) $[x_{N-1}] := [s], [x_i] := [z_i]$ for $i \in \{0, \dots, N-2\}$.

(3) $[w] \leftarrow \text{GroupSum}([g]^B, [x])$.

Correctness. By the definition of EvalLabel, one of the entries in each group in x stores the OP product of the group in v , and the other entries in x are all 0. Thus, GroupSum also works well to securely compute GroupOP. $\square$

Compared to the method using EvalPosCorr, an advantage of this method is that GroupSum can be computed independently of the structure of T . By avoiding the inductive construction of the position correction permutation, the number of communication rounds can be reduced to $O(1)$. Our two methods follow a trade-off between round complexity and communication complexity for the preparation phase. Table 2 provides a comparison between the two methods in terms of gates. Here, “Ours II” indicates the protocol using GroupSum. When combined with our EvalLabel protocol, the round complexity in the preparation phase of position correction is not essential, so the protocol $\Pi_{\text{EvalPosCorr}}$ is a good choice. In

contrast, when we use a constant-round (or loglog-round) EvalLabel protocol, it seems better to use GroupSum for round efficiency.

6.3 Our Propagate Protocol

We next describe a protocol for computing Propagate below. A single AppInvPerm call is necessary to compute the protocol during the execution phase.

Protocol 11. $\Pi_{\text{Propagate}}$

Input: $([g]^B, [v])$, where $g \in \mathbb{Z}_2^N$, $g_0 = 1$, $v \in M^N$.

Output: $[z]$, where $z_i = v_{\text{last}(g,i)}$ otherwise.

Preparation phase

(1) $([\pi], [\pi^{-1}]) \leftarrow \text{GenCycle}([g]^B)$

Execution phase

(1) $[x] \leftarrow \text{AppInvPerm}([v], [\pi])$.

(2) $[y_0] := 0, [y_i] := [x_i]$ for $i \in \{1, \dots, N-1\}$.

(3) $[z_i] := \sum_{j=0}^i ([v_j] - [y_j])$ for $i \in \{0, \dots, N-1\}$.

Correctness. Since $x = v \cdot \pi^{-1}$, the value of y_i is equal to 0 if $i = 0$, v_i if $g_i = 0$, and $v_{\text{last}(g,i-1)}$ if $g_i = 1$ and $i \neq 0$. By definition, $z_i = \sum_{j=0}^i (v_j - y_j)$ holds, so that for all $i \in \{0, \dots, N-1\}$, we have $z_i = v_0 + \sum \{v_j - v_{\text{last}(g,j-1)} \mid 1 \leq j \leq i, g_j = 1\} = v_{\text{last}(g,i)}$. $\square$

Comparison with GroupSum Protocol of HIKC21. We next give a protocol for computing GroupSum and Propagate simultaneously. This functionality is considered in [15] and shown to be useful for decision tree training. The definition of our protocol is described below. Our protocol is built by simply combining the above two protocols, so it is easy to show the correctness.

Protocol 12. $\Pi_{\text{GroupSum+Propagate}}$

Input: $([g]^B, [v])$, where $g \in \mathbb{Z}_2^N$, $v \in M^N$, $g_0 = 1$.

Output: $[z]$, where $z_i = \sum_{j=\text{last}(g,i)}^{\text{succ}(g,i)-1} v_j$.

Preparation phase

(1) $([\pi], [\pi^{-1}]) \leftarrow \text{GenCycle}([g]^B)$.

Execution phase

(1) $[s_i] := \sum_{j=i}^{N-1} [v_j]$ for $i \in \{0, \dots, N-1\}$.

(2) $[t_0] := 0, [t_i] := [s_i]$ for $i \in \{1, \dots, N-1\}$.

(3) $[w] \leftarrow \text{AppInvPerm}([s], [\pi])$.

(4) $[x] \leftarrow \text{AppInvPerm}([t], [\pi^{-1}])$.

(5) $[y_0] := 0, [y_i] := [w_i] - [t_i]$ for $i \in \{1, \dots, N-1\}$.

(6) $[z_i] := \sum_{j=0}^i ([s_i] - [x_i] - [y_i])$ for $i \in \{0, \dots, N-1\}$.

Table 4 shows a comparison to the GroupSum protocol of [15]. By considering GroupSum and Propagate separately, our scheme successfully separates the preparation and execution phases, and achieves further parallelized computation. In the scheme of [15], the execution phase requires a sequential invocations of two IfThen , AppInvPerm , and UnApply , so that it requires at least four communication rounds. On the other hand, the execution phase of our scheme only requires $r_{\text{AppInvPerm}}$ rounds. This also shows that our GroupSum and Propagate protocols each improve the round complexity in the execution phase compared to the scheme of [15].

7 APPLICATION: GRAPH ANALYSIS

We apply our GroupOP to securely compute shortest path distances in weighted graphs. Following [2, 5], we use the Bellman-Ford (BF) algorithm to operate breadth first search.

Encrypted Graph. Let M be a totally ordered set. An *encrypted graph* is a pair $([u], [v])$ of encrypted/secret-shared vectors, where

Table 4: The Number of Gates in GroupSum + Propagate Protocols

Phase	Operation	[15]		Our $\Pi_{\text{GroupSum+Propagate}}$ (§6.3)	
		# gates	depth	# gates	depth
prep.	B2A	$N - 1$	1	$N - 1$	1
	IfThen	$N - 1$	1	$N - 1$	1
	ApplInvPerm	-	-	1	1
	UnApply	-	-	2	1
exec.	IfThen	$2(N - 1)$	2	-	-
	ApplInvPerm	1	1	2	1
	UnApply	1	1	-	-

prep. and exec. are the preparation and execution phases of GroupSum, respectively.

$u, v \in M^N$. We note that in an encrypted graph, nodes and edges are represented as the same object. If $u \neq v$, then a pair (u, v) represents an edge from u to v . If $u = v$, then (u, v) represents the node corresponding to u .

Protocol Overview. We consider the following functionality:

- $[d] \leftarrow \text{SPD}([u], [v], [w], [s]^B, K)$, where $([u], [v])$ is an encrypted graph, $[w]$ is its edge weights, $s \in \mathbb{Z}_2^N$, K is the number of iteration steps, and $d \in M^N$ satisfies $d_i = \min\{d(i, j) \mid s_j = 1, u_j = v_j\}$ if $u_i = v_i$, and $d_i = 0$ otherwise.

The high-level structure of the protocol is the same as the protocol of Araki *et al.* [5]: In the preparation phase, the protocol takes an encrypted graph as input, and converts it into two kinds of grouped vectors, which are defined such that edges with the same tail (resp. head) node belong to the same group. The protocol also outputs two encrypted/secret-shared permutations for mutual conversion between the two grouped vectors. In the execution phase, by alternately applying the two permutations and Propagate and GroupOP, the protocol securely computes the Bellman-ford algorithm in the GraphSC framework. Particularly, we use our GroupMin and GroupOR. The concrete construction is rather straightforward from the plaintext Bellman-Ford and we describe it in the full version.

Performance Comparison. By counting arguments, we obtain the performance of ours and that of [5] in Table 5 and 6. The graph settings are from [2]. Each cell shows the number of communication bits per party required for a single protocol execution. All the protocols use 32-bit arithmetic sharings for all weights and indices. We refer to [2] for the values of their protocol. The [5] protocol communication estimation uses the same protocol as ours, except that the log-round Scatter and Gather protocols are used instead of the Propagate, GroupOR and GroupMin protocols. In ours and [5] protocols, we use the state-of-the-art concrete cost per party for each basic operation in the two-party setting shown in Table 7. Here, StableSort is the functionality for oblivious stable sorting we assume in the SPD protocol. We set the number of iterations K to the number of nodes minus 1. For Table 6, we estimate the running times as $(\text{comm.bits})/(\text{bandwidth}) + (\text{rounds}) \times (\text{latency})$. Since the values of *bandwidth* and *latency* depend on the network setting, we consider three different network environments as [2]; “high-bandwidth low-latency” (HBLL), “high-bandwidth high-latency” (HBHL), and “low-bandwidth high-latency” (LBHL) settings.

From Table 5, the communication complexity for ours is improved by a factor of 2.4 ~ 5.4 compared to [5] on a set of graphs with 50 to 3000 nodes and 400 to 50000 edges. As the asymptotic improvement in gate complexity suggests, the advantage of ours

improves as the size increases; the improvement is 2.4 times over [5] in the smallest case (400 edges), while it is 5.5 times in the largest case (499k edges). In Table 6, improvements in the low-latency setting reflect the communication complexity improvements. In the high-latency settings, the round complexity has a significant impact on running time as the graph size increases. The improvement of our scheme comes from the constant-round Propagate protocol, compared to the log-round Scatter protocol.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is supported by JSPS, KAKENHI Grant Numbers JP21H05052, JP19H01109, JST AIP Acceleration Research JPMJCR22U5, and JST, PRESTO Grant Number JPMJPR21P6, Japan.

Table 5: Communication Complexity among Secure BF protocols

Graph Size		Algorithm 1-3		Π_{SPD} with	
nodes	edges	(Ver. 1) of [2]	Gather [5]	Our GroupOP	
50	400	32 MB	4.1 MB	1.7 MB	
50	1225	74 MB	13.9 MB	4.8 MB	
200	600	165 MB	31 MB	10.3 MB	
200	19.9 k	2900 MB	1114 MB	260 MB	
700	2100	1570 MB	441 MB	122 MB	
700	10 k	5 GB	1937 MB	465 MB	
700	244 k	110 GB	56 GB	10.6 GB	
1 k	3 k	4 GB	898 MB	247 MB	
1 k	20 k	13 GB	5783 MB	1297 MB	
1 k	499 k	320 GB	172 GB	31 GB	
3 k	9 k	65 GB	9294 MB	2206 MB	
3 k	50 k	133 GB	46 GB	9746 MB	

Table 6: Running Time Estimation among Secure BF (ms)

Graph Size		[2]			[5]			Ours		
nodes	edges	HBLL	HBHL	LBHL	HBLL	HBHL	LBHL	HBLL	HBHL	LBHL
50	400	2.8k	799k	799.5k	32	76.8k	77.1k	13	63.0k	63.2k
50	1225	5.2k	1155k	1145k	106	92.5k	93.5k	36	74.8k	75.2k
200	600	17.9k	3270k	3261k	235	337k	339k	79	273k	274k
200	19.9 k	162k	5623k	5806k	8.5k	504k	581k	1980	394k	412k
700	2100	147k	15.4M	15.4M	3.4k	1404k	1433k	928	1121k	1129k
700	10 k	348k	18.5M	18.7M	14.8k	1639k	1771k	3546	1292k	1323k
700	244 k	5823k	35.9M	42.9M	426k	2498k	6335k	81.1k	1705k	2435k
1 k	3 k	288k	22M	22.2M	6.9k	2007k	2068k	1884	1602k	1619k
1 k	20 k	917k	25.9M	26.7M	44.1k	2524k	2921k	9895	1970k	2059k
1 k	499 k	16.6M	63.4M	81.7M	1308k	4428k	16.2M	236k	2675k	4795k
3 k	9 k	2318k	79.6M	81.1M	70.9k	7031k	7669k	16.8k	5537k	5688k
3 k	50 k	6675k	93.3M	100M	354k	8274k	11.5M	74.4k	6314k	6983k

HBLL, HBHL, and LBHL indicate the network settings where the bandwidth is 1 Gbps with no latency, 1 Gbps with 40ms latency, and 100 Mbps with 40ms latency, respectively.

Table 7: Round and Communication Complexity for Two-Party Protocols

Functionality	# rnds	online comm.	offline comm.	reference
AND / OR	1	2	1	[6, 8]
B2A	1	1	1	[6]
IfThen	1	33	32	[6]
ApplInvPerm	$2(1)^*$	$N(\lambda + 2 \log N)$	$N(\lambda + \log N)$	[6]
UnApply	1	λN	λN	[6]
StableSort	48	$2560N$	$5632N$	[6]
Min	3	234	2255	[7]

N is the input vector length, λ denotes the bit size of each entry in the input vector.

* The value in parentheses is the number of rounds in the execution phase.

REFERENCES

- [1] Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. 2021. Secure training of decision trees with continuous attributes. *PoPETs 2021*, 1 (Jan. 2021), 167–187.
- [2] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2021. Parallel Privacy-Preserving Shortest Path Algorithms. *Cryptography* 5, 4 (2021).
- [3] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. 2018. Generalizing the SPDZ Compiler For Other Protocols. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 880–895.
- [4] 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 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 805–817.
- [5] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 610–629.
- [6] Nuttapong Attrapadung, Goichiro Hanaoka, Takahiro Matsuda, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, Tadanori Teruya, and Kazunari Tozawa. 2021. Oblivious Linear Group Actions and Applications. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 630–650.
- [7] Nuttapong Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, and Kazunari Tozawa. 2022. Memory and Round-Efficient MPC Primitives in the Pre-Processing Model from Unit Vectorization. In *ASIACCS 22*, Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako (Eds.). ACM Press, 858–872.
- [8] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO'91 (LNCS, Vol. 576)*, Joan Feigenbaum (Ed.). Springer, Heidelberg, 420–432.
- [9] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *20th ACM STOC*. ACM Press, 1–10.
- [10] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-Shared Shuffle. In *ASIACRYPT 2020, Part III (LNCS, Vol. 12493)*, Shiho Moriai and Huaxiong Wang (Eds.). Springer, Heidelberg, 342–372.
- [11] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. 2019. An Efficient Secure Three-Party Sorting Protocol with an Honest Majority. *Cryptology ePrint Archive*, Report 2019/695. .
- [12] Ivan Damgård and Jesper Buus Nielsen. 2003. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *CRYPTO 2003 (LNCS, Vol. 2729)*, Dan Boneh (Ed.). Springer, Heidelberg, 247–264.
- [13] Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. 2014. Practical Secure Decision Tree Learning in a Teletreatment Application. In *FC 2014 (LNCS, Vol. 8437)*, Nicolas Christin and Reihaneh Safavi-Naini (Eds.). Springer, Heidelberg, 179–194.
- [14] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th ACM STOC*, Alfred Aho (Ed.). ACM Press, 218–229.
- [15] Koki Hamada, Dai Ikarashi, Ryo Kikuchi, and Koji Chida. 2021. Efficient decision tree training with new data structure for secure multi-party computation. *arXiv:2112.12906 [cs.CR]*
- [16] Joseph JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.
- [17] Ryo Kikuchi, Dai Ikarashi, Takahiro Matsuda, Koki Hamada, and Koji Chida. 2018. Efficient Bit-Decomposition and Modulus-Conversion Protocols with an Honest Majority. In *ACISP 18 (LNCS, Vol. 10946)*, Willy Susilo and Guomin Yang (Eds.). Springer, Heidelberg, 64–82.
- [18] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (1980), 831–838.
- [19] Peeter Laud. 2015. Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees. *PoPETs 2015*, 2 (April 2015), 188–205.
- [20] Yehuda Lindell. 2016. How To Simulate It - A Tutorial on the Simulation Proof Technique. *Cryptology ePrint Archive*, Report 2016/046. .
- [21] Payman Mohassel and Peter Rindal. 2018. ABY³: A Mixed Protocol Framework for Machine Learning. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 35–52.
- [22] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 19–38.
- [23] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 377–394.
- [24] Kazuma Ohara, Kazuo Ohta, Koutarou Suzuki, and Kazuki Yoneyama. 2014. Constant Rounds Almost Linear Complexity Multi-party Computation for Prefix Sum. In *AFRICACRYPT 14 (LNCS, Vol. 8469)*, David Pointcheval and Damien Vergnaud (Eds.). Springer, Heidelberg, 285–299.
- [25] Tomas Toft. 2007. *Primitives and Applications for Multi-party Computation*. Ph. D. Dissertation. University of Aarhus.
- [26] Heribert Vollmer. 1999. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag, Berlin, Heidelberg.
- [27] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd FOCS*. IEEE Computer Society Press, 160–164.