An Efficient Secure Three-Party Sorting Protocol with an Honest Majority

Koji Chida[1], Koki Hamada[1], Dai Ikarashi[1], Ryo Kikuchi[1], Naoto Kiribuchi[1], and Benny Pinkas[2]

[1] NTT,
kikuchi_ryo@fw.ipsj.or.jp
[2] Bar-Ilan University,
benny@pinkas.net

Abstract. We present a novel three-party sorting protocol secure against passive adversaries in the honest majority setting. The protocol can be easily combined with other secure protocols which work on shared data, and thus enable different data analysis tasks, such as data deduplication, set intersection, and computing percentiles.

The new sorting protocol is based on radix sort. It is asymptotically better compared to previous sorting protocols since it does not need to shuffle the entire length of the items after each comparison step. We further improve the concrete efficiency by using not only optimizations but also novel protocols, which are independent of interest.

We implemented our sorting protocol with those optimizations and protocols. Our experiments show that our implementation is concretely fast. For example, sorting one million 20-bit items takes 4.6 seconds in 1G connection. It enables a new set of applications on large-scale datasets since the known implementations handle thousands of items about 10 seconds.

Keywords: Secure computation, sorting, honest majority

1 Introduction

Sorting is a basic building block for many data analysis tasks such as private set intersection between many parties, join and equi-join, or data deduplication. We present an extremely efficient secure protocol for sorting data that is shared between three parties. Our new protocol is based on radix sort. It can be easily composed with circuits or other protocols on shared data which can implement arbitrary pre-sorting and post-sorting computations on the data, in order to compute more advanced functionalities that depend on sorting as a building block.

In order to describe and compare the asymptotic communication complexity of secure sorting, let us use the following notation: Let m be the number of items. We assume each item to be a (key, value) pair, where ℓ is the bit-length of the key (which is used for ordering the items in the sorting algorithm), and ℓ' is the bit-length of the value associated with the item.

Our new sorting protocol has a better asymptotic communication complexity than that of previous protocols for the same task — a sorting protocol based on the Sharemind system [27], oblivious quicksort [18], and secure sorting using a Batcher sorting network. (All protocols are run between three parties which share the data.) The different communication complexities are described in Table 1.

We further improve the concrete efficiency by using not only optimizations but also novel protocols: multiplication for Shamir's scheme, resharing, and shuffling protocols. These protocols are fundamental and independent of interest. Those optimizations and novel protocols significantly improve efficiency by reducing roughly 85% of communication.

We then implement our sorting protocol with the optimization techniques and novel protocols. We demonstrate in Section 6.2 that our implementation is, concretely, substantially efficient. For example, sorting one million keys of length $\ell = 20$ bits, our implementation takes 1.2 and 4.6 seconds in 10G and 1G connections. It enables a new set of applications on large-scale datasets, such as a multiset operation on 10 million

items, since the known best implementation [17] can handle only thousands of items within 10 seconds. We demonstrate it to implement data deduplication protocol by using sorting protocol. The protocol elminates duplicated items from one million 20-bit items within 1.4 and 5.3 seconds in 10G and 1G connections.

Table 1. Asymptotic Communication of secure sorting protocols.

Protocol	**Communication (bits)**
Ours	$O(\ell m \log m + \ell' m)$
Sharemind system [27, 6]	$O(\ell m \log m + \ell^2 m + \ell\ell' m)$
Quicksort [18]	$O(\ell m \log m + \ell' m \log m)$
Batcher sorting network	$O(\ell m \log^2 m + \ell' m \log^2 m)$

1.1 Sorting for MPC

Sorting is a basic building for many data analysis tasks. We present a sorting protocol in a setting of three servers, and data that is shared between these servers. This building block enables much greater flexibility compared to previous solutions to data analysis problem.

As an example for a data analysis task, consider the private set intersection problem (PSI) and its variants (see, e.g. [29] and references within). Previous solutions for this problem were mostly in the two-party setting, between two parties that have access to their own private input sets. Most of these solutions consisted of protocols that were specific for PSI, and were not easily composable with other secure protocols. For example, it is hard to use these protocols to compute the maximum of the values which appear in the intersection.

There are a few protocols that compute PSI using a circuit (e.g. [20, 30]). Such circuit-based protocols can be easily used to compose the result of the intersection with another computation which computes a function of the intersection. Existing circuit-based protocols have an overhead which is greater than that of specific PSI protocols. There are also a few protocols for PSI in the multi-party setting (with more than two parties) [23, 5]. See [25] and references within.

A sorting protocol enables to easily compute the intersection of n inputs sets (by sorting the union of the input sets and then looking in the sorted union for n consecutive items which are equal to each other [20]). A sorting protocol on shared data solves many of the issues associated with previous PSI protocols: Many parties (data owners) can share their data between three servers, which then run the secure computation. This **communication pattern** is very appealing for data owners since they do not need to communicate with (or even be aware of) other data owners. Furthermore, since sorting is computed using a circuit, it is easy to **compose different functions** on top of the result of the sorting algorithm. We describe here different examples of tasks which can efficiently and securely be computed with this new building block:

Threshold multi-party PSI Assume that many parties have private sets of data and wish to compute the intersection of these sets. Furthermore, assume that they wish to compute a relaxed version of the intersection functionality, which identifies each item which appears in at least 75% of the sets. This computation can be relevant for example for sharing cyber threat information between different companies, which collaborate in order to achieve better security through their combined knowledge: As the simplest example assume that the companies wish to identify if the same Indicator of Compromise – IOC (for example, a suspicious IP address) was observed by many of the companies.

Sorting enables an easy solution to computing this functionality: Suppose that there are n companies. Each company separately shares its data between the servers. The servers compute a circuit which first sorts all inputs, and then scans the list of sorted values looking for an item that appears in at least $0.75n$ consecutive locations. (This final scan is done in linear time.)

Data deduplication A specific example of threshold multi-party PSI is data deduplication: Many parties submit data sets that might contain duplicate items, and the goal is to compute a set which contains a single copy of each item which appeared in the input sets. This computation is useful for cleaning data from duplicates before further analyzing it. It was also suggested in the iDash 2017 privacy challenge, in a context where many hospitals have private lists of patients, and it is needed to find patients which are registered in more than one hospital.[3]

Computing percentiles Assume that there are multiple employers, and each of them has a database of contractors with their salaries. The total number m of contractors is known. The goal is to output the salaries at the 10%, 20%,...,90% percentiles. Sorting enables to securely implement this computation by sorting the salary of each contractor, and the output is the values at locations $0.1m, 0.2m, \ldots$.

1.2 Sorting

We review previous work on secure protocols for sorting in Section 1.5. In short, secure sorting can be implemented using sorting networks, such as Batcher's merge sort [4] with a complexity of $O(m \log^2 m)$ comparisons with a small constant factor. (Huang et al. [19] used a Bitonic merger network for sorting two sets for computing their intersection). Secure sorting can also be implemented using data-oblivious sorting algorithms, in which the control flow of the algorithm is independent of the input. For example, using the randomized Shellsort algorithm [15] was suggested by Goodrich. Hamada et al. [18] described a secure sorting algorithm that first randomly shuffled the data, and then applied quicksort to the shuffled data. The control flow of quicksort is data dependent, but since it is applied to a random permutation of the data it does not leak any information.

Bogdanov et al. proposed a secure sorting protocol which is based on radix sort and is the most similar to our protocol [6]. That protocol repeatedly sorts the items according to the successive bits of the keys starting from the least significant bit. For each bit, the parties compute a (shared) permutation which represents how to sort items based on that bit, and then permute the items according to this permutation.

The sorting protocol that we present, on the other hand, does not need to permute the full values in each of the bit-sorting steps of the protocol. Instead, it uses a new protocol component that computes the composition of shared permutations, in order to compose the permutations that sort by each of the key bits. As a result, the protocol only permutes the items once, instead of ℓ times in [8]. This significantly improves the performance since the bulk of the communication was previously used to permute the items according to each bit.

1.3 Contributions

Our contribution is a new sorting protocol based on radix sort and its optimized implementation. Our protocol and implementation have the following advantages compared to previous works on secure sorting of shared data.

- *Low communication complexity.* The asymptotic communication complexity of our sorting protocol is $O(\ell m \log m + m\ell')$ bits, which improves the communication of all previous protocols (as described in Table 1).
- *Novel fundamental protocols.* Novel multiplication for Shamir's scheme, resharing, and shuffling protocols are proposed to improve concrete efficiency. These protocols are independent of interest since they can be used for another protocol. Combining dedicated optimizations for the sorting protocol, we reduce roughly 85% of communication.

[3] Track 1: De-duplication for Global Alliance for Genomics and Health (GA4GH), `http://www.humangenomeprivacy.org/2017/competition-tasks.html`

– *Concretely efficient implementation.* We implemented our sorting protocol with the novel protocols and optimizations. Our implementation showed that, within 20 seconds, we can sort one million 60-bit items in 1G and 10G connections and $10,000$ items even in Internet simulation (50Mbps and 20ms roundtrip latency), while known implementations [6, 18] takes 10 seconds to handle a thousand items in 1G connection.

  Therefore, our implementation enables a new set of applications on datasets whose sizes were before beyond the reach of secure sorting protocols. We demonstrate it to implement sorting-based data deduplication protocol. In our experiment, we showed that it is possible to eliminate the duplication in one million 20-bit items within several seconds in both 10G and 1G connections.
– *Stable sort.* Our protocol implements stable sort, meaning that two items with the same key are always ordered according to their initial position. Stable sort reorders a vector of values in a *reproducible* fashion. Previous sorting protocols based on quicksort are unstable [18].

1.4 Goal

Our goal in this paper is to provide an implementation of efficient secure sorting protocol on shared data to enable sorting-based applications on large-scale datasets in practice. This is why we improve not only asymptotic complexity but also concrete efficiency through optimizations. To sort large-scale data, our most important measure is the throughput with respect to the number of items. Furthermore, the input and output of the protocol must be shared between the servers, so that the protocol can serve as a building block of a general secure computation framework. For example, the protocol is expected to PSI which are the results of secure computation, and the output of the protocol might be input to other secure protocols.

1.5 Related Work

Efficient sorting for MPC can be implemented based on sorting networks. Ajtai et al. proposed an asymptotically optimal sorting network known as the AKS sorting network, which has a complexity of $O(m \log m)$ comparisons, where m is the number of input items [1]. However, this algorithm is not practical since its constant factor is very large. By contrast, Batcher's merge sort [4] has a complexity of $O(m \log^2 m)$ comparisons with a small constant, and is more efficient for any reasonable input size.

Recently, data-oblivious sorting algorithms have been studied with the aim of using them in MPC schemes. We say that an algorithm is data-oblivious if the control flow of the algorithm is independent of the input. Similar to sorting networks, data-oblivious sorts are also efficiently applied to MPC protocols. Goodrich proposed a data-oblivious sort called randomized Shellsort [15]. Although randomized Shellsort returns a wrong output with low probability, it exhibits a complexity of $O(m)$ rounds and $O(m \log m)$ comparisons.

Quicksort is very efficient in practice, but the control flow of this algorithm is data dependent and therefore might leak information about the inputs, even if the comparisons themselves are implemented using a secure algorithm. Hamada et al. [18] describe a secure sorting algorithm that first randomly shuffles the data, and then applies quicksort to the shuffled data. Therefore, since quicksort is applied to a random permutation of the data, the control flow of the algorithm is independent of the original order of the inputs and is easily simulatable.

Zhang proposed several data-oblivious sorting algorithms [32]. Zhang's bead-sort and counting-sort cleverly compute the sorted list of items without comparisons. They convert the input key values into an aggregated form and then reveal the keys in a sorted form. These algorithms require $O(Rm)$ comparisons, where R represents the range of input values. However, these algorithms can handle only keys. That is, all the values to be sorted must be treated as keys. Zhang also proposed an algorithm with $O(m^2)$ comparisons, which can also handle key indexed data. Hamada et al. proposed a method for converting sorting algorithms into corresponding sorting protocols [18]. Their quicksort protocol exhibits $O(\log m)$ rounds and $O(m \log m)$ comparisons on average. However, $O(\log m)$ communication overhead is required to resolve the case in which the input values include duplications.

Jónsson et al. studied a general method to hide the number of input values for sorting protocols [23]. Goodrich and Mitzenmacher proposed a method to extend internal-memory sorting algorithms to external-memory sort [16].

As for implementations, Jónsson et al. [23] implemented Batcher's merge sort and other sorting protocols on the Sharemind MPC system [7]. Their implementation is optimized by using a technique called vectorization, and the vectorized Batcher's merge sort sorts $16,384$ secret-shared values in 197 seconds. Hamada et al. [18] implemented their quicksort protocol using a $(2,3)$-Shamir's secret-sharing scheme with a corruption tolerance of 1 [31]. It sorts 1 million 32-bit word secret-shared values in $1,227$ seconds. Thus, the sorting operation in MPC schemes is still expensive, and improving efficiency is an important issue to address.

2 Preliminaries

We describe here the notations and definitions used in this paper, and fundamental protocols used in our protocol.

Let $a := b$ denote that a is defined by b, $a||b$ denote the concatenation of a and b, and $\mathscr{G}$, $\mathscr{R}$, $\mathscr{F}$, $\mathbb{Z}$, and $\mathbb{Z}_2$ be a group, a ring, a field, the set of integers, and $\mathbb{Z}/2\mathbb{Z}$, respectively. If a is an ℓ-bit element, $a^{(i)}$ denotes the i-th bit of a, where we count the indices in the right-to-left order with 1 being the initial index, i.e., $a := a^{(\ell)}||\cdots||a^{(1)}$. If A is a probabilistic algorithm, $a \leftarrow \mathsf{A}(b)$ means a is an output of A on input b. If A is a set, $|A|$ denotes the required bits to represent an element in A.

2.1 Setting and security model

We assume a setting of three servers, of which at most one server might be passively corrupt. This is the same setting and security model as in [3].

We consider secret-sharing (SS)-based three-party computation secure against a single static corruption: There are three parties P_1, P_2, P_3, a secret is shared among these parties via SS, any two parties can reconstruct the secret from their shares, and an adversary corrupts up to a single party at the beginning of the protocol.

For notational simplicity, when we use an index to denote the i-th party, $i-1$ and $i+1$ refer to the previous and subsequent party. There are three parties so we define the party following P_3 as P_1. For example, P_{i+1} means P_3 if $i=2$, P_{i+1} means P_1 if $i=3$, and P_{i-1} means P_3 if $i=1$.

We consider the client/server model. This model is used to outsource secure computation, where any number of clients send shares of their inputs to the servers. Therefore, both the input and output of the servers are shares, and both of our protocols are therefore share-input and share-output protocols.

Regarding adversarial behavior, we consider a passive (semi-honest) adversary: In passive security, corrupted parties follow the protocol but might try to obtain private information from the transcripts of messages that they receive. Formally, we say that a protocol is passively secure if there is a simulator that simulates the view of the corrupted parties from the inputs and outputs of the protocol [14].

We prove the security of our protocols in a hybrid model, where parties run a protocol with real messages and also have access to a trusted party computing a subfunctionality for them. When the subfunctionality is g, we say that the protocol works in the g-hybrid model.

2.2 Secret sharing

We use linear SS in two rings: one is an arbitrary ring $\mathscr{R}$ (for which we denote a share x as $[x]$), whereas the other is another ring $\mathscr{R}'$ that is sufficiently large to contain an order of items (for which we denote a share x as $[\![x]\!]$). The input of our protocols consists of shares in $\mathscr{R}$ but some shares are converted into those in $\mathscr{R}'$ in the internal processing.

Linear secret sharing scheme. We use a linear SS scheme satisfying the following properties. A concrete example is Shamir's scheme [31] and the replicated SS scheme [22, 11].

- Share and ShareSim: On input $a \in \mathscr{R}$, this algorithm outputs shares of a. This algorithm is denoted by $[a] \leftarrow \mathsf{Share}(a)$, where $[a]_i$ denotes P_i's share, and $[a]$ denotes a tuple of all shares, $([a]_1, [a]_2, [a]_3)$. Let $[\overrightarrow{a}]$ (resp. $[\overrightarrow{a}]_i$) denote a vector of shares $([a_1], \ldots, [a_m])$ (resp. $([a_1]_i, \ldots, [a_m]_i)$). In addition, there is an algorithm $([a]_{i+1}, [a]_{i-1}) \leftarrow \mathsf{ShareSim}([a]_i, a)$, which computes the other shares from a single share and the secret.
- Reveal: Revealing is the protocol that on input a pair of shares, outputs a secret. For simplicity, we consider a simple revealing in which each P_i sends $[a]_i$ to P_{i+1} and then reconstructs a from $[a]_i$ and $[a]_{i-1}$.
- Local operations: Given shares $[a]$, $[b]$, and a scalar $\alpha \in \mathscr{R}$ the parties can generate shares of $[a+b]$, $[\alpha a]$, $[\alpha + a]$, and $[-a]$ using only local operations. The notations $[a]+[b]$, $\alpha[a]$, $\alpha+[a]$, and $-[a]$ denote those local operations, respectively.
- LocalAdditive: This is an algorithm that on input $[a]_i$ and $[a]_{i+1}$ for P_i and P_{i+1}, respectively, P_i and P_{i+1} individually generates β_i and β_{i+1} such that $a = \beta_i + \beta_{i+1}$.
- Reshare: This is a protocol that "re-randomizes" a share by *two* parties. The protocol is denoted by $[a] \leftarrow \textsc{Reshare}([a]_i, [a]_{i+1})$. Given a share $[a]_i$ for P_i and $[a]_{i+1}$ for P_{i+1}, this protocol guarantees that at the end of the execution, each party P_i has a renewed share $[a]_i$ for $i \in \{1, 2, 3\}$, where $[a]_{i-1}$ is uniformly random for P_{i-1}. A naïve resharing protocol is that P_i and P_{i+1} generate β_i and β_{i+1} via LocalAdditive, and secret-share them. The parties then add their shares to obtain a renewed share.

We specifically use the notation $[\![\cdot]\!]_i$ (resp. $[\![\cdot]\!]$) for a share in a restricted ring $\mathscr{R}'$. Informally, $\mathscr{R}'$ should satisfy that $|\mathscr{R}'| > \log m$, where m is the number of rows to be sorted. The above properties of a linear SS also hold for $[\![\cdot]\!]$.

Shamir's scheme In the three-party case, a secret $a \in \mathscr{F}$ lies on a polynomial $f(x) := a + rx$, where $r \leftarrow \mathscr{F}$, and each share is a coordinate of the polynomial. If $[\cdot]$ is a share of the Shamir's scheme, for any pair of parties (P_i, P_{i+1}), there exists the Lagrange coefficients $(\lambda_i^{(i,i+1)}, \lambda_i^{(i,i+1)})$ such that $a = \lambda_i^{(i,i+1)}[a]_i + \lambda_i^{(i,i+1)}[a]_{i+1}$. We will describe $\lambda_i^{(i,i+1)}$ as λ_i if it is obvious.

Replicated secret sharing scheme A replicated SS scheme [11, 22] is an SS scheme in a *group*. Let $\circ$ be a group operation. In the three-party case, a secret $\pi \in \mathscr{G}$ is divided into three sub-shares as $\pi = \pi_1 \circ \pi_2 \circ \pi_3$, and P_i has (π_i, π_{i+1}). Here, a single party P_i cannot obtain any information about π since P_i does not know π_{i-1}. In this paper, we use two groups for the replicated SS scheme. We use the notation $\langle\cdot\rangle_i$ (resp. $\langle\cdot\rangle$) for a share of $\mathbb{Z}_2$, and $\langle\!\langle \cdot \rangle\!\rangle_i$ (resp. $\langle\!\langle \cdot \rangle\!\rangle$) for a share of a permutation.

Pseudorandom secret sharing It is well known that the parties can generate shares of a random number *without interaction*. This is called pseudorandom secret sharing (PRSS) [11]. In PRSS, each pair of parties preliminary shares a key, i.e., P_i share key_i with P_{i-1} and key_{i+1} with P_{i+1}. When the parties compute shares of a random number α, each party P_i computes a pseudorandom function with their keys as $\alpha_i := \mathrm{Func}_{key_i}(ctr)$ and $\alpha_{i+1} := \mathrm{Func}_{key_{i+1}}(ctr)$ for some onetime counter ctr, and regard (α_i, α_{i+1}) as the share of a random number. Formally, PRSS securely computes the following functionality $\mathcal{F}_{\mathrm{rand}}$.

FUNCTIONALITY 2.1 ($\mathcal{F}_{\mathrm{rand}}$ – Generate shares of a random value)

When $\mathcal{F}_{\mathrm{rand}}$ is invoked by P_i for $1 \le i \le 3$, $\mathcal{F}_{\mathrm{rand}}$ randomly chooses $\alpha, \alpha_1, \alpha_2 \leftarrow \mathscr{G}$, chooses α_3 satisfying $\alpha = \alpha_1 \circ \alpha_2 \circ \alpha_3$, and sends (α_i, α_{i+1}) to P_i.

2.3 Permutation

The permutation

$$\sigma = \begin{pmatrix} 1 & 2 & \cdots & m \\ \sigma(1) & \sigma(2) & \cdots & \sigma(m) \end{pmatrix}$$

reorders $(1, 2, \ldots, m)$ into $(\sigma(1), \sigma(2), \ldots, \sigma(m))$. For example, if $\sigma(1) = 3$, $\sigma(2) = 4$, $\sigma(3) = 2$, and $\sigma(4) = 1$, an input (A, B, C, D) is reordered into (C, D, B, A) by σ.

For notational simplicity, $\sigma \cdot \overrightarrow{a}$ denotes applying σ to $\overrightarrow{a} = (a_1, \ldots, a_m)$, and the output is $(a'_1, \ldots, a'_m)$ such that $a'_i = a_{\sigma(i)}$. Similar to that, $[\![\sigma \cdot \overrightarrow{a}]\!]$ denotes that each P_i applies σ to $[\![\overrightarrow{a}]\!]_i = ([\![a_1]\!]_i, \ldots, [\![a_m]\!]_i)$, and obtains $([\![a'_1]\!]_i, \ldots, [\![a'_m]\!]_i)$ such that $a'_j = a_{\sigma(j)}$. $[\sigma \cdot \overrightarrow{a}]$ also denotes the same operation. Note that the application operation is right-associative. That is, $\pi \cdot \sigma \cdot \overrightarrow{a} = \pi \cdot (\sigma \cdot \overrightarrow{a})$ holds.

It is known that permutations form a non-abelian group. Therefore, one can compose two permutations, σ and π, as

$$\sigma \circ \pi = \begin{pmatrix} 1 & 2 & \cdots & m \\ \sigma(\pi(1)) & \sigma(\pi(2)) & \cdots & \sigma(\pi(m)) \end{pmatrix},$$

and there exists the inverse for any permutation. In fact, it holds that

$$\sigma^{-1} = \begin{pmatrix} \sigma(1) & \sigma(2) & \cdots & \sigma(m) \\ 1 & 2 & \cdots & m \end{pmatrix}.$$

Here, to compute $\sigma^{-1} \cdot \overrightarrow{a}$, $\sigma(i)$ can be regarded as the *destination* of i, i.e., the i-th item is moved by σ^{-1} to be the $\sigma(i)$-th item. Therefore, if $\sigma(1) = 3$, $\sigma(2) = 4$, $\sigma(3) = 2$, and $\sigma(4) = 1$, an input (A, B, C, D) is reordered into (D, C, A, B) by σ^{-1}.

In the context of secure computation, there are several representations to secretly share a permutation σ among the parties. In this paper, we use two representations: share-vector and replicated representations. The share-vector representation of σ, denoted by $[\![\overrightarrow{\sigma}]\!]$, is

$$[\![\overrightarrow{\sigma}]\!] = ([\![\sigma(1)]\!], \ldots, [\![\sigma(m)]\!]),$$

where each party P_i has $[\![\overrightarrow{\sigma}]\!]_i := ([\![\sigma(1)]\!]_i, \ldots, [\![\sigma(m)]\!]_i)$ and the parties can obtain σ by reconstructing $(\sigma(1), \ldots, \sigma(m))$. Alternatively, the replicated representation of π, denoted by $\langle\!\langle \pi \rangle\!\rangle$, shares π via replicated SS in the following way: π is shared as $\pi = \pi_1 \circ \pi_2 \circ \pi_3$ for random permutations π_1 and π_2, and P_i has $\langle\!\langle \pi \rangle\!\rangle_i := (\pi_i, \pi_{i+1})$. $\mathcal{F}_{\mathrm{rand}}$ can generate $\langle\!\langle \pi \rangle\!\rangle$ for a random permutation π since this is in fact shares of a random number in the replicated SS. In both representations, a permutation can be reconstructed by two parties while a single corrupted party cannot obtain information about the permutation.

We show two observations that are useful to see the correctness of our protocols.

Observation 2.2 *Let π and ρ be permutations of order n. Let $\overrightarrow{v}$ be a vector of length n. Then $\rho \cdot (\pi \cdot \overrightarrow{v}) = (\pi \circ \rho) \cdot \overrightarrow{v}$.*

Proof. Let $\overrightarrow{x} = \pi \cdot \overrightarrow{v}$ and $\overrightarrow{y} = \rho \cdot \overrightarrow{x} = \rho \cdot (\pi \cdot \overrightarrow{v})$. By the definition, $x_i = v_{\pi(i)}$ and $y_i = x_{\rho(i)} = v_{\pi(\rho(i))} = v_{(\pi \circ \rho)(i)}$ for $1 \leq i \leq n$. Therefore, by the definition of application, we have $\overrightarrow{y} = (\pi \circ \rho) \cdot \overrightarrow{v}$. Consequently, we have $\rho \cdot (\pi \cdot \overrightarrow{v}) = (\pi \circ \rho) \cdot \overrightarrow{v}$.

Observation 2.3 *Let ρ, σ and π be permutations of order n. Let $\overrightarrow{\rho} = (\rho(1), \ldots, \rho(n))$ and $\overrightarrow{\sigma} = (\sigma(1), \ldots, \sigma(n))$. If $\overrightarrow{\rho} = \pi \cdot \overrightarrow{\sigma}$ then $\rho = \sigma \circ \pi$.*

Proof. By the definition of application and $\overrightarrow{\rho} = \pi \cdot \overrightarrow{\sigma}$, we have $\rho_i = \sigma_{\pi(i)}$ for $1 \leq i \leq n$. By the definitions of $\overrightarrow{\rho}$ and $\overrightarrow{\sigma}$, we have $\rho_i = \rho(i)$ and $\sigma_i = \sigma(i)$ for $1 \leq i \leq n$. Therefore $\rho(i) = \rho_i = \sigma_{\pi(i)} = \sigma(\pi(i)) = (\sigma \circ \pi)(i)$ holds. Thus, we have $\rho = \sigma \circ \pi$.

3 Component protocols

In this section we specify the fundamental protocols used as the components of our new sorting protocol. Note that if the input of a protocol is shares of a (general) linear SS scheme ($[\cdot]$) then the protocol also accepts shares of a linear SS ($[\![\cdot]\!]$) and the replicated SS scheme in $\mathbb{Z}_2$ ($\langle\cdot\rangle$).

3.1 Multiplication

This is a (passively secure) protocol that on input $[a]$ and $[b]$ outputs $[ab]$. The functionality of the multiplication protocol appears in Functionality 3.1. There are known protocol that securely compute $\mathcal{F}_{\text{mult}}$ [13, 12].

FUNCTIONALITY 3.1 ($\mathcal{F}_{\text{mult}}$ – Multiplication)

Upon receiving $([a], [b])$ from P_i for $1 \leq i \leq 3$, $\mathcal{F}_{\text{mult}}$ reconstructs (a, b), computes $c = ab$, obtain shares $[c] \leftarrow \mathsf{Share}(c)$, and sends $[c]_i$ to P_i.

3.2 Bit decomposition and modulus conversion

The bit-decomposition protocol decomposes a shared secret from shares of an integer to shares of its bits. We use a bit-decomposition protocol that on an input consisting of shares of a linear SS, $[a]$, outputs shares of the replicated SS in $\mathbb{Z}_2$, $(\langle a^{(1)}\rangle, \ldots \langle a^{(\ell)}\rangle)$, where $a = a^{(\ell)}||\cdots||a^{(1)}$. We give the functionality of the bit-decomposition protocol in Functionality 3.2. If the ring $\mathscr{R}$ of a linear SS is $\mathbb{Z}_p$ for a prime number p, a protocol that securely computes $\mathcal{F}_{\text{bitdecomp}}$ appears in [24].

The modulus-conversion protocol changes the underlying group or ring while maintaining the secret. We use a specific case of a modulus-conversion protocol in which shares in $\mathbb{Z}_2$ are converted into shares in $\mathscr{R}'$. We give the functionality of the modulus-conversion protocol in Functionality 3.3. A protocol that securely computes $\mathcal{F}_{\text{modconv}}$ appears in [24].

FUNCTIONALITY 3.2 ($\mathcal{F}_{\text{bitdecomp}}^{[\cdot],\langle\cdot\rangle}$ – Bit decomposition)

Upon receiving $[a]$, $\mathcal{F}_{\text{bitdecomp}}$ reconstructs a, generates shares $\langle\!\langle\, a^{(1)} \,\rangle\!\rangle, \ldots, \langle\!\langle\, a^{(\ell)} \,\rangle\!\rangle$, and sends $\langle a^{(1)}\rangle_i, \ldots, \langle a^{(\ell)}\rangle_i$ to P_i

FUNCTIONALITY 3.3 ($\mathcal{F}_{\text{modconv}}^{\langle\cdot\rangle,[\![\cdot]\!]}$ – Modulus conversion)

Upon receiving $\langle a\rangle$, $\mathcal{F}_{\text{modconv}}$ reconstructs a, generates shares $[\![a]\!]$ whose shares are in $\mathscr{R}'$, and sends $[\![a]\!]_i$ to P_i

3.3 Shuffling and Unshuffling

We describe in Algorithm 1 the shuffling protocol proposed by Laur et al. [28]. The protocol is defined with the shuffling permutation given as an input (which will be equal to a random permutation), rather than generated in the protocol. This is because we apply the same (unknown) permutation to multiple vectors.

The parties prepare before the protocol a random permutation $\langle\!\langle \pi \rangle\!\rangle$ by using $\mathcal{F}_{\mathrm{rand}}$, where $\pi = \pi_1 \circ \pi_2 \circ \pi_3$ and P_i has (π_i, π_{i+1}). In the shuffling protocol, on inputs $[\overrightarrow{a}]$ and $\langle\!\langle \pi \rangle\!\rangle$, any pair of parties, P_i and P_{i+1}, applies the permutation π_{i+1} to their shares, $[\overrightarrow{a}]_i$ and $[\overrightarrow{a}]_{i+1}$, and then reshares all the shares for $1 \leq i \leq 3$. As a result, the parties obtain $[\pi_3 \cdot \pi_2 \cdot \pi_1 \cdot \overrightarrow{a}] = [\pi \cdot \overrightarrow{a}]$. Each party cannot know π since the shares are reshared in each step of applying π_i.

Algorithm 1 Shuffling protocol

Notation: $[\overrightarrow{a}'] \leftarrow \textsc{Shuffle}(\langle\!\langle \pi \rangle\!\rangle; [\overrightarrow{a}])$.
Input: A secret-shared vector $[\overrightarrow{a}]$ and permutation $\langle\!\langle \pi \rangle\!\rangle$.
Output: The secret-shared shuffled vector $[\overrightarrow{a}'] = [\pi \cdot \overrightarrow{a}]$.
1: Let $\langle\!\langle \pi \rangle\!\rangle_i = (\pi_i, \pi_{i+1})$.
2: **for** $i = 1$ **to** 3 **do**
3: P_{i-1} and P_i compute $[\overrightarrow{a'}]_{i-1} := [\pi_i \cdot \overrightarrow{a}]_{i-1}$ and $[\overrightarrow{a'}]_i := [\pi_i \cdot \overrightarrow{a}]_i$, respectively.
4: $[\overrightarrow{a'}] \leftarrow \textsc{Reshare}([\overrightarrow{a'}]_{i-1}, [\overrightarrow{a'}]_i)$.
5: The parties set $[\overrightarrow{a}] := [\overrightarrow{a'}]$.
6: **return** $[\overrightarrow{a}]$.

Laud [26] extended Shuffle to support *unshuffle*, which applies the reverse of the random permutation to a given vector. We describe the unshuffling protocol in Algorithm 2.

Algorithm 2 Unshuffling protocol

Notation: $[\overrightarrow{a}'] \leftarrow \textsc{Unshuffle}(\langle\!\langle \pi \rangle\!\rangle; [\overrightarrow{a}])$.
Input: A secret-shared vector $[\overrightarrow{a}]$ and permutation $\langle\!\langle \pi \rangle\!\rangle$
Output: The secret-shared shuffled vector $[\overrightarrow{a}'] = [\pi^{-1} \cdot \overrightarrow{a}]$.
1: Let $\langle\!\langle \pi \rangle\!\rangle_i = (\pi_i, \pi_{i+1})$.
2: **for** $i = 3$ **to** 1 (in the descent order) **do**
3: P_{i-1} and P_i compute $[\overrightarrow{a'}]_{i-1} := [\pi_i^{-1} \cdot \overrightarrow{a}]_{i-1}$ and $[\overrightarrow{a'}]_i := [\pi_i^{-1} \cdot \overrightarrow{a}]_i$, respectively.
4: $[\overrightarrow{a'}] \leftarrow \textsc{Reshare}([\overrightarrow{a'}]_{i-1}, [\overrightarrow{a'}]_i)$.
5: The parties set $[\overrightarrow{a}] := [\overrightarrow{a'}]$.
6: **return** $[\overrightarrow{a}]$.

4 The Secure Sorting protocol

4.1 Setting

The input to the sorting protocol consists of shares of a key and of a value. Without loss of generality, we assume that each item consists of a single key and a single value. Denote the bit-length of the key as ℓ, the bit-length of the value as ℓ', and let $[\overrightarrow{k}]$ and $[\overrightarrow{v}]$ denote the key and value columns of the input. Here, $[\overrightarrow{v}]$ can be the same as $[\overrightarrow{k}]$.

The sorting protocol implements a stable sort: it rearranges the order of the items based on the keys, and maintains the relative order of items that have equal keys. In other words, the protocol outputs $[\overrightarrow{v}'] = ([v'_1], \ldots, [v'_m])$ satisfying the following condition. Let σ be the permutation that satisfies $\overrightarrow{v}' = \sigma \cdot \overrightarrow{v}$, and $\overrightarrow{k}' := \sigma \cdot \overrightarrow{k}$. It holds that $k_i \leq k_{i+1}$, and if $k_i = k_j$, then $\sigma^{-1}(i) < \sigma^{-1}(j)$ only when $i < j$.

4.2 Asymptotic improvement over previous work

Bogdanov et al. proposed a sorting protocol [6] which is based on radix sort. We describe the protocol in Section 4.3. In a nutshell, that protocol works by repeatedly sorting the items, where in each step the items are ordered according to a bit of the keys, starting from the least significant bit. For each bit, the parties learn shares of a permutation that sorts the values according to that bit, and then run a protocol which permutes the shared items using that shared permutation. (This protocol requires composing the permutation with a random shuffle permutation and then opening the result.) Note that for each bit it is required to permute and shuffle all items in their full length (including the key and the value).

Our protocol, on the other hand, uses a new protocol component that computes the composition of shared permutations. This enables us to compute the composition of the permutations which sort according to each bit of the keys. Consequently, there is no need to permute the full values in the bit-sorting steps of the protocol, but rather only the keys. As a result, we need much less communication (since these permutations constitute the bulk of the communication), and achieve much better performance.

4.3 Existing secure stable sort from radix sort

The protocol of Bogdanov et al. [6] (implicitly) uses two component protocols: GenBitPerm and ApplyInv.

GenBitPerm, described in Algorithm 3, is a protocol that computes *the inverse* of the permutation of a stable sort *for a single bit*. We say that this protocol outputs the inverse since it computes the *destinations* for the input vector. For example, if an input is $[\![\overrightarrow{a}]\!] = ([\![1]\!], [\![1]\!], [\![0]\!])$, the output of GenBitPerm is $[\![\overrightarrow{\rho}]\!] = ([\![2]\!], [\![3]\!], [\![1]\!])$, and the values satisfy $(0, 1, 1) = \rho^{-1} \cdot \overrightarrow{a}$.

Intuitively, GenBitPerm proceeds as follows. If an input is $[\![\overrightarrow{a}]\!] = ([\![1]\!], [\![1]\!], [\![0]\!])$, the parties compute $f_i^{(0)}$ and $f_i^{(1)}$, which represent whether or not the i-th secret is 0 and 1, respectively. By concatenating them, the parties obtain $[\![\overrightarrow{f}]\!] := ([\![0]\!], [\![0]\!], [\![1]\!], [\![1]\!], [\![1]\!], [\![0]\!])$, where the former three elements are the flags $f_i^{(0)}$. For example, the second element of the input is 1 so the second and fifth elements of $[\![\overrightarrow{f}]\!]$ are 0 and 1, respectively. The parties then count the concatenated flags from the initial element and obtain $[\![\overrightarrow{s}]\!] := ([\![0]\!], [\![0]\!], [\![1]\!], [\![2]\!], [\![3]\!], [\![3]\!])$. By multiplying $[\![\overrightarrow{f}]\!]$ with $[\![\overrightarrow{s}]\!]$, we obtain $[\![\overrightarrow{t}]\!] := ([\![0]\!], [\![0]\!], [\![1]\!], [\![2]\!], [\![3]\!], [\![0]\!])$, where the non-zero values are destinations of corresponding shares. The parties finally obtain the destination of i-th element of the input by adding the i-th and $(i+3)$-th elements of $[\![\overrightarrow{t}]\!]$.

Algorithm 3 Generating permutation of stable sort for a single bit

Notation: $[\![\overrightarrow{\rho}]\!] \leftarrow$ GenBitPerm$([\![\overrightarrow{k}]\!])$.
Input: Secret-shared bit-wise keys $[\![\overrightarrow{k}]\!]$, where $\overrightarrow{k} = (k_1, \ldots, k_m)$ and $k_i \in \{0, 1\}$ for $1 \leq i \leq m$.
Output: The secret-shared permutation $[\![\overrightarrow{\rho}]\!]$ such that ρ^{-1} is the stable sorting by $\overrightarrow{k}$.

for $1 \leq i \leq m$ **do**
  $[\![f_i^{(0)}]\!] := 1 - [\![k_i]\!]$.
  $[\![f_i^{(1)}]\!] := [\![k_i]\!]$.
$[\![s]\!] := [\![0]\!]$.
for $j = 0$ **to** 1 **do**
  for $i = 1$ **to** m **do**
    $[\![s]\!] := [\![s]\!] + [\![f_i^{(j)}]\!]$.
    $[\![s_i^{(j)}]\!] := [\![s]\!]$.
for $1 \leq i \leq m$ **do**
  The parties send $([\![f_i^{(0)}]\!], [\![s_i^{(0)}]\!])$ and $([\![f_i^{(1)}]\!], [\![s_i^{(1)}]\!])$ to $\mathcal{F}_{\text{mult}}$, and receive $[\![t_0]\!]$ and $[\![t_1]\!]$.
  $[\![\rho(i)]\!] := [\![t_0]\!] + [\![t_1]\!]$.
return $[\![\overrightarrow{\rho}]\!] = ([\![\rho(1)]\!], \ldots, [\![\rho(m)]\!])$.

The size of the ring $\mathscr{R}'$ for a linear SS $[\![\cdot]\!]$ at least $\lceil \log m \rceil$ since $s \leq m$. Therefore, the communication complexity of GenBitPerm is $O(m \log m)$ bits if we regard the complexity of an instance of $\mathcal{F}_{\text{mult}}$ as $O(\log m)$.

ApplyInv, described in Algorithm 4, is a protocol that applies the inverse of a shared permutation to a shared-vector representation. To simplify the description, Algorithm 4 assumes that the input is a single vector (if one wants to apply ρ^{-1} to multiple vectors, the parties shuffle the vectors in Step 2, and apply $(\rho'')^{-1}$ to each vector in Step 5). Intuitively speaking, if $\overrightarrow{\rho}$ is revealed, the parties can apply ρ^{-1} to $[\overrightarrow{k}]$ in the clear. However, we cannot do that since $\overrightarrow{\rho}$ discloses the permuted order of $\overrightarrow{k}$. Therefore, the parties first shuffle $[\overrightarrow{k}]$ and $[\![\overrightarrow{\rho}]\!]$ by using the *same permutation*, and reveal a shuffled $\overrightarrow{\rho}$. Even though $[\overrightarrow{k}]$ and $[\![\overrightarrow{\rho}]\!]$ are shuffled, the revealed values are the destinations of the corresponding items of $[\overrightarrow{k}]$, and the parties can therefore compute $[\rho^{-1} \cdot \overrightarrow{k}]$. Regarding security, an adversary can only obtain a shuffled $\overrightarrow{\rho}$, i.e., $\rho \circ \pi$, which is just a random permutation.

Algorithm 4 Applying the inverse of a share-vector permutation

Notation: $[\![\overrightarrow{k}']\!] \leftarrow \text{ApplyInv}([\![\overrightarrow{\rho}]\!]; [\overrightarrow{k}])$.
Input: A secret-shared permutation $[\![\overrightarrow{\rho}]\!]$ and a secret-shared vector $[\overrightarrow{k}] = ([k_1], \ldots, [k_m])$.
Output: The secret-shared vector $[\![k']\!]$ such that $\overrightarrow{k}' = \rho^{-1} \cdot \overrightarrow{k}$.
1: The parties call $\mathcal{F}_{\text{rand}}$ and receive $\langle\!\langle \pi \rangle\!\rangle$.
2: $[\![\overrightarrow{\rho}'']\!] \leftarrow \text{Shuffle}(\langle\!\langle \pi \rangle\!\rangle; [\![\overrightarrow{\rho}]\!])$.
3: The parties reveal $[\![\overrightarrow{\rho}'']\!]$ and obtain ρ''.
4: $[\overrightarrow{k}''] \leftarrow \text{Shuffle}(\langle\!\langle \pi \rangle\!\rangle; [\overrightarrow{k}])$.
5: The parties apply $(\rho'')^{-1}$ with $[\overrightarrow{k}'']$ and obtain $[\overrightarrow{k}']$.
6: **return** $[\overrightarrow{k}']$.

The sorting protocol in [6] uses GenBitPerm and ApplyInv. It assumes that a key is shared in a bit-wise fashion, i.e., $([\![\overrightarrow{k}^{(1)}]\!], \ldots, [\![\overrightarrow{k}^{(\ell)}]\!])$ where $[\![\overrightarrow{k}^{(j)}]\!] = ([\![k_1^{(j)}]\!], \ldots, [\![k_m^{(j)}]\!])$ and $k_i = k_i^{(\ell)} || \cdots || k_i^{(1)}$ for $1 \leq i \leq m$ and $1 \leq j \leq \ell$. Informally, the secure sorting protocol based on the radix sort in [6] is as follows:

For $1 \leq j \leq \ell$,

1. The parties obtain the stable sort $[\![\overrightarrow{\rho_j}]\!]$ of the j-th bit by using GenBitPerm with input $[\![\overrightarrow{k}^{(j)}]\!]$.
2. The parties apply ρ_j^{-1} to the value and the remaining bits of the key, $[\![\overrightarrow{k}^{(j+1)}]\!], \ldots, [\![\overrightarrow{k}^{(\ell)}]\!]$, and $[\![\overrightarrow{v}]\!]$, by using ApplyInv.[4]
3. The parties go back to (1) with $j := j + 1$.

The communication complexity of this protocol is $O(\ell m(\log m + \ell + \ell'))$ bits since in Step 2 the parties reshare all the elements, and this requires communication of $O(m(\log m + \ell + \ell'))$ bits in each step (of sorting by one of the bits). Although this protocol securely computes a radix sort, Bogdanov et al. empirically concluded that this protocol was less efficient than other secure sorting protocols, such as a secure quicksort [6].

4.4 Our sorting protocol

We point out that (2) of the sorting protocol of [6] (applying ρ_j^{-1} to all items) incurs a large amount of communication. We avoid this step in our protocol by separating the generation of the permutation from sorting the items, and instead computing a *composition* of the relevant permutations.

[4] In [6] the protocol is described as sorting only the key, rather than a key and an associated value.

In our protocol the parties have a "current" permutation $[\![\overrightarrow{\sigma_j}]\!]$ which is the composition of the permutations as $\sigma_j := \rho_j \circ \cdots \circ \rho_1$.[5] In other words, (the inverse of) σ_j sorts the lower significant j bits. When the parties obtain $[\![\overrightarrow{\rho}_{j+1}]\!]$, they compose it with the current temporary permutation to obtain $[\![\overrightarrow{\sigma_{j+1}}]\!] := [\![\overrightarrow{\rho_{j+1} \circ \sigma_j}]\!]$. After the parties compute the stable sort of all bits of the key, they apply (the inverse of) σ to the items. Consequently, in the j-th iteration it is not required to apply ρ_j^{-1} to the values of the items and to the most significant $\ell - j$ bits of the keys. This reduces the communication complexity significantly.

Composition of permutations We describe how to compose $[\![\overrightarrow{\sigma}]\!]$ and $[\![\overrightarrow{\rho}]\!]$ by using Shuffle and Unshuffle. We can compose two permutations that are shared by the share-vector representation as described in Algorithm 5. One can confirm the completeness as

$$\sigma' = \rho' \circ \pi^{-1} = (\rho \circ \sigma'') \circ \pi^{-1} = \rho \circ (\sigma \circ \pi) \circ \pi^{-1} = \rho \circ \sigma.$$

Algorithm 5 Composition of two share-vector permutations

Notation: $[\![\overrightarrow{\sigma'}]\!] \leftarrow \textsc{Compose}([\![\overrightarrow{\sigma}]\!], [\![\overrightarrow{\rho}]\!])$.
Input: Secret-shared two permutations $([\![\overrightarrow{\sigma}]\!], [\![\overrightarrow{\rho}]\!])$.
Output: The secret-shared permutation $[\![\overrightarrow{\sigma'}]\!]$, where $\sigma'^{-1} = \sigma^{-1} \circ \rho^{-1}$.
1: The parties call $\mathcal{F}_{\text{rand}}$ and obtain $\langle\!\langle \pi \rangle\!\rangle$.
2: $[\![\overrightarrow{\sigma}'']\!] \leftarrow \textsc{Shuffle}(\langle\!\langle \pi \rangle\!\rangle; [\![\overrightarrow{\sigma}]\!])$.
3: The parties reveal $[\![\overrightarrow{\sigma}'']\!]$ and obtain σ''.
4: The parties apply σ'' to $[\![\overrightarrow{\rho}]\!]$ and obtain $[\![\overrightarrow{\rho'}]\!]$.
5: $[\![\overrightarrow{\sigma}']\!] \leftarrow \textsc{Unshuffle}(\langle\!\langle \pi \rangle\!\rangle; [\![\overrightarrow{\rho}']\!])$.
6: **return** $[\![\overrightarrow{\sigma}']\!]$.

Generating the permutation of stable sort We describe in Algorithm 6 how to obtain the permutation of stable sort. The input of this protocol is shares of integers in a linear SS scheme, and we therefore convert them into shares of another linear SS scheme in a sufficiently large ring, i.e., $|\mathscr{R}'| > \lceil \log m \rceil$, by using $\mathcal{F}_{\text{bitdecomp}}$ and $\mathcal{F}_{\text{modconv}}$.

Each permutation $[\![\overrightarrow{\sigma}_j]\!]$ is a temporary permutation whose inverse sorts the least significant j bits. To obtain $[\![\overrightarrow{\sigma}_j]\!]$ from $[\![\overrightarrow{k^{(j)}}]\!]$ and $[\![\overrightarrow{\sigma}_{j-1}]\!]$, the parties compute $\sigma_{j-1}^{-1} \cdot \overrightarrow{k^{(j)}}$ (that is, the j-th bit of the keys reordered by the permutation that sorts the least significant $j-1$ bits), compute ρ_j, and compose σ_{j-1} and ρ_j to obtain σ_j. Note that since GenBitPerm outputs the destination of the items, $\sigma^{-1} \cdot \overrightarrow{k}$ is the sorted vector.

Putting it all together To sort the values $[\overrightarrow{v}]$ with the keys $[\overrightarrow{k}]$, the parties compute the permutation of the stable sort by using Algorithm 6 as $[\![\overrightarrow{\sigma}]\!] \leftarrow \textsc{GenPerm}([\overrightarrow{k}])$, and then compute the sorted values by using Algorithm 4 as $[\overrightarrow{v}'] \leftarrow \textsc{ApplyInv}([\![\overrightarrow{\sigma}]\!]; [\overrightarrow{v}])$.

Note, by using similar technique, we can "unsort" a vector of shares, which restores the original order of a vector from the sorted vector. This "unsorting" may be useful for some applications using sorting, and the protocol appears in Appendix B.

[5] GenBitPerm outputs $[\![\overrightarrow{\rho}_i]\!]$ such that ρ_i^{-1} is the stable sort of the i-th bit. Therefore, the stable sort of the all bits, σ_j^{-1}, satisfies $\sigma_j^{-1} := \rho_1^{-1} \circ \cdots \circ \rho_j^{-1}$, which is the same as $\sigma_j := \rho_j \circ \cdots \circ \rho_1$.

Algorithm 6 Generating permutation of stable sort

Notation: $[\![\overrightarrow{\sigma}]\!] \leftarrow \textsc{GenPerm}([\overrightarrow{k}])$.
Input: Secret-shared keys $[\overrightarrow{k}] = ([k_1], \ldots, [k_m])$.
Output: The secret-shared permutation $[\![\overrightarrow{\sigma}]\!]$ such that σ^{-1} is the stable sorting of $\overrightarrow{k}$.

1: If $[\overrightarrow{k}]$ is not shares of the linear SS scheme in $\mathscr{R}'$, the parties call $\mathcal{F}_{\text{bitdecomp}}$ and $\mathcal{F}_{\text{modconv}}$, and obtain bit-wise shares in $\mathscr{R}'$: $([\![\overrightarrow{k}^{(1)}]\!], \ldots, [\![\overrightarrow{k}^{(\ell)}]\!])$.
2: $[\![\overrightarrow{\rho}_1]\!] \leftarrow \textsc{GenBitPerm}([\![\overrightarrow{k}^{(1)}]\!])$.
3: $[\![\overrightarrow{\sigma}_1]\!] := [\![\overrightarrow{\rho}_1]\!]$.
4: **for** $j = 2$ **to** ℓ **do**
5: $\quad [\![\overrightarrow{k}'^{(j)}]\!] \leftarrow \textsc{ApplyInv}([\![\overrightarrow{\sigma}_{j-1}]\!]; [\![\overrightarrow{k}^{(j)}]\!])$.
6: $\quad [\![\overrightarrow{\rho}_j]\!] \leftarrow \textsc{GenBitPerm}([\![\overrightarrow{k}'^{(j)}]\!])$.
7: $\quad [\![\overrightarrow{\sigma}_j]\!] \leftarrow \textsc{Compose}([\![\overrightarrow{\sigma}_{j-1}]\!], [\![\overrightarrow{\rho}_j]\!])$.
8: **return** $[\![\overrightarrow{\sigma}_\ell]\!]$.

4.5 Communication complexity

We examine the communication complexity (in bits) of our sorting protocol. To obtain the exact numbers, we assume that Shamir's scheme is the underlying SS scheme of $[\cdot]$ and $[\![\cdot]\!]$,[6] $\mathcal{F}_{\text{mult}}$ is instantiated by [13], $\mathcal{F}_{\text{bitdecomp}}$ and $\mathcal{F}_{\text{modconv}}$ are instanciated by [24], and $\mathcal{F}_{\text{rand}}$ requires no communication by using PRSS. We count the sum of communicated bits of *all the parties* since several protocols, such as Reshare, is asymmetric protocol and it is hard to count that per party. For example, if each party sends $|\mathscr{R}'|$ bits to another, the communication complexity is $3|\mathscr{R}'|$. For simplicity, we say "m Shuffle for $[\![\cdot]\!]$" if the parties invoke Shuffle m times with (a vector of) shares in $\mathscr{R}'$.

The communication complexities of GenPerm to the key and ApplyInv to the value are as follows. The complexities of several steps are reduced to components protocols.

- GenPerm:
  1. The bit-decomposition [24] of $[\overrightarrow{k}]$ (an m-length vector of ℓ-bit key shares): $3(5\ell+2)m$ bits communication.
  2. The modulus-conversion [24] of the decomposed shares: $3(1+|\mathscr{R}'|)m$ bits.
  3. GenBitPerm: $2m$ $\mathcal{F}_{\text{mult}}$.
  4. $\ell - 1$ invocations of:
     - ApplyInv: Two Shuffle and m revealing.
     - GenBitPerm: $2m$ $\mathcal{F}_{\text{mult}}$.
     - Compose: single Shuffle, single Shuffle, and m revealing.
- ApplyInv to the value: single Shuffle for $\langle\!\langle \pi \rangle\!\rangle$, single Shuffle for the value shares, and m revealing.

In addition, the complexities of component protocols are as follows.

- Multiplication protocol [13]: $6|\mathscr{R}'|$ bits.
- Reveal: $3|\mathscr{R}'|$ bits.
- Reshare: $4|\mathscr{R}'|$ bits.
- Shuffle and Unshuffle for $[\![\cdot]\!]$: $12m|\mathscr{R}'|$ bits.
- Shuffle of for ℓ'-bit shares, i.e., the value shares: $12m\ell'$ bits.

Therefore, the total communication complexity is

$$\begin{aligned}
&3(5\ell+2)m + 3(1+|\mathscr{R}'|)m + 18m|\mathscr{R}'| \\
&+(\ell-1)\left((24+3)m|\mathscr{R}'| + 12m|\mathscr{R}'| + (24+3)m|\mathscr{R}'|\right) + 12m|\mathscr{R}'| + 12m\ell' \\
&= m\left((66\ell-33)|\mathscr{R}'| + 15\ell + 12\ell' + 9\right).
\end{aligned}$$

[6] Although recent results of high-throughput secure computation [3, 2, 10] used the replicated SS scheme, we chose Shamir's scheme. This is because efficient bit-decomposition and modulus-conversion protocols are known for Shamir's scheme, and small share-size is preferable to manage a large amount of data.

This is regarded as $O(\ell m \log m + m\ell')$ by setting $|\mathscr{R}'| = O(\log m)$. Note that the size of the database itself is already $m(\ell + \ell')$ bits, and therefore the communication of our protocol is close to optimal, especially when the length of the keys ℓ is much smaller than the size of the values ℓ'. This communication complexity is asymptotically better than that of all the previous protocols, as depicted in Table 1.

4.6 Security

In order to prove security, we first define the functionality of stable sorting. Our sorting protocol supports three functionalities: It can generate the permutation of a stable sort, and can sort a vector of shares by applying this permutation. Therefore, we define the functionality to support these two functions. The functionality $\mathcal{F}_{\text{sort}}$ is defined in Functionality 4.1. Algorithms GenPerm and ApplyInv correspond to **GenPerm** and **Sort** of Functionality 4.1.

Therefore, we claim the following theorem.

FUNCTIONALITY 4.1 ($\mathcal{F}_{\text{sort}}$ – Stable sorting)

GenPerm: Upon receiving $(\texttt{GenPerm}, [\overrightarrow{k}] = ([k_1], \ldots, [k_m]))$, $\mathcal{F}_{\text{sort}}$ reconstructs $\overrightarrow{k}$, computes σ such that $k'_j \leq k'_{j+1}$ for $\overrightarrow{k'} = \sigma^{-1} \cdot \overrightarrow{k}$, and $\sigma(j) < \sigma(j')$ if $k_j = k_{j'}$ and $j < j'$, generates $[\![\overrightarrow{\sigma}]\!]$, and sends $[\![\overrightarrow{\sigma}]\!]_i$ to P_i.

Sort: Upon receiving $(\texttt{Sort}, [\![\overrightarrow{\sigma}]\!], [\overrightarrow{v}] = ([v_1], \ldots, [v_m]))$, $\mathcal{F}_{\text{sort}}$ reconstructs $\overrightarrow{v}$ and σ, computes $\overrightarrow{v}' := \sigma^{-1} \cdot \overrightarrow{v}$, generates $[\overrightarrow{v}']$, and sends $[\overrightarrow{v}']_i$ to P_i.

Theorem 4.2. *(GenPerm, ApplyInv) securely compute $\mathcal{F}_{sort}$ in the $(\mathcal{F}_{rand}, \mathcal{F}_{bitdecomp}, \mathcal{F}_{modconv})$-hybrid model against a single corruption by a passive adversary.*

Proof. Algorithms GenPerm and ApplyInv use smaller algorithms as components. Three of these component algorithms, GenBitPerm, ApplyInv, and Shuffle, have already been proved in [28, 6] to be secure against a single corruption by a passive adversary, and the simulator can simulate the views of these protocols. The only remaining algorithm which is used in the constructions is Compose. Therefore, in order to fully simulate the view of our new we discuss here the security of Compose.

Note that the only step in Compose which does not use an existing algorithm which was already proved is Step 3, where the parties reveal the permutation σ''. Without loss of generality, we assume that P_1 is corrupted. Then all steps except for Step 3 can be simulated in a straightforward way. In step 3, the simulator generates a random permutation $\tilde{\sigma}''$, and simulates the received value, $[\![\overrightarrow{\sigma}'']\!]_3$, by using $\mathsf{ShareSim}$: The simulator computes $(\cdot, \widetilde{[\![\sigma''(i)]\!]_3}) \leftarrow \mathsf{ShareSim}([\![\sigma(i)]\!]_1, \tilde{\sigma}''(i))$ for $1 \leq i \leq m$, where $\widetilde{[\![\sigma''(i)]\!]_3}$ is the simulated value. Here, σ'' is a random permutation due to shuffling, and therefore the distributions of $\tilde{\sigma}''(i)$ and $\widetilde{[\![\sigma''(i)]\!]_3}$ are identical to those in the real execution. ∎

5 Improve concrete efficiency

In this section, we propose efficient multiplication, resharing, and shuffling protocols, and optimization techniques to improve concrete efficiency of our sorting protocol. These protocols are not only useful for the sorting protocol but also other protocols since they are fundamental protocols. Due to space limitation, the functionalities of those appear in Appendix F. In this section, we use $\mathsf{ShareSim}$ and $\mathsf{LocalAdditive}$ of a linear SS scheme.

5.1 Novel protocols

Optimized multiplication and sum-of-product for Shamir's scheme We propose an optimized multiplication protocol in which each party sends a single element to another per multiplication in $\mathcal{F}_{\text{rand}}$-hybrid model. If an input is shares in $\mathscr{R}'$, the communication complexity, $3\,|\mathscr{R}'|$, is a half of the one of the original protocol [13], $6\,|\mathscr{R}'|$, and the same as the one of multiplication protocols for the replicated SS scheme [3, 21, 10].

The optimized multiplication protocol appears in Algorithm 7. This protocol is the same as the original one except the step of secret-sharing $\widehat{\beta}_i$. Instead of generating shares honestly, each P_i sets $[\widehat{\beta}_i]_{i+1} := \alpha_{i+1}$, where α_{i+1} is a random value from $\mathcal{F}_{\text{rand}}$ and common in P_i and P_{i+1}.

Algorithm 7 Optimized multiplication protocol for Shamir's scheme

Notation: $[c] \leftarrow \textsc{OptMult}([a], [b])$.
Input: Two secret-shared values $([a], [b])$.
Output: A secret-shared value $([c])$, where $c = ab$.

1: Each P_i for $1 \le i \le 3$ computes $\widehat{\beta}_i := \widehat{\lambda}_i [a]_i [b]_i$, where $\widehat{\lambda}_i$ are Lagrange coefficients satisfying $ab = \sum_{i=1}^{3} \widehat{\lambda}_i [a]_i [b]_i$.
2: The parties call $\mathcal{F}_{\text{rand}}$ and receive (α_i, α_{i+1}) for P_i.
3: **for** $i = 1$ **to** 3 (in parallel) **do**
4: $\quad P_i$ and P_{i+1} set $[\widehat{\beta}_i]_{i+1} := \alpha_{i+1}$.
5: $\quad P_i$ computes $([\widehat{\beta}_i]_i, [\widehat{\beta}_i]_{i-1}) \leftarrow \mathsf{ShareSim}(\widehat{\beta}_i, [\widehat{\beta}_i]_{i+1})$.
6: $\quad P_i$ sends $[\widehat{\beta}_i]_{i-1}$ to P_{i-1}.
7: $[c] := [\widehat{\beta}_1] + [\widehat{\beta}_2] + [\widehat{\beta}_3]$.
8: **return** $[c]$.

Theorem 5.1. *Algorithm 7 securely computes $\mathcal{F}_{mult}$ in the $\mathcal{F}_{rand}$-hybrid model against a single corruption by a passive adversary.*

Proof. Without loss of generality, assume an adversary corrupts P_{i^*}. In the protocol, P_{i^*} receives $[\widehat{\beta}_{i^*+1}]_{i^*}$ only. The simulator randomly picks $(\widetilde{\alpha}_1, \widetilde{\alpha}_2, \widetilde{\alpha}_3)$ to simulate $\mathcal{F}_{\text{rand}}$, computes $[\widehat{\beta}_{i^*}]_{i^*}$ and $[\widehat{\beta}_{i^*-1}]_{i^*}$ by following the protocol, and computes the simulated value as $[c_i] - [\widehat{\beta}_{i^*}]_{i^*} - [\widehat{\beta}_{i^*-1}]_{i^*}$. ∎

The multiplication protocol can be extended to a sum-of-product protocol that on input $([a_1], \ldots, [a_m])$ and $([b_1], \ldots, [b_m])$ outputs $[c]$, where $c = \sum_{j=1}^{m} a_i b_i$ [9]. The functionality $\mathcal{F}_{\text{product}}$ and the sum-of-product protocol, OptProduct, appears in Appendix A. The communication and round complexities are the same as the multiplication protocol.

Optimized resharing We propose an optimized resharing protocol whose communication complexity is $2\,|\mathscr{R}'|$. It is a half of the naïve resharing protocol in Section 2.2.

The optimized resharing protocol appears in Algorithm 8. Here, an input of the optimized resharing protocol is not $[a]_i$ and $[a]_{i+1}$ but β_i and β_{i+1}, where $a = \beta_i + \beta_{i+1}$. We change the input syntax to later combine it with the optimized shuffling protocol. The parties invoke $\mathsf{LocalAdditive}$ before OptReshare to obtain a resharing protocol with an ordinary input.

Theorem 5.2. *Algorithm 8 securely computes $\mathcal{F}_{reshare}$ in the $\mathcal{F}_{rand}$-hybrid model against a single corruption by a passive adversary.*

Proof. $[\beta_i]_{i+1}$ and $[\beta_{i+1}]_i$ are generated by $\mathsf{ShareSim}$, where on of the input of $\mathsf{ShareSim}$ is a random element. Therefore, those elements are uniformly random and the simulator chooses random elements for the simulation. ∎

Algorithm 8 Optimized resharing protocol

Notation: $[a] \leftarrow \textsc{OptReshare}(\beta_i, \beta_{i+1})$.
Input: β_i and β_{i+1} for P_i and P_{i+1}, respectively, where $a = \beta_i + \beta_{i+1}$.
Output: A (re-randomized) secret-shared value $[a]$.
1: The parties call $\mathcal{F}_{\text{rand}}$ and receive (α_i, α_{i+1}) for P_i.
2: P_{i-1} and P_i set $[\beta_i]_{i-1} := \alpha_i$, and P_{i+1} and P_{i-1} set $[\beta_{i+1}]_{i-1} := \alpha_{i-1}$.
3: P_i computes $([\beta_i]_i, [\beta_i]_{i+1}) \leftarrow \mathsf{ShareSim}(\beta_i, [\beta_i]_{i-1})$.
4: P_{i+1} computes $([\beta_{i+1}]_{i+1}, [\beta_{i+1}]_i) \leftarrow \mathsf{ShareSim}(\beta_{i+1}, [\beta_{i+1}]_{i-1})$.
5: P_i and P_{i+1} send $[\beta_i]_{i+1}$ and $[\beta_{i+1}]_i$ to each other.
6: The parties compute $[a] := [\beta_i] + [\beta_{i+1}]$.
7: **return** $[a]$.

Optimized shuffling The shuffling protocol (Alg. 1) invokes the resharing protocol three times. Therefore, even if we use the optimized resharing protocol, the communication complexity is $6m\,|\mathscr{R}'|$. We show the optimized shuffling protocol in Algorithm 9 whose communication complexity is $4m\,|\mathscr{R}'|$ bits.

Recall that the shuffling protocol repeatedly permutes shares by π_i three times. The idea is that P_1 knows π_2 and π_1 that used for the first and second shuffling steps, and the randomness used in resharing also can be obtained via $\mathcal{F}_{\text{rand}}$ beforehand. Therefore, P_1 can permute $\overrightarrow{a}$ using π_2 and π_1 at once.

Algorithm 9 Optimized shuffling protocol

Notation: $[\overrightarrow{a}'] \leftarrow \textsc{OptShuffle}(\langle\!\langle\, \pi \,\rangle\!\rangle; [\overrightarrow{a}])$
Input: A secret-shared vector $[\overrightarrow{a}]$ and a permutation $\langle\!\langle\, \pi \,\rangle\!\rangle$.
Output: The secret-shared shuffled vector $[\overrightarrow{a}'] = [\pi\overrightarrow{a}]$.
1: Let $\langle\!\langle\, \pi \,\rangle\!\rangle_i = (\pi_i, \pi_{i+1})$.
2: The parties call $\mathcal{F}_{\text{rand}}$ m times and obtain $(\overrightarrow{\alpha}_i, \overrightarrow{\alpha}_{i+1})$ for P_i.
3: P_1 and P_3 compute $\overrightarrow{\beta}_1$ and $\overrightarrow{\beta}_3$ from $[\overrightarrow{a}]_1$ and $[\overrightarrow{a}]_3$, respectively, via $\mathsf{LocalAdditive}$.
4: P_3 computes $\overrightarrow{\gamma} := \pi_1 \cdot \overrightarrow{\beta}_3 + \overrightarrow{\alpha}_1$ and sends it to P_2.
5: P_1 computes $\overrightarrow{\delta} := \pi_2 \cdot (\pi_1 \cdot \overrightarrow{\beta}_1 - \overrightarrow{\alpha}_1) - \overrightarrow{\alpha}_2$ and sends it to P_3.
6: P_2 computes $\overrightarrow{\beta}'_2 := \pi_3 \cdot (\pi_2 \cdot \overrightarrow{\gamma} + \overrightarrow{\alpha}_2)$.
7: P_3 computes $\overrightarrow{\beta}'_3 := \pi_3 \cdot \overrightarrow{\delta}$.
8: $[\overrightarrow{a}'] \leftarrow \textsc{OptReshare}(\overrightarrow{\beta}'_2, \overrightarrow{\beta}'_3)$
9: **return** $[\overrightarrow{a}']$

First, let us confirm completeness. Regarding $\overrightarrow{\beta}'_2$ and $\overrightarrow{\beta}'_3$,

$$\begin{aligned}\overrightarrow{\beta}'_2 &= \pi_3 \cdot (\pi_2 \cdot \overrightarrow{\gamma} + \overrightarrow{\alpha}_2) = \pi_3 \cdot (\pi_2 \cdot (\pi_1 \cdot \overrightarrow{\beta}_3 + \overrightarrow{\alpha}_1) + \overrightarrow{\alpha}_2) \\ &= \pi_3 \cdot \pi_2 \cdot \pi_1 \cdot \overrightarrow{\beta}_3 + \pi_3 \cdot \pi_2 \cdot \overrightarrow{\alpha}_1 + \pi_3 \cdot \overrightarrow{\alpha}_2\end{aligned}$$

and

$$\begin{aligned}\overrightarrow{\beta}'_3 &= \pi_3 \cdot \overrightarrow{\delta} = \pi_3 \cdot (\pi_2 \cdot (\pi_1 \cdot \overrightarrow{\beta}_1 - \overrightarrow{\alpha}_1) - \overrightarrow{\alpha}_2) \\ &= \pi_3 \cdot \pi_2 \cdot \pi_1 \cdot \overrightarrow{\beta}_1 - \pi_3 \cdot \pi_2 \cdot \overrightarrow{\alpha}_1 - \pi_3 \cdot \overrightarrow{\alpha}_2.\end{aligned}$$

Therefore,

$$\overrightarrow{\beta}'_2 + \overrightarrow{\beta}'_3 = \pi_3 \cdot \pi_2 \cdot \pi_1(\overrightarrow{\beta}_1 + \overrightarrow{\beta}_3) = \pi \cdot \overrightarrow{a}.$$

It means that the optimized resharing protocol accepts $\overrightarrow{\beta}'_2$ for P_2 and $\overrightarrow{\beta}'_3$ for P_3 as an input.

Theorem 5.3. *Algorithm 9 securely computes $\mathcal{F}_{shuffle}$ in the $\mathcal{F}_{rand}$-hybrid model against a single corruption by a passive adversary.*

Proof. The all values that P_i receives in the protocol is masked by $\overrightarrow{\alpha}_{i-1}$, which is uniformly random for P_i. Therefore, the simulator can simulate $\overrightarrow{\gamma}$ and $\overrightarrow{\delta}$. ■

Optimized unshuffling By changing the order of parties, we have the optimized unshuffling protocol, OptUnshuffle. Due to space limitation, the protocol appears in Appendix C.

Shuffling with reveal We observe that in the step 2-3 of ApplyInv and Compose (Alg. 4 and 5) the output of shuffling is to be revealed. The communication complexity of a sequential invocation of the (optimized) shuffling and reveal protocols is $4m\,|\mathscr{R}'| + 3m\,|\mathscr{R}'|$ bits. We can reduce that into $4m\,|\mathscr{R}'|$ bits by combining the shuffling and reveal protocols.

The shuffling with reveal protocol appears in Algorithm 10. We can confirm completeness as

$$\overrightarrow{a'} = \pi_3 \cdot \pi_2 \cdot (\overrightarrow{\gamma} + \overrightarrow{\delta}) = \pi_3 \cdot \pi_2 \cdot (\pi_1 \cdot \overrightarrow{\beta}_3 + \overrightarrow{\alpha}_1 + \pi_1 \cdot \overrightarrow{\beta}_1 - \overrightarrow{\alpha}_1) = \pi_3 \cdot \pi_2 \cdot \pi_1 \cdot (\overrightarrow{\beta}_3 + \overrightarrow{\beta}_1) = \pi \cdot \overrightarrow{a}.$$

Theorem 5.4. *Algorithm 9 securely compute $\mathcal{F}_{ShuffleReveal}$ in the $\mathcal{F}_{rand}$-hybrid model against a single corruption by a passive adversary.*

Proof. The simulator can simulate the view of P_2 by choosing random vector $\widetilde{\overrightarrow{\gamma}}$ and $\widetilde{\overrightarrow{\delta}}$ such that $\widetilde{\overrightarrow{\gamma}} + \widetilde{\overrightarrow{\delta}} = \pi_2^{-1} \cdot \pi_3^{-1} \cdot \overrightarrow{a}'$. ■

Algorithm 10 Combining shuffling and Reveal

Notation: $\overrightarrow{a}' \leftarrow \text{ShuffleReveal}(\langle\!\langle \pi \rangle\!\rangle; [\overrightarrow{a}])$
Input: A secret-shared vector $[\overrightarrow{a}]$ and a permutation $\langle\!\langle \pi \rangle\!\rangle$.
Output: The shuffled vector $\overrightarrow{a}' = \pi\overrightarrow{a}$.
1: Let $\langle\!\langle \pi \rangle\!\rangle_i = (\pi_i, \pi_{i+1})$.
2: The parties call $\mathcal{F}_{\text{rand}}$ m times and obtain $(\overrightarrow{\alpha}_i, \overrightarrow{\alpha}_{i+1})$ for P_i.
3: P_1 and P_3 compute $\overrightarrow{\beta}_1 := \lambda_1[\overrightarrow{a}]_1$ and $\overrightarrow{\beta}_3 := \lambda_3[\overrightarrow{a}]_3$, respectively.
4: P_3 computes $\overrightarrow{\gamma} := \pi_1 \cdot \overrightarrow{\beta}_3 + \overrightarrow{\alpha}_1$ and sends it to P_2.
5: P_1 computes $\overrightarrow{\delta} := \pi_1 \cdot \overrightarrow{\beta}_1 - \overrightarrow{\alpha}_1$ and sends it to P_2.
6: P_2 computes $\overrightarrow{a'} = \pi_3 \cdot \pi_2 \cdot (\overrightarrow{\gamma} + \overrightarrow{\delta})$ and send it to P_1 and P_3.
7: **return** $\overrightarrow{a'}$

5.2 Optimizations

Reusing permutation in shuffling We observe that the first shuffling in Compose (Alg. 5) can be omitted by reusing a permutation. In both steps 5 and 7 in Alg. 6, the parties receive a random permutation and invokes shuffle with reveal. Thus, we can commonalize them: the parties skip Steps 1 to 3 in Alg. 5 by reusing the values in Steps 1 to 3 in Alg. 4.

Regarding security, this reuse does not affect the construction of a simulator. The simulator uniformly randomly permuted vectors of $(1, \ldots, m)$ as a simulated value of a permutation, and it applies the permutation to $[\![\overrightarrow{k}^{(j)}]\!]$ and $[\![\overrightarrow{\rho_j}]\!]$ in the simulation.

Changing the order of modulus conversion and shuffling from second bit We reduce the communication complexity of one shuffling protocol in the step 5 of GenPerm (Alg. 6) by changing the order of modulus conversion and shuffling protocols. This changes the communication complexity of the corresponding shuffling protocol from $2m\,|\mathscr{R}'|$ to $2m$ bits.

In Algorithm 6, the key shares are decomposed to bit-wise shares in $\mathbb{Z}_2$ and converted to those in $\mathscr{R}'$ at first. These shares of 2nd and latter bits then shuffled in the step 5. Here, the shuffling protocol accepts not only shares of $[\![\cdot]\!]$ but also $\langle\cdot\rangle$, and the communication complexity of the protocol depends on the group/ring size of shares. Therefore, we can reduce the communication complexity by shuffling $\langle \overrightarrow{k}^{(i)} \rangle$ for $2 \leq i \leq \ell$ before converting them to $[\![\overrightarrow{k}^{(i)}]\!]$.

Algorithm 11 Generating permutation of stable sort for multiple bits

Notation: $[\![\overrightarrow{\rho}]\!] \leftarrow \textsc{GenMultiBitSort}([\![\overrightarrow{k^{(1)}}]\!], \ldots, [\![\overrightarrow{k^{(L)}}]\!])$.
Input: Secret-shared L vectors of keys $[\![\overrightarrow{k^{(1)}}]\!], \ldots, [\![\overrightarrow{k^{(L)}}]\!]$.
Output: Shares of the permutation $[\![\overrightarrow{\rho}]\!]$ of the stable sorting by $(\overrightarrow{k}^{(1)}, \ldots, \overrightarrow{k}^{(L)})$
1: **for** $j = 0$ **to** $2^L - 1$ **do**
2: Regard j as an L-bit element $j = B^{(L)} || \cdots || B^{(1)}$.
3: **for** $k' = 1$ **to** L **do**
4: The parties locally compute $[\![D_{k'}]\!] := [\![B^{(k')} \overrightarrow{k}^{(k')} + (1 - B^{(k')})(1 - \overrightarrow{k}^{(k')})]\!]$.
5: The parties compute $[\![\overrightarrow{f}^{(j)}]\!] := [\![\prod_{i=1}^{L} D_i]\!]$ by using OptMult.
6: $[\![s]\!] := [\![0]\!]$
7: **for** $j = 0$ **to** $2^L - 1$ **do**
8: **for** $i = 1$ **to** m **do**
9: $[\![s]\!] := [\![s]\!] + [\![f_i^{(j)}]\!]$.
10: $[\![s_i^{(j)}]\!] := [\![s]\!]$.
11: **for** $1 \leq i \leq m$ **do in parallel**
12: $[\![\rho(i)]\!] \leftarrow \textsc{OptProduct}(([\![f_i^{(0)}]\!], \ldots, [\![f_i^{(2^L-1)}]\!]), ([\![s_i^{(0)}]\!], \ldots, [\![s_i^{(2^L-1)}]\!]))$
13: **return** $[\![\overrightarrow{\rho}]\!] = ([\![\rho(1)]\!], \ldots, [\![\rho(m)]\!])$.

Batch processing of bitwise keys We improve both communication and round complexities of GenPerm (Alg. 6) by processing multiple bitwise keys at a time.

In Algorithm 6, we iteratively compute a secret-shared permutation that represents the stable sort by lower bits of the key by adding bits one by one. Roughly speaking, the number of invocations of permutation-related protocols such as ApplyInv or Compose is reduced to $1/L$ by processing L bits at a time.

We extend Algorithm 3, which can only handle single bit keys, to an algorithm that can handle L-bit keys. The extended algorithm is shown in Algorithm 11. In the extended algorithm, we compute the appearance order of each key from 0 to $2^L - 1$ in order. The number of invocations of OptMult to compute $[\overrightarrow{f}^{(j)}]$ for $0 \leq j \leq 2^L - 1$ is $m(2^L - L - 1)$ by reusing previously computed values.[7] Since the sum-of-product protocol requires the same communication cost as a single multiplication,p the total communication complexity is the same as $m(2^L - L)$ invocations of multiplications.

Although the round complexity gets better for larger L, too large L causes worse communication complexity. We rigorously evaluated the communication complexity and show that $L = 3$ was reasonable in Section 5.3.

5.3 The optimized sorting protocol

We finally obtain the optimized sorting protocol (OptGenPerm, OptApplyInv, OptCompose) in Algorithm 12, 13, and 14 by applying all the protocols and techniques in this section. Let $\widehat{\ell} := \lceil \frac{\ell}{L} \rceil$.

[7] The parties prepare all the combinations of $\{\overrightarrow{k}^{(k')}\}_{1 \leq k' \leq L}$, which costs $m(2^L - L - 1)$ multiplications. Then the parties can obtain $\prod_{i=1}^{L} D_{k'}$ from a linear combination of the prepared combinations since $D_{k''}$ is either $D_{k'}$ or $(1 - D_{k'})$ for any k' and k''.

Algorithm 12 Optimized permutation generation of stable sort

Notation: $[\![\overrightarrow{\sigma}]\!] \leftarrow \textsc{OptGenPerm}([\overrightarrow{k}])$.

Input: Secret-shared keys $[\overrightarrow{k}] = ([k_1], \ldots, [k_m])$.

Output: The secret-shared permutation $[\![\overrightarrow{\sigma}]\!]$ such that σ^{-1} is the stable sorting of $\overrightarrow{k}$.

1: If $[\overrightarrow{k}]$ is not shares of the linear SS scheme in $\mathscr{R}'$, the parties call $\mathcal{F}_{\text{bitdecomp}}$, and obtain bit-wise shares in $\mathbb{Z}_2$: $(\langle \overrightarrow{k}^{(1)} \rangle, \ldots, \langle \overrightarrow{k}^{(\ell)} \rangle)$.
2: The parties send $(\langle \overrightarrow{k}^{(1)} \rangle, \ldots, \langle \overrightarrow{k}^{(L)} \rangle)$ to $\mathcal{F}_{\text{modconv}}$, and receive $([\![\overrightarrow{k}^{(1)}]\!], \ldots, [\![\overrightarrow{k}^{(L)}]\!])$.
3: $[\![\overrightarrow{\rho}_1]\!] \leftarrow \textsc{GenMultiBitSort}([\![\overrightarrow{k}^{(1)}]\!], \ldots, [\![\overrightarrow{k}^{(L)}]\!])$.
4: $[\![\overrightarrow{\sigma}_1]\!] := [\![\overrightarrow{\rho}_1]\!]$.
5: **for** $j = 2$ **to** $\widehat{\ell}$ **do**
6: $\quad \left([\![\overrightarrow{k}'^{((j-1)\widehat{\ell}+1)}]\!], \ldots, [\![\overrightarrow{k}'^{(j\widehat{\ell})}]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''_{j-1}\right) \leftarrow \textsc{OptApplyInv}\left([\![\overrightarrow{\sigma}_{j-1}]\!]; \langle \overrightarrow{k}^{((j-1)\widehat{\ell}+1)} \rangle, \ldots, \langle \overrightarrow{k}^{(j\widehat{\ell})} \rangle\right)$
7: $\quad [\![\overrightarrow{\rho}_j]\!] \leftarrow \textsc{GenMultiBitSort}([\![\overrightarrow{k}'^{((j-1)\widehat{\ell}+1)}]\!], \ldots, [\![\overrightarrow{k}'^{(j\widehat{\ell})}]\!])$.
8: $\quad [\![\overrightarrow{\sigma}_j]\!] \leftarrow \textsc{OptCompose}([\![\overrightarrow{\sigma}_{j-1}]\!], [\![\overrightarrow{\rho}_j]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''_{j-1})$.
9: **return** $[\![\overrightarrow{\sigma}_{\widehat{\ell}}]\!]$.

Algorithm 13 Optimized inverse application of a permutation

Notation: $\left([\![\overrightarrow{k}'^{(1)}]\!], \ldots, [\![\overrightarrow{k}'^{(L)}]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''\right) \leftarrow \textsc{OptApplyInv}\left([\![\overrightarrow{\sigma}]\!]; \langle \overrightarrow{k}^{(1)} \rangle, \ldots, \langle \overrightarrow{k}^{(L)} \rangle\right)$.

Input: A secret-shared permutation $[\![\overrightarrow{\sigma}]\!]$ and a secret-shared vector $\langle \overrightarrow{k}^{(1)} \rangle, \ldots, \langle \overrightarrow{k}^{(L)} \rangle$.

Output: The secret-shared vector $[\![\overrightarrow{k}'^{(1)}]\!], \ldots, [\![\overrightarrow{k}'^{(L)}]\!]$ such that $\overrightarrow{k}'^{(i)} = \sigma^{-1} \cdot \overrightarrow{k}^{(i)}$ for $1 \le i \le L$, a shared permutation $\langle\!\langle \pi \rangle\!\rangle$, and a s permutation σ'' such that $\pi \cdot \sigma$.

1: The parties call $\mathcal{F}_{\text{rand}}$ and receive $\langle\!\langle \pi \rangle\!\rangle$.
2: $\overrightarrow{\sigma}'' \leftarrow \textsc{ShuffleReveal}(\langle\!\langle \pi \rangle\!\rangle; [\![\overrightarrow{\sigma}]\!])$.
3: **for** $1 \le i \le L$ (in parallel) **do**
4: $\quad \langle \overrightarrow{k}''^{(i)} \rangle \leftarrow \textsc{OptShuffle}(\langle\!\langle \pi \rangle\!\rangle; \langle \overrightarrow{k}^{(i)} \rangle)$.
5: $\quad$ The parties send $\langle \overrightarrow{k}''^{(i)} \rangle$ to $\mathcal{F}_{\text{modconv}}$, and receive $[\![\overrightarrow{k}^{(i)}]\!]$.
6: $\quad$ The parties apply $(\sigma'')^{-1}$ with $[\![\overrightarrow{k}''^{(i)}]\!]$ and obtain $[\![\overrightarrow{k}'^{(i)}]\!]$.
7: **return** $\left([\![\overrightarrow{k}'^{(1)}]\!], \ldots, [\![\overrightarrow{k}'^{(L)}]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''\right)$.

Communication improvement The communication complexity of the optimized sorting protocol is as follows.

- GenPerm:
  1. The bit-decomposition and modulus-conversion protocols [24]: $3(5\ell + 2)m$ and $3(1 + |\mathscr{R}'|)m$ bits.
  2. GenMultiBitSort: $m(2^L - L - 1)$ OptMult and m OptProduct.
  3. $\widehat{\ell} - 1$ invocations of:
     - OptApplyInv: single ShuffleReveal for $[\![\cdot]\!]$ and L OptShuffle for $\langle\cdot\rangle$ (the communication of modulus-conversion has already counted in (1)).
     - GenMultiBitSort: $m(2^L - L - 1)$ OptMult and m OptProduct.
     - Compose: single OptUnshuffle for $[\![\cdot]\!]$.
- OptApplyInv to the value: single ShuffleReveal for $[\![\cdot]\!]$ and single OptShuffle for the key shares.

In addition, the communication complexity of component protocols are as follows.

- OptMult and OptProduct: $3\,|\mathscr{R}'|$ bits.
- OptReshare: $2\,|\mathscr{R}'|$ bits.
- OptShuffle, OptUnshuffle, and ShuffleReveal for $[\![\cdot]\!]$: $4m\,|\mathscr{R}'|$ bits.
- OptShuffle for $\langle\cdot\rangle$: $4m$ bits.
- OptShuffle for the key shares: $4m\ell'$ bits.

Algorithm 14 Optimized composition of two permutations

Notation: $[\![\overrightarrow{\sigma'}]\!] \leftarrow \textsc{OptCompose}([\![\overrightarrow{\sigma}]\!], [\![\overrightarrow{\rho}]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma'')$.
Input: Secret-shared two permutations $([\![\overrightarrow{\sigma}]\!], [\![\overrightarrow{\rho}]\!])$.
Output: The secret-shared permutation $[\![\overrightarrow{\sigma'}]\!]$, where $\sigma'^{-1} = \sigma^{-1} \circ \rho^{-1}$.

1: The parties apply σ'' to $[\![\overrightarrow{\rho}]\!]$ and obtain $[\![\overrightarrow{\rho'}]\!]$.
2: $[\![\overrightarrow{\sigma}']\!] \leftarrow \textsc{OptUnshuffle}(\langle\!\langle \pi \rangle\!\rangle; [\![\overrightarrow{\rho}']\!])$.
3: **return** $[\![\overrightarrow{\sigma}']\!]$.

Therefore, the total communication complexity is

$$\begin{aligned}
&3(5\ell+2)m + 3(1+|\mathscr{R}'|)m + 3m(2^L - L)\,|\mathscr{R}'| \\
&+(\widehat{\ell}-1)\left(4m\,|\mathscr{R}'| + 4mL + 3m(2^L-L)\,|\mathscr{R}'| + 4m\,|\mathscr{R}'|\right) + 4m\,|\mathscr{R}'| + 4m\ell' \\
&= m\left(15\ell - 13\,|\mathscr{R}'| + 9 + 4\ell' + \widehat{\ell}(8\,|\mathscr{R}'| + 4L + 3(2^L-L)\,|\mathscr{R}'|)\right)
\end{aligned}$$

For simplicity, assuming $\widehat{\ell} = \ell/L$, we have

$$m\left(15\ell - (13+3\ell)\,|\mathscr{R}'| + 9 + 4\ell' + 4\ell + \frac{\ell}{L}(8\,|\mathscr{R}'| + 2^L 3\,|\mathscr{R}'|)\right).$$

Here, we determine the exact value of L. The part of the equation depending on L is $\frac{1}{L}(8\,|\mathscr{R}'| + 32^L\,|\mathscr{R}'|)$, and this is 4.67, 3.33, and 3.56 if $L = 1, 2, 3$, respectively, and larger than 4.5 when $L > 3$. Therefore, with regard to the communication complexity, $L = 2$ and 3 are reasonable options. On the other hand, larger L reduces the round complexity since the parties invoke OptCompose roughly $1/L$ times. Therefore, we choose $L = 3$.

Substituting $L = 3$, we have

$$m\left(15\ell + (\frac{23}{3}\ell - 13)\,|\mathscr{R}'| + 4\ell' + 4\ell + 9\right).$$

Now we compare the communication complexity of the non-optimized sorting protocol,

$$m\left((66\ell - 33)\,|\mathscr{R}'| + 15\ell + 12\ell' + 9\right).$$

We example the communication bits with parameters $\ell = 20, 40, 60$, $|\mathscr{R}'| = 31, 61$, and $\ell' = 20, 200, 2000$. In any case of the above parameters, 80-88% of the communication bits are cut by the optimization.

6 Implementation and Experiment

We describe in this section the experimental measurements of our implementations of the sorting protocol.

6.1 Settings

Domain of input/output shares The input and output of our protocols are shares of Shamir's scheme in $\mathbb{Z}_p$, where $p = 2^{61} - 1$.

Component protocols We used the following protocols as instantiations of the different functionalities: pseudorandom secret sharing (PRSS) with AES was used as a PRF for $\mathcal{F}_{\text{rand}}$, Algorithm 7 and 15 were used for $\mathcal{F}_{\text{mult}}$ and $\mathcal{F}_{\text{product}}$, the bit-decomposition and modulus-conversion protocols of [24] were used for $\mathcal{F}_{\text{bitdecomp}}$ and $\mathcal{F}_{\text{modconv}}$.

Table 2. Processing times (ms) of our implementation.

Network	Number of rows	Processing time (ms)		
		$\ell = 20$	$\ell = 40$	$\ell = 60$
10G	1×10^4	58	117	168
	1×10^5	159	319	470
	1×10^6	1,190	2,398	3,602
	1×10^7	15,562	31,887	47,134
1G	1×10^4	92	187	273
	1×10^5	490	990	1,463
	1×10^6	4,555	9,064	13,497
	1×10^7	47,714	95,472	141,658
Internet simulation	1×10^4	1,402	2,860	4,220
	1×10^5	6,342	12,794	19,032
	1×10^6	55,367	111,575	167,517
	1×10^7	570,208	1,144,491	1,717,884

Implementation Our implementation utilized asynchronous processing, multi-threading, pipelining of CPU and network, and fast implementation of low level cryptographic operations such as a pseudorandom function. We utilized the extended instructions of AES-NI, RdRand, and SSE4.

Software We implemented the protocols in C++11, on CentOS 7.2.1511, using GCC 4.8.5 and our original cryptography library.

Hardware Each party consisted of a server with two Intel Xeon Gold 6144 3.50GHz and 768GB memory. We prepared three networks for the experiments: as Internet simulation, 1G (Gigabit Ethernet), and 10G (10G Ethernet). For the Internet simulation we limited the network connection by 50 Mbps with 20 ms of roundtrip latency. Regarding network topologies, each server connects through a Gigabit L2 switch, Intel(R) I210, for Internet simulation and 1G connection, and connects with each other directly via an Intel(R) X550T dual port for 10G connection.

The processing times described in this section are the average time in five executions.

6.2 Experiments

We set the bit-length of the key to be $\ell \in \{20, 40, 60\}$. The input shares are converted into bit-wise shares in $\mathbb{Z}_2$ by the bit-decomposition protocol, and then converted into shares in $\mathbb{Z}_{p'}$, where $p' = 2^{31} - 1$. In the experiment, the parties sorted the keys themselves.

We present the experimental result of our sorting protocol for 10G, 1G, and the Internet simulation connections in Table 2. We measured the total time to obtain the permutation via Algorithm 6 and apply it to the key via Algorithm 4. The results demonstrate that within several seconds we can sort a million records in 1G and 10G connections, and 100,000 records in the Internet simulation.

When the number of rows is 10^6 and 10^7, the processing time in the 10G (resp. 1G) connection is roughly 3 (resp. 12) times faster than that in 1G (resp. Internet). These speedups of the processing times are smaller than the bandwidth speedup. One reason is that local computations may not be negligible in terms of processing time. For example, each party locally shuffles m elements in the Shuffle step, which causes many random memory accesses. Another possible cause is that the execution of the Prefix-sum algorithm since it cannot be parallelized.

6.3 Sorting-based data deduplication

We implemented the data deduplication protocol to show that we can construct an application protocol that handles large-scale dataset by using our implementation of the sorting protocol. Suppose that m elements

Table 3. Processing times (ms) of data deduplication.

Network	Number of rows	Processing time (ms)
10G	1×10^6	1,391
	1×10^7	18,836
1G	1×10^6	5,293
	1×10^7	55,479

are secret-shared among three parties. The data deduplication protocol proceeds as follows. First, the parties sort the input as key and value. For each element, the parties compute a secret-shared flag that is 1 when a corresponding element equals to the previous one and 0 otherwise. The parties reveal the sum of flags, f, which represents the number of replicated items. The parties then sort the (sorted) input by the flags as a key. Here, the replicated elements are located at the last of the output since their flags are 1. Finally, the parties discard the last f elements and obtain non-duplicated elements. We describe the precise algorithm in Appendix E.

We experimented the protocol in 1G and 10G connection with large-scale data. The result is shown in Table 3. The data deduplication of 10^6 and 10^7 items takes 1.4 and 19 seconds in 10G connection. It means that the data deduplication of even one million records can be computed efficiently by using our sorting protocol.

7 Conclusion

We proposed a novel three-party sorting protocol secure against passive adversaries in the honest majority setting. The new sorting protocol is based on radix sort and therefore it is stable. It is asymptotically better compared to previous sorting protocols since it does not need to shuffle the entire length of the elements after each comparison step. We then proposed novel protocols and optimizations that reduce about 85% of communication.

We implemented our protocol with those protocols and optimizations. Our experiments showed that our implementation of the sorting protocol is faster by more than two orders of magnitude compared to existing implementations. Our implementation can enable a new set of applications on datasets whose sizes were beyond the reach of secure sorting protocols. We demonstrate it to experiment sorting-based data deduplication protocol. As a result, our implementation did the task in one million items within two seconds in 10G connection. It means that it is now possible to construct applications that handle millions of items.

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC*, pages 1–9, 1983.
2. T. Araki, A. Barak, J. Furukawa, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *IEEE Symposium on Security and Privacy, SP 2017*, 2017.
3. T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *CCS*, pages 805–817, 2016.
4. K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.
5. M. Blanton and E. Aguiar. Private and oblivious set and multiset operations. In *ASIACCS '12*, pages 40–41, New York, NY, USA, 2012. ACM.
6. D. Bogdanov, S. Laur, and R. Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *Nordic Conference on Secure IT Systems*, pages 59–74. Springer, 2014.
7. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.
8. D. Bogdanov, R. Talviste, and J. Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In *Financial Cryptography (FC'12)*, volume 7397 of *LNCS*, pages 57–64. Springer, 2012.

9. K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO 2018*, pages 34–64, 2018.
10. K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, and B. Pinkas. High-throughput secure AES computation. In *WAHC@CCS 2018*, pages 13–24, 2018.
11. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography*, pages 342–362. Springer, 2005.
12. I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.
13. R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111. ACM, 1998.
14. O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
15. M. T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *SODA*, pages 1262–1277, 2010.
16. M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP (2)*, pages 576–587, 2011.
17. K. Hamada, D. Ikarashi, K. Chida, and K. Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint Archive, Report 2014/121, 2014.
18. K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, pages 202–216, 2012.
19. Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.
20. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security'11*, pages 539–554. USENIX, 2011.
21. D. Ikarashi, R. Kikuchi, K. Hamada, and K. Chida. Actively private and correct mpc scheme in $t < n/2$ from passively secure schemes with small overhead. Cryptology ePrint Archive, Report 2014/304, 2014.
22. M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.
23. K. V. Jónsson, G. Kreitz, and M. Uddin. Secure multi-party sorting and applications. In *ACNS*. Springer, 2011.
24. R. Kikuchi, D. Ikarashi, T. Matsuda, K. Hamada, and K. Chida. Efficient bit-decomposition and modulus-conversion protocols with an honest majority. In *ACISP 2018*, pages 64–82, 2018.
25. V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *CCS 2017*, pages 1257–1272, 2017.
26. P. Laud. Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. *PoPETs*, 2015(2):188–205, 2015.
27. S. Laur, R. Talviste, and J. Willemson. From oblivious AES to efficient and secure database join in the multiparty setting. In *ACNS*, pages 84–101, 2013.
28. S. Laur, J. Willemson, and B. Zhang. Round-efficient oblivious database manipulation. In *ISC 2011*, pages 262–277, 2011.
29. B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security'15*, pages 515–530. USENIX, 2015.
30. B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based PSI via cuckoo hashing. In *EUROCRYPT*, pages 125–157, 2018.
31. A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
32. B. Zhang. Generic constant-round oblivious sorting algorithm for MPC. In *ProvSec*, pages 240–256, 2011.

A Sum-of-product protocol

Several multiplication protocols [13, 12] can be extended to compute sum-of-product with *no extra communication cost* [9]. This functionality appears in Functionality A.1.

We show the sum-of-product protocol in Algorithm 15.

The security proof is almost the same as that of OptMult so we omit it.

FUNCTIONALITY A.1 ($\mathcal{F}_{\text{product}}$ – Sum-of-product)

Upon receiving $(([a_1], \ldots, [a_m]), ([b_1], \ldots, [b_m]))$ from P_i for $1 \leq i \leq 3$, $\mathcal{F}_{\text{product}}$ reconstructs (a_j, b_j) from the inputs for $1 \leq j \leq m$, computes $c = \sum_{j=1}^{m} a_i b_i$, obtains shares $[c] \leftarrow \mathsf{Share}(c)$, and sends $[c]_i$ to P_i.

Algorithm 15 Optimized sum-of-product protocol

Notation: $[c] \leftarrow \textsc{OptProduct}(([a_1], \ldots, [a_m]), ([b_1], \ldots, [b_m]))$.
Input: Secret-shared values $([a_1], \ldots, [a_m]), ([b_1], \ldots, [b_m])$.
Output: A secret-shared value $([c])$, where $c = \sum_{j=1}^{m} a_j b_j$.

1: Each P_i for $1 \leq i \leq 3$ computes $\widehat{\beta}_{j,i} := \widehat{\lambda_i}[a_j]_i[b_j]_i$ for $1 \leq j \leq m$, where $\widehat{\lambda_i}$ are Lagrange coefficients, i.e., $ab = \sum_{i=1}^{3} \widehat{\lambda_i}[a_j]_i[b_j]_i$.
2: The parties call $\mathcal{F}_{\text{rand}}$ m times and receive $(\alpha_{j,i}, \alpha_{j,i+1})$ for P_i and $1 \leq j \leq m$.
3: **for** $i = 1$ **to** 3 (in parallel) **do**
4: P_i and P_{i+1} set $[\widehat{\beta}_{j,i}]_{i+1} := \alpha_{j,i+1}$ for $1 \leq j \leq m$.
5: P_i computes $[\widehat{\beta}_i]_k := \sum_{j=1}^{m} [\widehat{\beta}_{j,i}]_k$ for $1 \leq k \leq 3$.
6: P_i computes $([\widehat{\beta}_i]_i, [\widehat{\beta}_i]_{i-1}) \leftarrow \mathsf{ShareSim}(\widehat{\beta}_i, [\widehat{\beta}_i]_{i+1})$.
7: P_i sends $[\widehat{\beta}_i]_{i-1}$ to P_{i-1}.
8: $[c] := [\widehat{\beta}_1] + [\widehat{\beta}_2] + [\widehat{\beta}_3]$.
9: **return** $[c]$.

B Applying a share-vector permutation

Algorithm 5 can be regarded as a protocol applying a share-vector permutation: If we replace $[\![\overrightarrow{\rho}]\!]$ with $[\overrightarrow{k}]$, the output is $[\sigma \cdot \overrightarrow{k}]$, which is applying a share-vector permutation to a vector of shares. We denote this protocol as Apply, and describe that in Algorithm 16. The completeness can be confirmed as

$$\overrightarrow{k}' = \pi^{-1} \cdot \overrightarrow{k}'' = \pi^{-1} \cdot (\sigma \circ \pi) \cdot \overrightarrow{k} = ((\sigma \circ \pi) \circ \pi^{-1}) \cdot \overrightarrow{k} = \sigma \cdot \overrightarrow{k}.$$

Algorithm 16 Applying a share-vector permutation

Notation: $[\overrightarrow{k}'] \leftarrow \textsc{Apply}([\![\overrightarrow{\sigma}]\!]; [\overrightarrow{k}])$.
Input: A secret-shared permutation $[\![\overrightarrow{\sigma}]\!]$ and vector of shares $[\overrightarrow{k}]$
Output: A vector of shares $[\overrightarrow{k}]$ such that $\overrightarrow{k}' = \sigma \overrightarrow{k}$.

1: The parties call $\mathcal{F}_{\text{rand}}$ and obtain $\langle\!\langle \pi \rangle\!\rangle$.
2: $[\![\overrightarrow{\sigma}'']\!] \leftarrow \textsc{Shuffle}(\langle\!\langle \pi \rangle\!\rangle; [\![\overrightarrow{\sigma}]\!])$
3: The parties reveal $[\![\overrightarrow{\sigma}'']\!]$ and obtain σ''.
4: The parties apply σ'' with $[\overrightarrow{k}]$ and obtain $[\overrightarrow{k}'']$.
5: $[\overrightarrow{k}'] \leftarrow \textsc{Unshuffle}(\langle\!\langle \pi \rangle\!\rangle; [\overrightarrow{k}''])$
6: **return** $[\overrightarrow{k}']$.

By using Apply, we can "unsort" a vector of shares, which restores the order of a vector from the sorted vector.

C Optimized unshuffling

We show the optimized unshuffling protocol in Algorithm 17. Similar to the optimized shuffling protocol,

Algorithm 17 Optimized unshuffling protocol

Notation: $[\![\overrightarrow{a}']\!] \leftarrow \text{OptUnshuffle}(\langle\!\langle \pi \rangle\!\rangle; [\overrightarrow{a}])$
Input: A secret-shared vector $[\overrightarrow{a}]$ and a permutation $\langle\!\langle \pi \rangle\!\rangle$.
Output: The secret-shared shuffled vector $[\overrightarrow{a}'] = [\pi^{-1}\overrightarrow{a}]$.

1: Let $\langle\!\langle \pi \rangle\!\rangle_i = (\pi_i, \pi_{i+1})$.
2: The parties call $\mathcal{F}_{\text{rand}}$ m times and obtain $(\overrightarrow{\alpha}_i, \overrightarrow{\alpha}_{i+1})$ for P_i.
3: P_2 and P_3 compute $\overrightarrow{\beta}_2$ and $\overrightarrow{\beta}_3$ from $[\overrightarrow{a}]_2$ and $[\overrightarrow{a}]_3$, respectively, via LocalAdditive.
4: P_3 computes $\overrightarrow{\gamma} := \pi_3^{-1} \cdot \overrightarrow{\beta}_3 + \overrightarrow{\alpha}_3$ and sends it to P_1.
5: P_2 computes $\overrightarrow{\delta} := \pi_2^{-1} \cdot (\pi_3^{-1} \cdot \overrightarrow{\beta}_2 - \overrightarrow{\alpha}_3) - \overrightarrow{\alpha}_2$ and sends it to P_3.
6: P_1 computes $\overrightarrow{\beta}'_1 := \pi_1^{-1} \cdot (\pi_2^{-1} \cdot \overrightarrow{\gamma} + \overrightarrow{\alpha}_2)$.
7: P_3 computes $\overrightarrow{\beta}'_3 := \pi_1^{-1} \cdot \overrightarrow{\delta}$.
8: $[\overrightarrow{a}'] \leftarrow \text{OptReshare}(\overrightarrow{\beta}'_1, \overrightarrow{\beta}'_3)$.
9: **return** $[\overrightarrow{a}']$

Table 4. Comparison with existing sorting protocols in 1G connection

	Number of rows	Processing time [ms]
Radix sort in [6]	1×10^4	40,000
	1×10^5	400,000
Quicksort in [6]	1×10^4	10,000
	1×10^5	150,000
Quicksort in [18]	1×10^4	9,859
	1×10^5	93,674
	1×10^6	1,226,267
Ours ($\ell = 60$)	1×10^4	273
	1×10^5	1,463
	1×10^6	13,497

regarding $\overrightarrow{\beta}'_1$ and $\overrightarrow{\beta}'_3$,

$$\begin{aligned}\overrightarrow{\beta}'_1 &= \pi_1^{-1} \cdot (\pi_2^{-1} \cdot \overrightarrow{\gamma} + \overrightarrow{\alpha}_2) = \pi_1^{-1} \cdot (\pi_2^{-1} \cdot (\pi_3^{-1} \cdot \overrightarrow{\beta}_3 + \overrightarrow{\alpha}_3) + \overrightarrow{\alpha}_2) \\ &= \pi_1^{-1} \cdot \pi_2^{-1} \cdot \pi_3^{-1} \cdot \overrightarrow{\beta}_3 + \pi_1 \cdot \pi_2 \cdot \overrightarrow{\alpha}_3 + \pi_1 \cdot \overrightarrow{\alpha}_2\end{aligned}$$

and

$$\begin{aligned}\overrightarrow{\beta}'_3 &= \pi_1^{-1} \cdot \overrightarrow{\delta} = \pi_1^{-1} \cdot (\pi_2^{-1} \cdot (\pi_3^{-1} \cdot \overrightarrow{\beta}_2 - \overrightarrow{\alpha}_3) - \overrightarrow{\alpha}_2) \\ &= \pi_1^{-1} \cdot \pi_2^{-1} \cdot \pi_3^{-1} \cdot \overrightarrow{\beta}_2 - \pi_1 \cdot \pi_2 \cdot \overrightarrow{\alpha}_3 - \pi_1 \cdot \overrightarrow{\alpha}_2.\end{aligned}$$

Therefore,

$$\overrightarrow{\beta}'_1 + \overrightarrow{\beta}'_3 = \pi_1^{-1} \cdot \pi_2^{-1} \cdot \pi_3^{-1}(\overrightarrow{\beta}_2 + \overrightarrow{\beta}_3) = \pi^{-1} \cdot \overrightarrow{a}.$$

The security proof of OptUnshuffle is almost the same as that of OptShuffle so we omit it.

D Running times of other implementations of sorting protocol

A comparison with the existing sorting protocols is given in Table 4. The table shows running times for sorting the keys themselves. The comparison is over a 1G connection since the existing results were tested in that setting. As for the results of [6] and [18], it is difficult to simply compare our results with the run times that were reported in these papers, since the experiments reported in the papers were held in 2012-2014 and therefore used weaker machine specs.

If the number of rows is 10^5 or 10^6, our new protocol is about 100 times faster than the best reported result. In addition, we recall that all these experiments tested sorting of records which include only the keys.

If the records include values in addition to keys then our protocol performs even better compared to existing protocols.

E Algorithm for data deduplication

We show the data-deduplication protocol in Algorithm 18, where $\textsc{Eq}([a''_i], [a''_{i-1}])$ outputs $\langle 1 \rangle$ if $a''_i = a''_{i-1}$ and $\langle 0 \rangle$ otherwise.

Algorithm 18 Sorting-based data deduplication

Input: Secret-shared values that may have duplication $[\overrightarrow{a}] := ([a_1], \ldots, [a_m])$.
Output: Secret-shared values $[\overrightarrow{a}'] := ([a'_1], \ldots, [a'_{m'}])$, where $a'_i \neq a'_j$ for $i \neq j$ and $1 \leq i, j \leq m'$.

$[\![\sigma]\!] \leftarrow \textsc{OptGenPerm}([\overrightarrow{a}])$
$[\overrightarrow{a}''] \leftarrow \textsc{OptApplyInv}([\![\sigma]\!]; [\overrightarrow{a}])$
$[f_1] := [0]$
for $2 \leq i \leq m$ (in parallel) **do**
    $[f_i] \leftarrow \textsc{Eq}([a''_i], [a''_{i-1}])$
$[\![\sigma']\!] \leftarrow \textsc{OptGenPerm}(\langle \overrightarrow{f} \rangle)$
$[\overrightarrow{a}^*] \leftarrow \textsc{OptApplyInv}([\![\sigma']\!]; [\overrightarrow{a}''])$
Reveal $f := \sum_{i=1}^{m} [f_i]$.
$[\overrightarrow{a}'] := ([a^*_1], \ldots, [a^*_{m-f}])$
return $[\overrightarrow{a}']$.

F Functionalities

FUNCTIONALITY F.1 ($\mathcal{F}_{\text{reshare}}$ – Resharing)

Upon receiving $[a]$, $\mathcal{F}_{\text{reshare}}$ reconstructs a, generates shares $[a] \leftarrow \mathsf{Share}(a)$, and sends $[a]_i$ to P_i

FUNCTIONALITY F.2 ($\mathcal{F}_{\text{shuffle}}$ – Shuffling)

Upon receiving $[\overrightarrow{a}]$ and $\langle\!\langle \pi \rangle\!\rangle$, $\mathcal{F}_{\text{shuffle}}$ reconstructs $\overrightarrow{a}$ and π, computes $\overrightarrow{a}' := \pi \cdot \overrightarrow{a}$, generates shares $[\overrightarrow{a}'] \leftarrow \mathsf{Share}(\overrightarrow{a}')$, and sends $[\overrightarrow{a}']_i$ to P_i

FUNCTIONALITY F.3 ($\mathcal{F}_{\text{unshuffle}}$ – Unshuffling)

Upon receiving $[\overrightarrow{a}]$ and $\langle\!\langle\, \pi \,\rangle\!\rangle$, $\mathcal{F}_{\text{unshuffle}}$ reconstructs $\overrightarrow{a}$ and π, computes $\overrightarrow{a}' := \pi^{-1} \cdot \overrightarrow{a}$, generates shares $[\overrightarrow{a}'] \leftarrow \mathsf{Share}(\overrightarrow{a}')$, and sends $[\overrightarrow{a}']_i$ to P_i

FUNCTIONALITY F.4 ($\mathcal{F}_{\text{ShuffleReveal}}$ – Shuffle with reveal)

Upon receiving $[\overrightarrow{a}]$ and $\langle\!\langle\, \pi \,\rangle\!\rangle$, $\mathcal{F}_{\text{ShuffleReveal}}$ reconstructs $\overrightarrow{a}$ and π, computes $\overrightarrow{a}' := \pi \cdot \overrightarrow{a}$, and sends $\overrightarrow{a}'$ to P_i