Oblivious Radix Sort: An Efficient Sorting Algorithm for Practical Secure Multi-party Computation

Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi

NTT Secure Platform Laboratories, 3-9-11, Midori-cho, Musashino-shi, Tokyo 180-8585 Japan
{hamada.koki,ikarashi.dai,chida.koji,takahashi.katsumi}@lab.ntt.co.jp

Abstract. We propose a simple and efficient sorting algorithm for secure multi-party computation (MPC). The algorithm is designed to be efficient when the number of parties and the size of the underlying field are small. For a constant number of parties and a field with a constant size, the algorithm has $O(n \log n)$ communication complexity, which is asymptotically the same as the best previous algorithm but achieves $O(1)$ round complexity, where n is the number of items. The algorithm is constructed with the help of a new technique called "shuffle-and-reveal." This technique can be seen as an analogue of the frequently used technique of "add random number and reveal." The feasibility of our algorithm is demonstrated by an implementation on an MPC scheme based on Shamir's secret-sharing scheme with three parties and corruption tolerance of 1. Our implementation sorts 1 million 32-bit word secret-shared values in 197 seconds.

1 Introduction

Secure multi-party computation (MPC) protocols allow a set of participants (*parties*) to compute a function privately. That is, when a function is represented as $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$, each party with its private input x_i obtains only the output y_i and nothing else. Although any function can be securely computed using a circuit representation of the function [3, 14], it is not easy to design efficient MPC protocols for complex algorithms. Therefore, efficient MPC protocols for specific important operations, such as computing bit-decomposition and comparison [9] and modulo reduction [23], have been proposed as building blocks.

Sorting is one of the most important primitives in various systems, and many sorting algorithms have been studied. The importance of sorting is also true in the context of MPC protocols. MPC sorting protocols are often required in various database operations, and they have many applications such as in cooperative intrusion detection systems [19], oblivious RAM [10], or private set intersection [18]. Therefore, a number of MPC sorting protocols have also been studied.

Some efficient sorting algorithms used for MPC are known as sorting networks. Ajtai et al. proposed an asymptotically optimal sorting network known as the AKS sorting network, which exhibits a complexity of $O(n \log n)$ comparisons, where n is the number of input elements [1]. However, this algorithm is not practical since its constant factor is very large. By contrast, Batcher's merge sort [2] is more efficient unless n is quite large [20]. This algorithm exhibits a complexity of $O(n \log^2 n)$ comparisons with a smaller constant factor.

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 $O(n)$ rounds and $O(n \log n)$ comparisons. Zhang proposed some data-oblivious sorting algorithms [30]. 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 reconstruct the keys in a sorted form. These algorithms require $O(Rn)$ 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 $O(n^2)$ comparisons, which can also handle key indexed data. Hamada et al. proposed a method for converting sorting algorithms into corresponding sorting protocols [17]. Their quicksort protocol exhibits $O(\log n)$ rounds and $O(n \log n)$ comparisons on average. However, $O(\log n)$ communication overhead is required to resolve the case in which the input values include duplications.

Methods for strengthening MPC sorting protocols have also been studied. Jónsson et al. studied a general method to hide the number of input values for sorting protocols [19]. Goodrich and Mitzenmacher proposed a method to extend internal-memory sorting algorithms to external-memory sort [16].

In addition, a few experimental results have been reported. Wang et al. [29] implemented sorting algorithms on an MPC system called Fairplay [22]. The running times of Batcher's merge sort [2] and randomized shellsort [15] for 256 input values are approximately $3,000$ and $6,200$ seconds, respectively. Jónsson et al. [19] implemented Batcher's merge sort [2] and other sorting protocols on an MPC system known as Sharemind [4]. 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. [17] implemented their quicksort protocol on $(2, 3)$-Shamir's secret-sharing scheme with a corruption tolerance of 1 [27]. 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.

1.1 Our Contributions

We propose an oblivious sorting algorithm called oblivious radix sort for secret-sharing-based MPC schemes. Our algorithm exhibits a complexity of $O(\mathcal{M}\ell)$ rounds and $O(\ell^2 \log \ell m^2 n + \ell^2 m^3 n + \mathcal{M}\ell m^2 n \log n)$ communications, where n is the number of input values, m is the number of parties, ℓ is the bit-length of the underlying field, and $\mathcal{M} = 2^m/\sqrt{m}$. The 2nd and 3rd columns of Table 1 present a comparison between some widely used oblivious sorting algorithms and our algorithm.

Our algorithm is designed to be efficient, especially when m and ℓ are limited. Such a case seems to be common if we think about the practical use. In practice, almost all computers use fixed-sized values, such as 32-bit or 64-bit word integers, and the number of parties is very limited, for example, an auction with three parties by Bogetoft et al. [5] and network monitoring with three parties by Burkhart et al. [6]. The right two

Table 1. Complexities of sorting algorithms. Here, n, m, and ℓ represent number of input values, number of parties, and bit-length of underlying field, respectively. $\mathcal{M} = 2^m/\sqrt{m}$.

Sorting scheme	Complexities		Complexities when m and ℓ are constant	
	Round	Comm.	Round	Comm.
AKS sorting network [1]	$O(\log n)$	$O(\ell^2 m^2 n \log n)$	$O(\log n)$	$O(n \log n)$
Batcher's merge sort [2]	$O(\log^2 n)$	$O(\ell^2 m^2 n \log^2 n)$	$O(\log^2 n)$	$O(n \log^2 n)$
Randomized shellsort [15]	$O(n)$	$O(\ell^2 m^2 n \log n)$	$O(n)$	$O(n \log n)$
Oblivious keyword sort [30]	$O(1)$	$O(\ell^2 m^2 n^2)$	$O(1)$	$O(n^2)$
Quicksort (average case) [17]	$O(\mathcal{M} + \log n)$	$O\left(\begin{matrix}\ell m^2 n \log^2 n + \ell^2 m^2 n \log n \\ +\mathcal{M}(mn \log n + \ell m^2 n)\end{matrix}\right)$	$O(\log n)$	$O(n \log^2 n)$
Proposed algorithm	$O(\mathcal{M}\ell)$	$O\left(\begin{matrix}\ell^2 \log \ell m^2 n + \ell^2 m^3 n \\ +\mathcal{M}\ell m^2 n \log n\end{matrix}\right)$	$O(1)$	$O(n \log n)$

columns of Table 1 present a comparison of complexities under the assumption that m and ℓ are constants. Our algorithm has $O(n \log n)$ communication complexity, which is asymptotically the same as the best previous algorithm, but achieves $O(1)$ round complexity.

To construct the proposed algorithm, we also propose a technique for MPC, which we call the "shuffle-and-reveal" technique. This technique is based on the property that if the set of items included in the encrypted vector is already known, only its order is newly leaked when the vector is revealed. Namely, if the vector has already been randomly permuted, no additional information is leaked. This can be seen as an analogue of the frequently used technique of "add random number and reveal."

The proposed algorithm is efficient not only from a theoretical aspect but also in practice. The feasibility of our sorting algorithm is demonstrated by means of an implementation on an MPC scheme based on $(2, 3)$-Shamir's secret-sharing scheme with corruption tolerance $t = 1$. We also implemented other sorting algorithms [2, 15, 30, 17] for comparison. As a result, the proposed algorithm sorts 1 million 32-bit word secret-shared values in 197 seconds. We present an intuitive graph in Fig.1 and describe detailed experimental results in Section 5.

2 Overview of Proposed Method

In this section, we give an overview of our sorting algorithm. We only describe the input, output, and some simple comments of our algorithm. Then, we show an example of the execution.

Our algorithms are designed to be used as building blocks in the paradigm of computing on shared values, which is one of the most common paradigms for MPC protocols [7]. In this paradigm, secret values are preliminarily shared with a secret-sharing scheme to all parties that participate in the MPC protocols. Then the MPC protocols take secret-shared values as inputs from each party and output the result in secret-shared form.

First, we introduce some necessary notations. For a secret value a, we use $[\![a]\!]_{\mathcal{P}_i}$ to denote the share for the party $\mathcal{P}_i$ where the secret value is a. We use $[\![a]\!]$ to denote the

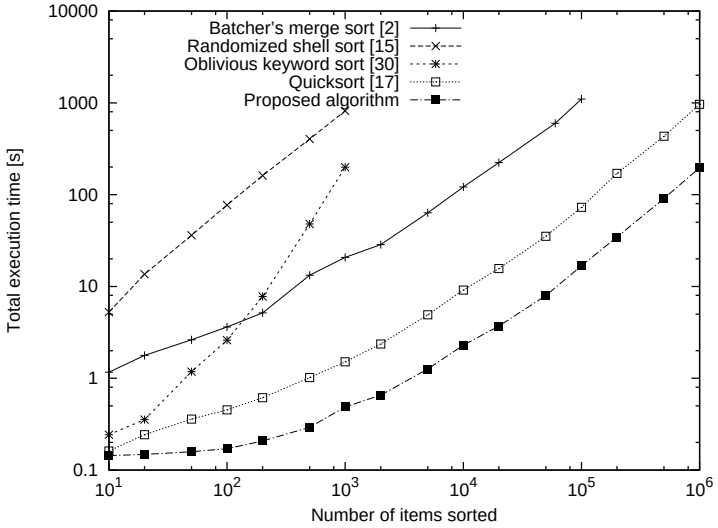

Fig. 1. Running time of five compared sorting implementations. Number of elements on x-axis and y-axis are on log-scale.

list of shares for all parties $[\![a]\!]_{\mathcal{P}_1}, \ldots, [\![a]\!]_{\mathcal{P}_m}$, where m denotes the number of parties. All protocols are executed by m parties $\mathbb{U} = \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$. For a matrix $\boldsymbol{A}$, we use $[\![\boldsymbol{A}]\!]_{\mathcal{P}_i}$ to denote a matrix of shares of elements in $\boldsymbol{A}$ for the party $\mathcal{P}_i$.

Roughly speaking, our sorting algorithm provides the following function:

Definition 1 (Sorting protocol (informal)). *The sorting protocol receives shares of keys and data from each party as inputs. Then it sorts the data according to the keys, and outputs renewed shares of data to each party.*

More precisely, the sorting protocol accepts shares of keys $[\![k_1]\!]_{\mathcal{P}_i}, \ldots, [\![k_m]\!]_{\mathcal{P}_i}$ and shares of data $[\![a_1]\!]_{\mathcal{P}_i}, \ldots, [\![a_m]\!]_{\mathcal{P}_i}$ to be sorted from $\mathcal{P}_i \in \mathbb{U}$, and outputs new shares of data $[\![b_1]\!]_{\mathcal{P}_i}, \ldots, [\![b_m]\!]_{\mathcal{P}_i}$ sorted according to the keys to $\mathcal{P}_i \in \mathbb{U}$. The shares satisfy $k'_i \leq k'_{i+1}$, $k'_j = k_{\pi(i)}$, and $b_j = a_{\pi(i)}$ for a permutation $\pi : \{1, \ldots, m\} \to \{1, \ldots, m\}$. Note that one cannot check the correspondence between $[\![b_j]\!]_{\mathcal{P}_k}$ and $[\![a_{\pi(i)}]\!]_{\mathcal{P}_k}$, where $b_j = a_{\pi(i)}$, since $[\![b_j]\!]_{\mathcal{P}_k}$ is renewed.

Now, we show an example of the execution. For simplicity, we consider the case where the input data and sort keys are given as a matrix of n rows and a column vector of size n, respectively. Our objective is to reorder the matrix column-wise according to the given sort keys in non-descending order. For example, we are given a matrix $\boldsymbol{D}$ and a vector $\boldsymbol{k}$ as below, and an output matrix $\boldsymbol{D}'$ to be computed is as below.

$$\boldsymbol{k} = \begin{pmatrix} 3 \\ 6 \\ 10 \\ 5 \\ 3 \end{pmatrix}, \boldsymbol{D} = \begin{pmatrix} 3 & 5 \\ 6 & 6 \\ 10 & 5 \\ 5 & 5 \\ 3 & 1 \end{pmatrix}, \boldsymbol{D}' = \begin{pmatrix} 3 & 5 \\ 3 & 1 \\ 5 & 5 \\ 6 & 6 \\ 10 & 5 \end{pmatrix}.$$

We show an example of executing our sorting protocol when the underlying secret-sharing scheme is $(2, 3)$-Shamir's secret-sharing scheme [27] with corruption tolerance 1, and each value is an element over a field $\mathbb{Z}_{11} = \{0, 1, \ldots, 10\}$. That is, $[\![s]\!]_{\mathcal{P}_j} = rj + s \bmod 11$ $(1 \leq j \leq 3)$ holds for a random value $r \in \mathbb{Z}_{11}$. The protocols are executed among parties $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$. We demonstrate how the secret-shared values are sorted using our protocol. Let us consider the following situation. If secret values $\boldsymbol{D}$ and keys $\boldsymbol{k}$ are as the above example, and random values for the matrix and keys are chosen as

$$\boldsymbol{R} = \begin{pmatrix} 2 & 0 \\ 4 & 2 \\ 10 & 1 \\ 5 & 3 \\ 8 & 9 \end{pmatrix}, \boldsymbol{r} = \begin{pmatrix} 1 \\ 2 \\ 6 \\ 10 \\ 8 \end{pmatrix},$$

then $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ are given

$$[\![\boldsymbol{D}]\!]_{\mathcal{P}_1} = \begin{pmatrix} 5 & 5 \\ 10 & 8 \\ 9 & 6 \\ 10 & 8 \\ 0 & 10 \end{pmatrix}, [\![\boldsymbol{D}]\!]_{\mathcal{P}_2} = \begin{pmatrix} 7 & 5 \\ 3 & 10 \\ 8 & 7 \\ 4 & 0 \\ 8 & 8 \end{pmatrix}, [\![\boldsymbol{D}]\!]_{\mathcal{P}_3} = \begin{pmatrix} 9 & 5 \\ 7 & 1 \\ 7 & 8 \\ 9 & 3 \\ 5 & 6 \end{pmatrix},$$

and

$$[\![\boldsymbol{k}]\!]_{\mathcal{P}_1} = \begin{pmatrix} 4 \\ 8 \\ 5 \\ 4 \\ 0 \end{pmatrix}, [\![\boldsymbol{k}]\!]_{\mathcal{P}_2} = \begin{pmatrix} 5 \\ 10 \\ 0 \\ 3 \\ 8 \end{pmatrix}, [\![\boldsymbol{k}]\!]_{\mathcal{P}_3} = \begin{pmatrix} 6 \\ 1 \\ 6 \\ 2 \\ 5 \end{pmatrix},$$

respectively. Next $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ execute our sorting protocol with the above shared values as inputs. After that, $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ receive

$$[\![\boldsymbol{D}']\!]_{\mathcal{P}_1} = \begin{pmatrix} 6 & 7 \\ 9 & 1 \\ 7 & 4 \\ 0 & 9 \\ 0 & 9 \end{pmatrix}, [\![\boldsymbol{D}']\!]_{\mathcal{P}_2} = \begin{pmatrix} 9 & 9 \\ 4 & 1 \\ 9 & 3 \\ 5 & 1 \\ 1 & 2 \end{pmatrix}, [\![\boldsymbol{D}']\!]_{\mathcal{P}_3} = \begin{pmatrix} 1 & 0 \\ 10 & 1 \\ 0 & 2 \\ 10 & 4 \\ 2 & 6 \end{pmatrix},$$

respectively if random values $\boldsymbol{R}'$ are chosen as

$$\boldsymbol{R}' = \begin{pmatrix} 3 & 2 \\ 6 & 0 \\ 2 & 10 \\ 5 & 3 \\ 1 & 4 \end{pmatrix}.$$

With our sorting protocol, secret values are correctly sorted as $\boldsymbol{D}'$ in the above example, where the $\pi(i)$-th row of $\boldsymbol{D}$ is equal to the i-th row of $\boldsymbol{D}'$ for a permutation $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 5 & 4 & 2 & 3 \end{pmatrix}$. Note that all the output shared values are renewed, and no party knows the permutation π.

Table 2. Notations for shared vectors and matrices used in this paper.

Notation	Description
$[\![\boldsymbol{b}]\!]_{\mathcal{P}_i} = ([\![\boldsymbol{b}[1]]\!]_{\mathcal{P}_i}, \ldots, [\![\boldsymbol{b}[n]]\!]_{\mathcal{P}_i})^{\mathrm{T}}$	vector of shares (for a party $\mathcal{P}_i$)
$[\![\boldsymbol{b}]\!] = \{[\![\boldsymbol{b}]\!]_{\mathcal{P}_1}, \ldots, [\![\boldsymbol{b}]\!]_{\mathcal{P}_m}\}$	shared vector
$[\![\boldsymbol{B}]\!]_{\mathcal{P}_i} = \begin{pmatrix} [\![\boldsymbol{B}[1,1]]\!]_{\mathcal{P}_i} & \cdots & [\![\boldsymbol{B}[1,d]]\!]_{\mathcal{P}_i} \\ \vdots & \ddots & \vdots \\ [\![\boldsymbol{B}[n,1]]\!]_{\mathcal{P}_i} & \cdots & [\![\boldsymbol{B}[n,d]]\!]_{\mathcal{P}_i} \end{pmatrix}$	matrix of shares (for a party $\mathcal{P}_i$)
$[\![\boldsymbol{B}]\!] = \{[\![\boldsymbol{B}]\!]_{\mathcal{P}_1}, \ldots, [\![\boldsymbol{B}]\!]_{\mathcal{P}_m}\}$	shared matrix

3 Preliminaries

3.1 Assumptions and Notations

We focus on secret-sharing-based MPC. For simplicity, we assume that m parties $\mathcal{P}_1, \ldots, \mathcal{P}_m$ are connected by secure channels. We also assume that the secret-sharing scheme is constructed over a finite field $\mathbb{Z}_P = \{0, 1, \ldots, P-1\}$. Namely, input and output values for the MPC protocols belong to the field $\mathbb{Z}_P$. We use $[\![s]\!]_{\mathcal{P}_i}$ to denote a *share* for $\mathcal{P}_i$ where a *secret value* is $s \in \mathbb{Z}_P$. Let $\mathbb{Q}$ be a coalition of parties and $[\![s]\!]_{\mathbb{Q}}$ denote a set of shares $\{[\![s]\!]_{\mathcal{P}_i} : \mathcal{P}_i \in \mathbb{Q}\}$. When $\mathbb{U}$ represents all parties, we simply denote $[\![s]\!]_{\mathbb{U}}$ as $[\![s]\!]$ and call it a *shared value*. We sometimes use shared values over some fields other than $\mathbb{Z}_P$. We use superscripts with parentheses to explicitly distinguish the field. For example, $[\![s]\!]^{(Q)}$ refers to a shared value over a finite field $\mathbb{Z}_Q = \{0, 1, \ldots, Q-1\}$.

We also use some notations on vectors and matrices. Bold-face, lower-case letters refer to vectors; bold-face, capital letters refer to matrices; and square brackets refer to vector or matrix elements. For example, $\boldsymbol{b}[i]$ refers to the i-th element of vector $\boldsymbol{b}$, and $\boldsymbol{B}[i, j]$ refers to the element in row i and column j of matrix $\boldsymbol{B}$. We also use similar notations for a matrix of shares, or a vector of shared values as indicated in Table 2.

3.2 Security Model

We consider unconditional, perfect security against a semi-honest adversary with static corruption of at most t. We say that a protocol is secure if there is a simulator that simulates the view of corrupted parties from the inputs and outputs of the protocol. We use $\mathbb{I} = \{\mathcal{P}_{i_1}, \mathcal{P}_{i_2}, \ldots, \mathcal{P}_{i_t}\} \subset \mathbb{U}$ to denote the parties that are corrupted.

We give the formal definition of the security against a semi-honest adversary with static corruption. Let $\mathbf{x} = (x_1, \ldots, x_n)$, $\mathbf{x}_{\mathbb{I}} = (x_{i_1}, \ldots, x_{i_t})$, $f_i(\mathbf{x})$ be the i-th output of $f(\mathbf{x})$, and $f_{\mathbb{I}}(\mathbf{x}) = (f_{i_1}(\mathbf{x}), \ldots, f_{i_t}(\mathbf{x}))$. We denote the view of $\mathcal{P}_i$ during the protocol execution of ρ on inputs $\mathbf{x}$ as $\textsc{view}^{\rho}_{\mathcal{P}_i}(\mathbf{x}) = (x_i, r_i; \mu_1, \ldots, \mu_\ell)$ where r_i is $\mathcal{P}_i$'s random tape, and μ_j is the j-th message that $\mathcal{P}_i$ received in the protocol execution. We also denote the output of $\mathcal{P}_i$ as $\textsc{output}^{\rho}_{\mathcal{P}_i}(\mathbf{x})$.

We are now ready to define the security notion in the presence of semi-honest adversaries.

Definition 2 ([13]). *Let* $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ *be a probabilistic n-ary functionality,* ρ *be a protocol,* $\text{VIEW}^{\rho}_{\mathbb{I}}(\mathbf{x}) = (\text{VIEW}^{\rho}_{\mathcal{P}_{i_1}}(\mathbf{x}), \ldots, \text{VIEW}^{\rho}_{\mathcal{P}_{i_t}}(\mathbf{x}))$*, and*

$$\text{OUTPUT}^{\rho}(\mathbf{x}) = (\text{OUTPUT}^{\rho}_{\mathcal{P}_1}(\mathbf{x}), \ldots, \text{OUTPUT}^{\rho}_{\mathcal{P}_n}(\mathbf{x})).$$

We say that ρ *t-privately computes* f *if there exists* $\mathcal{S}$ *such that for all* $\mathbb{I} \subset \mathbb{U}$ *of cardinality of at most t and all* $\mathbf{x}$ *, it holds that*

$$\{(\mathcal{S}(\mathbb{I}, \mathbf{x}_{\mathbb{I}}, f_{\mathbb{I}}(\mathbf{x})), f(\mathbf{x}))\} \equiv \left\{\left(\text{VIEW}^{\rho}_{\mathbb{I}}(\mathbf{x}), \text{OUTPUT}^{\rho}(\mathbf{x})\right)\right\}.$$

It is well known that a protocol satisfying the above security notions can be securely composed with other protocols in a semi-honest setting. To explain this composition property, we introduce the security notion for a protocol that computes a function with the help of an oracle.

Definition 3 ([13]). *Let* $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ *be a probabilistic n-ary functionality,* $g : (\{0,1\}^*)^m \to (\{0,1\}^*)^m$ *be a probabilistic m-ary functionality, and* ρ *be a protocol. We say that* ρ *t-privately reduces g to* f *if* ρ *privately computes g with an oracle access of the functionality of* f*.*

We introduce an informal description of the composition theorem. Suppose that a protocol Π^g privately reduces g to f and a protocol Π^f privately computes f. Then the protocol $\Pi^{g|f}$, which is the same as Π^g except that all oracle calls are substituted with the executions of Π^f, privately computes g. This implies that we can treat a constitutive protocol as a black box to prove the security of a high-level protocol.

3.3 Complexity Metrics in MPC

We use two metrics, *round complexity* and *communication complexity*, to evaluate the overall running time of protocols. The round complexity of a protocol is the number of rounds of parallel invocations of the communication. The communication complexity of a protocol is the total amount of data communicated between parties.

3.4 Secret-Sharing Scheme

A secret-sharing scheme Π_{SS} is a pair of algorithms, *dealing* and *revealing*. The dealing algorithm takes a secret value s as input and outputs a uniformly random shared value $[\![s]\!]$. We say a shared value $[\![s]\!]$ is *uniformly random* if it is uniformly randomly chosen from the set of possible shared values whose secret value is s. The revealing algorithm takes a subset of shares $[\![s]\!]$ and outputs s. We assume that the complexities of dealing and revealing protocols are as summarized in Table 3. We can easily construct protocols with such complexities. For dealing, for example, a party that has the value to be shared computes the shares for all parties by applying the dealing algorithm and then sends the shares to each party. For revealing, for example, every party receives the shares from all other parties and, reconstructs the secret value by applying the revealing algorithm.

Table 3. Round and communication complexities of existing protocols implemented on Shamir's secret-sharing scheme. Input and output shared values of bit-field-conversion protocol are over $\mathbb{Z}_P$ and $\mathbb{Z}_Q$, and number of input shared values of shuffling protocol is n, and $\mathcal{M} = 2^m/\sqrt{m}$. ℓ_P and ℓ_Q represent bit-lengths of fields $\mathbb{Z}_P$ for input and $\mathbb{Z}_Q$ for output, respectively.

Protocol	Round complexity	Communication complexity
Dealing	$O(1)$	$O(\ell m)$
Revealing	$O(1)$	$O(\ell m^2)$
Addition	-	-
Multiplication	$O(1)$	$O(\ell m^2)$
Bit-decomposition [9]	$O(1)$	$O(\ell^2 \log \ell m^2)$
Bit-field-conversion [11]	$O(1)$	$O((\ell_P + \ell_Q)m^3)$
Shuffling [21]	$O(\mathcal{M})$	$O(\mathcal{M}(mn \log n + \ell m^2 n))$

3.5 Arithmetic on Secret-Shared Values

We also assume that the secret-sharing scheme provides secure *addition* and *multiplication* protocols on shared values. That is, we are given two shared values $[\![a]\!]$ and $[\![b]\!]$; we compute shared values $[\![a + b \mod P]\!]$ and $[\![ab \mod P]\!]$. When we write $[\![a]\!] + [\![b]\!] = [\![a + b \mod P]\!]$ and $[\![a]\!] \times [\![b]\!] = [\![ab \mod P]\!]$, it means that the parties perform these operations. We also use $\sum$, for example $\sum_{i=1}^{3} [\![a_i]\!]$, to denote $[\![a_1]\!] + [\![a_2]\!] + [\![a_3]\!]$.

In typical secret-sharing schemes such as Shamir's secret-sharing scheme, the addition is conducted by only local computations [3], and multiplication is conducted with $O(1)$ rounds and $O(\ell m^2)$ communications [12]. The complexities are summarized in Table 3. We use these complexities for evaluating the complexities of our algorithms.

3.6 Existing Protocols

We introduce some existing MPC protocols used as building blocks of our algorithm. The complexities for the protocols that we will use are summarized in Table 3.

Note that our protocols are designed to be used as building blocks in complex MPC protocols. That is, our protocols receive a sequence of secret-shared values as input and the output values are also in secret-shared form.

Bit-decomposition protocol The *bit-decomposition* protocol [9, 24] converts a shared value into shared bits of input values. More precisely, the bit-decomposition protocol accepts $[\![a]\!]_{\mathcal{P}_i}$ from each $\mathcal{P}_i \in \mathbb{U}$ as input and outputs $([\![a_\ell]\!]_{\mathcal{P}_i}, \ldots, [\![a_1]\!]_{\mathcal{P}_i})$ to each $\mathcal{P}_i \in \mathbb{U}$ such that $a_j \in \{0, 1\}$ and $a = \sum_j a_j 2^{j-1}$. We denote this protocol as

$$([\![a_\ell]\!], \ldots, [\![a_1]\!]) \leftarrow \mathsf{Bit\text{-}Decomposition}([\![a]\!]).$$

We formally define the bit-decomposition protocol with the following function f^{BD}.

Definition 4 (Bit-decomposition function). *On inputting $[\![a]\!]_{\mathcal{P}_i}$ from each $\mathcal{P}_i \in \mathbb{U}$, it reveals a with the revealing algorithm of Π_{SS}, computes the bits $a_\ell, \ldots, a_1$ of a, and generates $[\![a_\ell]\!], \ldots, [\![a_1]\!]$ with the dealing algorithm of Π_{SS}. Finally, it outputs $([\![a_\ell]\!]_{\mathcal{P}_i}, \ldots, [\![a_1]\!]_{\mathcal{P}_i})$ to each $\mathcal{P}_i \in \mathbb{U}$.*

The bit-decomposition protocol proposed by Damgård et al. [9] exhibits the complexity of $O(1)$ rounds and $O(\ell \log \ell)$ invocations of multiplication protocols where ℓ is the bit-length of the field.

Shuffling protocol The *shuffling* protocol receives some shared values and outputs renewed shared values where their secret values are uniformly randomly permuted. More precisely, the shuffling protocol accepts $[\![a_1]\!]_{\mathcal{P}_i}, \ldots, [\![a_m]\!]_{\mathcal{P}_i}$ from each $\mathcal{P}_i \in \mathbb{U}$ and outputs $[\![b_1]\!]_{\mathcal{P}_i}, \ldots, [\![b_m]\!]_{\mathcal{P}_i}$ to each $\mathcal{P}_i \in \mathbb{U}$ such that $b_j = a_{\pi(j)}$ for a uniformly random permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and every $j \in \{1, \ldots, n\}$. A run of this protocol is denoted as $[\![b_1]\!], \ldots, [\![b_m]\!] \leftarrow \mathsf{Shuffle}([\![a_1]\!], \ldots, [\![a_m]\!])$.

$$[\![b_1]\!], \ldots, [\![b_m]\!] \leftarrow \mathsf{Shuffle}([\![a_1]\!], \ldots, [\![a_m]\!]).$$

We formally define the shuffling protocol with the following function f^{Shuffle}.

Definition 5 (Shuffling function). *On inputting* $([\![a_1]\!]_{\mathcal{P}_i}, \ldots, [\![a_n]\!]_{\mathcal{P}_i})$ *from each* $\mathcal{P}_i \in \mathbb{U}$*, it reveals* $a_1, \ldots, a_n$ *with the revealing algorithm of* Π_{SS}*, selects a permutation* $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ *uniformly at random, sets* $b_i = a_{\pi(i)}$ *for* $i \in \{1, \ldots, n\}$*, and generates* $[\![b_1]\!], \ldots, [\![b_n]\!]$ *with the dealing algorithm of* Π_{SS}*. Finally, it outputs* $([\![b_1]\!]_{\mathcal{P}_i}, \ldots, [\![b_n]\!]_{\mathcal{P}_i})$ *to each* $\mathcal{P}_i \in \mathbb{U}$*.*

Laur et al. proposed efficient shuffling protocols [21]. One of their protocols exhibits the complexity of $O(2^m/\sqrt{m})$ rounds and $O(2^m m^{3/2} n \log n)$ communications. When the number of parties is constant, it exhibits $O(1)$ rounds and $O(n \log n)$ communications. We use this protocol as the shuffling protocol. We sometimes write $([\![\boldsymbol{a}']\!][1], [\![\boldsymbol{b}']\!][1]), \ldots, ([\![\boldsymbol{a}']\!][n], [\![\boldsymbol{b}']\!][n]) \leftarrow \mathsf{Shuffle}(([\![\boldsymbol{a}]\!][1], [\![\boldsymbol{b}]\!][1]), \ldots, ([\![\boldsymbol{a}]\!][n], [\![\boldsymbol{b}]\!][n]))$ as $[\![\boldsymbol{a}']\!], [\![\boldsymbol{b}']\!] \leftarrow \mathsf{Shuffle}([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$.

Bit-field-conversion protocol The *bit-field-conversion* protocol receives a shared bit over one field and outputs a shared bit over another field. More precisely, the bit-field-conversion protocol accepts a share $[\![b]\!]^{(P)}_{\mathcal{P}_i}$ over a finite field $\mathbb{Z}_P$ from each $\mathcal{P}_i \in \mathbb{U}$ and outputs a share $[\![b]\!]^{(Q)}_{\mathcal{P}_i}$ over another field $\mathbb{Z}_Q$ to each $\mathcal{P}_i \in \mathbb{U}$. A run of this protocol is denoted as

$$[\![b]\!]^{(Q)} \leftarrow \mathsf{Bit\text{-}Field\text{-}Conversion}([\![b]\!]^{(P)}).$$

We formally define the bit-field-conversion protocol with the following function f^{BFC}.

Definition 6 (Bit-field-conversion function). *On inputting* $[\![b]\!]^{(P)}_{\mathcal{P}_i}$ *from each* $\mathcal{P}_i \in \mathbb{U}$*, it reveals* b *with the revealing algorithm of* Π_{SS} *over* $\mathbb{Z}_P$ *and generates* $[\![b]\!]^{(Q)}$ *with the dealing algorithm of* Π_{SS} *over* $\mathbb{Z}_Q$*. Finally, it outputs* $[\![b]\!]^{(Q)}_{\mathcal{P}_i}$ *to each* $\mathcal{P}_i \in \mathbb{U}$*.*

Dodis et al. proposed an efficient bit-field-conversion protocol [11]. Their protocol exhibits the complexity of $O(1)$ rounds and $O(m^3(\ell_P + \ell_Q))$ communications. We use this protocol as the bit-field-conversion protocol.

Revealing protocol The *revealing* protocol accepts $[\![a]\!]_{\mathcal{P}_i}$ from each $\mathcal{P}_i \in \mathbb{U}$ and outputs a to each $\mathcal{P}_i \in \mathbb{U}$. This protocol just serves as the revealing algorithm in a multi-party setting. A run of this protocol is denoted as

$$a \leftarrow \mathsf{Reveal}([\![a]\!]).$$

We formally define the revealing protocol with the following function f^{Reveal}.

Definition 7 (Revealing function). *On inputting $[\![a]\!]_{\mathcal{P}_i}$ from each $\mathcal{P}_i \in \mathbb{U}$, it reveals a with the revealing algorithm of Π_{SS} and outputs a to each $\mathcal{P}_i \in \mathbb{U}$.*

The revealing protocol can be easily constructed in a semi-honest model by distributing all shares among all parties. Even in a malicious model it can be constructed by using the *secret-sharing scheme against cheaters* [26, 25].

3.7 Radix Sort

Our algorithm is based on the sorting algorithm *radix sort* [8]. Radix sort is a kind of sorting algorithm that sorts fixed-length integers by repeatedly applying a stable sort algorithm for smaller integers. We say a sorting algorithm is stable if it maintains the relative order of items with equal keys. Radix sort uses the following property of stable sorts. If the values are stably sorted according to the lower i digits, we can stably sort the values according to the lower $i + 1$ digits by just applying stable sort according to the $(i + 1)$th digit. Namely, radix sort sorts the values by iteratively applying digitwise stable sort from the least significant digit to the most significant digit.

4 Proposed Algorithm

In this section, we propose an efficient oblivious sorting algorithm, which we call *oblivious radix sort*. We first propose a stable sorting algorithm called *binary key sort*, which is applicable only when the input keys are binary. Then we efficiently extend the binary key sort to the case in which the input keys are not limited to being binary by constructing a secure variant of the pointer technique for MPC.

4.1 Binary Key Sort

We begin by introducing two building-block algorithms: *destination computation* and *reveal sort*. Then we use them to construct the binary key sort.

Destination computation The destination computation algorithm receives a shared vector and outputs another shared vector. The i-th element of the output vector represents the destination position for the i-th element of the input vector after the input vector is stably sorted.

To be more precise, the input shared vector is given in a special form, which we call a *matrix representation*. We say an $n \times d$ matrix $\boldsymbol{B}$ is a matrix representation of a vector $\boldsymbol{k}$ if the condition

$$\boldsymbol{B}_{i,j} = \begin{cases} 1 \text{ if } \boldsymbol{k}[i] = j - 1, \\ 0 \text{ otherwise} \end{cases}$$

Algorithm 1 Destination computation

Notation: $[\![\boldsymbol{c}]\!] \leftarrow \mathsf{Dest\text{-}Comp}([\![\boldsymbol{B}]\!])$

Data: An $n \times d$ shared matrix $[\![\boldsymbol{B}]\!]$, where $\boldsymbol{B}$ is a matrix representation of $\boldsymbol{k} \in \{0, \ldots, d-1\}^n$.

Result: A shared vector $[\![\boldsymbol{c}]\!]$, where $\boldsymbol{c}[i]$ represents the position of $\boldsymbol{k}[i]$ after $\boldsymbol{k}$ is stably sorted.

1: Compute the prefix sum of $[\![\boldsymbol{B}]\!]$ in column-major order, and let the resultant shared matrix be $[\![\boldsymbol{S}]\!]$. That is, $[\![\boldsymbol{S}[i,j]]\!] := \sum_{(i',j') \text{s.t.} (j'<j) \vee (j'=j \wedge i' \leq i)} [\![\boldsymbol{B}[i',j']]\!]$ for $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,d\}$.
2: $[\![\boldsymbol{T}[i,j]]\!] := [\![\boldsymbol{S}[i,j]]\!] \times [\![\boldsymbol{B}[i,j]]\!]$ for $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,d\}$. ▷ $O(d\ell m^2 n)$ com, $O(1)$ rnd
3: $[\![\boldsymbol{c}[i]]\!] := \sum_{j=1}^{d} [\![\boldsymbol{T}[i,j]]\!]$ for $i \in \{1,\ldots,n\}$.
4: **return** $[\![\boldsymbol{c}]\!]$.

holds for $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,d\}$, where $\boldsymbol{k} \in \{0,\ldots,d-1\}^n$. We give an example of a vector $\boldsymbol{k}$ and its matrix representation $\boldsymbol{B}$ for illustration:

$$\boldsymbol{k} = \begin{pmatrix} 2 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \boldsymbol{B} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Now, we formally define the destination computation algorithm with the following function f^{DC}:

Definition 8 (Destination computation). *On inputting $[\![\boldsymbol{B}]\!]_{\mathcal{P}_i}$ from each $\mathcal{P}_i \in \mathbb{U}$, it reveals $\boldsymbol{B}$ with the revealing algorithm of Π_{SS}, computes $\boldsymbol{k}$ s.t. $\boldsymbol{k}[i] = \sum_{j=1}^{d} \boldsymbol{B}[i,j] \times (j-1)$, stably sorts $(\boldsymbol{k}[1],1),\ldots,(\boldsymbol{k}[n],n)$ to $(\boldsymbol{k}'[1],\pi(1)),\ldots,(\boldsymbol{k}'[n],\pi(n))$ such that $\boldsymbol{k}'[i] \leq \boldsymbol{k}'[i+1]$ for $i \in \{1,\ldots,n-1\}$, computes $\boldsymbol{c}$ s.t. $\boldsymbol{c}[i] = \pi^{-1}(i)$, and generates $[\![\boldsymbol{c}]\!]$ with the dealing algorithm of Π_{SS}. Finally, it outputs $[\![\boldsymbol{c}]\!]_{\mathcal{P}_i}$ to each $\mathcal{P}_i \in \mathbb{U}$.*

The algorithm is shown as Algorithm 1. We first compute every prefix sum of the input matrix in column-major order. Starting from the top-left element, we sum up the values along the first column then continue to the following columns. The computed prefix sums are output as a matrix $\boldsymbol{S}$. Then, we multiply $\boldsymbol{S}[i,j]$ and $\boldsymbol{B}[i,j]$, and obtain another matrix $\boldsymbol{T}$. Finally, we sum up the elements of $\boldsymbol{T}$ for each row and obtain the final destination positions $\boldsymbol{c}$. For matrix $\boldsymbol{B}$ described above, $\boldsymbol{S}$, $\boldsymbol{T}$, and $\boldsymbol{c}$ are computed as follows:

$$\boldsymbol{S} = \begin{pmatrix} 0 & 2 & 4 \\ 1 & 2 & 4 \\ 2 & 2 & 4 \\ 2 & 3 & 4 \end{pmatrix}, \boldsymbol{T} = \begin{pmatrix} 0 & 0 & 4 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}, \boldsymbol{c} = \begin{pmatrix} 4 \\ 1 \\ 2 \\ 3 \end{pmatrix}.$$

We can efficiently construct the matrix representation when the elements of $\boldsymbol{k}$ are binary since the negation of $\boldsymbol{k}$ and $\boldsymbol{k}$ correspond to the first and second columns of $\boldsymbol{k}$'s matrix representation. For example, if $\boldsymbol{k} = (1,0,0,1)^{\mathrm{T}}$, we join its negation $(0,1,1,0)^{\mathrm{T}}$ and the original vector and obtain the matrix representation of $\boldsymbol{k}$ as $\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}^{\mathrm{T}}$. This conversion requires no communication since the negation for a binary value $b \in \{0,1\}$ is computed by $1-b$.

Algorithm 2 Reveal sort

Notation: $\boldsymbol{k}', [\![\boldsymbol{D}']\!] \leftarrow \mathsf{Reveal\text{-}Sort}([\![\boldsymbol{k}]\!]^{(Q)}, [\![\boldsymbol{D}]\!])$

Data: A shared vector $[\![\boldsymbol{k}]\!]^{(Q)}$ of size n and an $n \times e$ shared matrix $[\![\boldsymbol{D}]\!]$, where $n \in \mathbb{Z}_Q$.

Result: A shared matrix $[\![\boldsymbol{D}']\!]$, where $\boldsymbol{D}'$ is a sorted matrix of $\boldsymbol{D}$ according to $\boldsymbol{k}$. The components of $\boldsymbol{k}$ are revealed.

1: $[\![\boldsymbol{k}'']\!]^{(Q)}, [\![\boldsymbol{D}'']\!] \leftarrow \mathsf{Shuffle}([\![\boldsymbol{k}]\!]^{(Q)}, [\![\boldsymbol{D}]\!])$. ▷ $O(\mathcal{M}e\ell m^2 n + \mathcal{M}m^2 n \log n)$ com, $O(\mathcal{M})$ rnd

2: $\boldsymbol{k}''[i] \leftarrow \mathsf{Reveal}([\![\boldsymbol{k}''[i]]\!]^{(Q)})$ for $i \in \{1, \ldots, n\}$. ▷ $O(m^2 n \log n)$ com, $O(1)$ rnd

3: Sort $\boldsymbol{k}''$ and $[\![\boldsymbol{D}'']\!]$ according to $\boldsymbol{k}''$, and output as $\boldsymbol{k}'$ and $[\![\boldsymbol{D}']\!]$, respectively.

Correctness The destination position for the i-th element is determined by the number of elements smaller than the i-th element and the number of elements that are equal to the i-th element and that appear no later than the i-th element. By the property of the matrix representation, the former is equal to the number of 1's in the left $\boldsymbol{k}[i]$ columns, and the latter is equal to the number of 1's in the upper i rows in the $(\boldsymbol{k}[i]+1)$th column. Therefore, $\boldsymbol{S}[i, \boldsymbol{k}[i] + 1]$ is equal to the destination position for the i-th element. Since $\boldsymbol{B}[i, j] = 1$ if and only if $j = \boldsymbol{k}[i] + 1$, we have $\boldsymbol{c}[i] = \boldsymbol{S}[i, \boldsymbol{k}[i] + 1]$.

Security Since Algorithm 1 consists of only addition and multiplication protocols, we have the following theorem.

Theorem 1. *Algorithm 1 privately computes* f^{DC}.

Complexity Since the addition requires no communication, we only take into account the multiplications. The complexities for each step are described in Algorithm 1 as "▷ x com, y rnd," where x and y represent the communication and round complexities, respectively. Since dn multiplications are conducted in the algorithm and these multiplications can be conducted in parallel, this protocol exhibits $O(d\ell m^2 n)$ communications in $O(1)$ rounds.

Reveal sort We introduce a special efficient sorting algorithm that we call reveal sort. The reveal sort is a sorting algorithm when the keys, which are components of inputs, are publicly known. We can use this algorithm to sort the data associated with positions calculated by the destination computation, since we know that the positions are permutations of $(1, \ldots, n)$. We formally define the reveal sort algorithm with the following function f^{RS}:

Definition 9 (Reveal sort). *On inputting* $(\![\boldsymbol{k}]\!]^{(Q)}_{\mathcal{P}_i}, [\![\boldsymbol{D}]\!]_{\mathcal{P}_i})$ *from each* $\mathcal{P}_i \in \mathbb{U}$*, it reveals* $\boldsymbol{k}$ *and* $\boldsymbol{D}$ *with the revealing algorithm of* Π_{SS}*, sorts* $(\boldsymbol{k}[1], \boldsymbol{D}[1, 1], \ldots, \boldsymbol{D}[1, e])$ $, \ldots, (\boldsymbol{k}[n], \boldsymbol{D}[n, 1], \ldots, \boldsymbol{D}[n, e])$ *to* $(\boldsymbol{k}'[1], \boldsymbol{D}'[1, 1], \ldots, \boldsymbol{D}'[1, e]), \ldots, (\boldsymbol{k}'[n], \boldsymbol{D}'[n, 1],$ $\ldots, \boldsymbol{D}'[n, e])$ *such that* $\boldsymbol{k}'[i] \leq \boldsymbol{k}'[i + 1]$ *for* $i \in \{1, \ldots, n - 1\}$*, and generates* $[\![\boldsymbol{D}']\!]$ *with the dealing algorithm of* Π_{SS}*. Finally, it outputs* $(\boldsymbol{k}', [\![\boldsymbol{D}']\!]_{\mathcal{P}_i})$ *to each* $\mathcal{P}_i \in \mathbb{U}$.

The idea is to use the property that the components of input keys are publicly known. In such a case, we can see the keys since we have already known the set of keys; however, we should not learn how the keys are ordered in the input vector. To hide the order of the keys, we propose a technique called "shuffle-and-reveal." As the name implies,

the technique is just shuffling the values and then revealing them. We can efficiently construct reveal sort by using the "shuffle-and-reveal" technique: we first shuffle the keys and associated data then sort the data according to the revealed keys. The details of the algorithm are shown as Algorithm 2.

Correctness Since $[\![\boldsymbol{k}]\!]^{(\mathcal{Q})}$ and $[\![\boldsymbol{D}]\!]$ are shuffled together and $[\![\boldsymbol{D}'']\!]$ is sorted according to $\boldsymbol{k}''$, $\boldsymbol{D}'$ is a sorted matrix of $\boldsymbol{D}$ according to $\boldsymbol{k}$.

Security Since Algorithm 2 consists of secure subprotocols, the only possibility of information leakage would be the vector $\boldsymbol{k}''$ revealed by $\mathsf{Reveal}([\![\boldsymbol{k}'']\!]^{(\mathcal{Q})})$. $[\![\boldsymbol{k}'']\!]^{(\mathcal{Q})}$ is shuffled before $\mathsf{Reveal}([\![\boldsymbol{k}'']\!]^{(\mathcal{Q})})$ is conducted, and only the components of $\boldsymbol{k}$, which are part of the outputs, are revealed. Thus, Algorithm 2 leaks no additional information.

Theorem 2. *Algorithm 2 privately reduces f^{RS} to f^{Shuffle} and f^{Reveal}.*

Proof. The view of adversaries consists of their inputs $[\![\boldsymbol{k}]\!]_{\mathbb{I}}^{(\mathcal{Q})}$ and $[\![\boldsymbol{D}]\!]_{\mathbb{I}}$, random tapes, $[\![\boldsymbol{k}'']\!]_{\mathbb{I}}^{(\mathcal{Q})}$, $\boldsymbol{k}''$, and $[\![\boldsymbol{D}'']\!]_{\mathbb{I}}$. The output consists of $\boldsymbol{k}'$ and $[\![\boldsymbol{D}']\!]_{\mathbb{I}}$. Note that the adversaries have no view of the subprotocols $\mathsf{Shuffle}(\cdot)$ and $\mathsf{Reveal}(\cdot)$, since the execution of these protocols are substituted with the oracle invocation of functionalities f^{Shuffle} and f^{Reveal}, respectively.

We construct the simulator $\mathcal{S}$ as follows. Inputs and outputs are the same as those of adversaries, and $\mathcal{S}$ selects random tapes uniformly at random. As for $\boldsymbol{k}''$ and $[\![\boldsymbol{D}'']\!]_{\mathbb{I}}$, $\mathcal{S}$ reorders $(\boldsymbol{k}', [\![\boldsymbol{D}']\!]_{\mathbb{I}})$ column-wise according to uniformly randomly chosen permutation and sets the reordered values as the simulated values for $(\boldsymbol{k}'', [\![\boldsymbol{D}'']\!]_{\mathbb{I}})$. This perfectly simulates $(\boldsymbol{k}'', [\![\boldsymbol{D}'']\!]_{\mathbb{I}})$ since $(\boldsymbol{k}', [\![\boldsymbol{D}']\!]_{\mathbb{I}})$ is the column-wise permutation of $(\boldsymbol{k}'', [\![\boldsymbol{D}'']\!]_{\mathbb{I}})$, $(\boldsymbol{k}'', [\![\boldsymbol{D}'']\!]_{\mathbb{I}})$ is the output of $\mathsf{Shuffle}(\cdot)$, and $(\boldsymbol{k}', [\![\boldsymbol{D}']\!]_{\mathbb{I}})$ is independent of how $(\boldsymbol{k}'', [\![\boldsymbol{D}'']\!]_{\mathbb{I}})$ is ordered. As for $[\![\boldsymbol{k}'']\!]_{\mathbb{I}}^{(\mathcal{Q})}$, $\mathcal{S}$ generates uniformly random numbers, and sets them as the simulated values for $[\![\boldsymbol{k}'']\!]_{\mathbb{I}}^{(\mathcal{Q})}$. Since $[\![\boldsymbol{k}'']\!]_{\mathbb{I}}^{(\mathcal{Q})}$ is the output share of $\mathsf{Shuffle}(\cdot)$, and $\mathsf{Shuffle}(\cdot)$ outputs uniformly random shares, the above simulation is perfect.

Thus, $\mathcal{S}$ perfectly simulates the view of adversaries. □

Complexity Communication and round complexities for each step are denoted as comments in Algorithm 2. Therefore, the complexities of reveal sort are $O(\mathcal{M})$ rounds and $O(\mathcal{M}\ell m^2 n + \mathcal{M} m^2 n \log n)$ communications.

Binary key sort By combining the two aforementioned algorithms, we derive the binary key sort, which sorts data according to associated binary keys. We formally define the binary key sort algorithm with the following function f^{Binary}:

Definition 10 (Binary key sort). *On inputting $([\![\boldsymbol{b}]\!]_{\mathcal{P}_i}, [\![\boldsymbol{D}]\!]_{\mathcal{P}_i})$ from each $\mathcal{P}_i \in \mathbb{U}$, it reveals $\boldsymbol{b}$ and $\boldsymbol{D}$ with the revealing algorithm of Π_{SS}, stably sorts $(\boldsymbol{b}[1], \boldsymbol{D}[1,1], \ldots, \boldsymbol{D}[1,e])$ $, \ldots, (\boldsymbol{b}[n], \boldsymbol{D}[n,1], \ldots, \boldsymbol{D}[n,e])$ to $(\boldsymbol{b}'[1], \boldsymbol{D}'[1,1], \ldots, \boldsymbol{D}'[1,e]), \ldots, (\boldsymbol{b}'[n], \boldsymbol{D}'[n,1], \ldots$ $, \boldsymbol{D}'[n,e])$ such that $\boldsymbol{b}'[i] \leq \boldsymbol{b}'[i+1]$ for $i \in \{1, \ldots, n-1\}$, and generates $[\![\boldsymbol{D}']\!]$ with the dealing algorithm of Π_{SS}. Finally, it outputs $[\![\boldsymbol{D}']\!]_{\mathcal{P}_i}$ to each $\mathcal{P}_i \in \mathbb{U}$.*

Algorithm 3 Binary key sort

Data: A shared vector $[\![\boldsymbol{b}]\!]$ of size n and an $n \times e$ shared matrix $[\![\boldsymbol{D}]\!]$, where $\boldsymbol{b} = \{0,1\}^n$.
Result: A shared matrix $[\![\boldsymbol{D}']\!]$, where $\boldsymbol{D}'$ is a stably sorted matrix $\boldsymbol{D}$ according to $\boldsymbol{b}$.
1: Parties agree on a field $\mathbb{Z}_Q$ s.t. $n \in \mathbb{Z}_Q$ and the bit-length of $\mathbb{Z}_Q$ is $\Theta(\log n)$.
▷ $\Theta(m \log n)$ com, $O(1)$ rnd
2: $[\![\boldsymbol{b}[i]]\!]^{(Q)} \leftarrow$ Bit-Field-Conversion($[\![\boldsymbol{b}[i]]\!]$) for $i \in \{1, \ldots, n\}$.
▷ $O(\ell m^3 n + m^3 n \log n)$ com, $O(1)$ rnd
3: Locally construct a matrix representation $[\![\boldsymbol{B}]\!]^{(Q)}$ of $[\![\boldsymbol{b}]\!]^{(Q)}$.
4: Compute the destination position of each item.
$[\![\boldsymbol{c}]\!]^{(Q)} \leftarrow$ Dest-Comp($[\![\boldsymbol{B}]\!]^{(Q)}$). ▷ $O(m^2 n \log n)$ com, $O(1)$ rnd
5: Reorder $\boldsymbol{D}$ according to the computed positions $\boldsymbol{c}$.
$\boldsymbol{c}', [\![\boldsymbol{D}']\!] \leftarrow$ Reveal-Sort($[\![\boldsymbol{c}]\!]^{(Q)}, [\![\boldsymbol{D}]\!]$). ▷ $O(\mathcal{M}e\ell m^2 n + \mathcal{M}m^2 n \log n)$ com, $O(\mathcal{M})$ rnd
6: **return** $[\![\boldsymbol{D}']\!]$.

The details of the algorithm are shown as Algorithm 3. In the first step, the parties agree on a field $\mathbb{Z}_Q$ s.t. $n \in \mathbb{Z}_Q$. This is required since we need to compute the destination position of each row, and the positions are elements in $\{1, \ldots, n\}$. Then, we convert the input binary keys to the shared values over the new field $\mathbb{Z}_Q$ in order to avoid overflow on elements in $\boldsymbol{c}$. To compute the destination positions, we first construct a matrix representation of the input binary keys. Since the input keys are binary, this is easy, as we mentioned in Section 4.1. Namely, we simply construct an $n \times 2$ matrix whose first column consists of negated keys and second column consists of the original keys. Finally, we compute the destination positions by applying Dest-Comp() then reorder the data according to the computed positions by applying Reveal-Sort().

Correctness Dest-Comp($[\![\boldsymbol{B}]\!]^{(Q)}$) computes the destination positions, and Reveal-Sort($[\![\boldsymbol{c}]\!]^{(Q)}, [\![\boldsymbol{D}]\!]$) orders $\boldsymbol{D}'$ according to the computed positions. Thus, $\boldsymbol{D}'$ is stably sorted.

Security Since Algorithm 3 consists of secure subprotocols, the only possibility of information leakage would be the vector $\boldsymbol{c}'$ revealed by Reveal-Sort($[\![\boldsymbol{c}]\!]^{(Q)}, [\![\boldsymbol{D}]\!]$). However, the correctness of Algorithm 3 guarantees that $\boldsymbol{c}' = (1, \ldots, n)^{\mathrm{T}}$ and leaks no additional information.

Theorem 3. *Algorithm 3 privately reduces* f^{Binary} *to* f^{BFC}, f^{DC}, *and* f^{RS}.

Proof. The view of adversaries consists of their inputs $[\![\boldsymbol{b}]\!]_{\mathbb{I}}$ and $[\![\boldsymbol{D}]\!]_{\mathbb{I}}$, random tapes, $[\![\boldsymbol{b}]\!]_{\mathbb{I}}^{(Q)}$, $[\![\boldsymbol{c}]\!]_{\mathbb{I}}^{(Q)}$, $\boldsymbol{c}'$, and $[\![\boldsymbol{D}']\!]_{\mathbb{I}}$. The output consists of $[\![\boldsymbol{D}']\!]_{\mathbb{I}}$. Note that adversaries have no view of the subprotocols Bit-Field-Conversion($\cdot$), Dest-Comp($\cdot$), or Reveal-Sort($\cdot$) since the execution of these protocols is substituted with the oracle invocation of functionalities f^{BFC}, f^{DC}, and f^{RS}, respectively.

We construct the simulator $\mathcal{S}$ as follows. Inputs and outputs are the same as those of adversaries, and $\mathcal{S}$ selects random tapes uniformly at random. For $\boldsymbol{c}'$, $\mathcal{S}$ sets $(1, \ldots, n)^{\mathrm{T}}$ as the simulated values for $\boldsymbol{c}'$. This perfectly simulates $\boldsymbol{c}'$ since the correctness of f^{RS} and f^{DC} guarantees that $\boldsymbol{c}'$ is always equal to $(1, \ldots, n)^{\mathrm{T}}$. For $[\![\boldsymbol{b}]\!]_{\mathbb{I}}^{(Q)}$ and $[\![\boldsymbol{c}]\!]_{\mathbb{I}}^{(Q)}$, $\mathcal{S}$ picks uniformly random numbers, and sets them as the simulated values for these shares. Since these shares are output shares of either Bit-Field-Conversion($\cdot$) or Dest-Comp($\cdot$), and these subprotocols output uniformly random shares, the above simulation is perfect.

Algorithm 4 Oblivious radix sort

Data: A shared vector $[\![\boldsymbol{k}]\!]$ of size n and an $n \times e$ shared matrix $[\![\boldsymbol{D}]\!]$.
Result: A shared matrix $[\![\boldsymbol{D}']\!]$, where $\boldsymbol{D}'$ is a stably sorted matrix $\boldsymbol{D}$ according to $\boldsymbol{k}$.

1: Parties agree on a field $\mathbb{Z}_Q$ s.t. $n \in \mathbb{Z}_Q$ and the bit-length of $\mathbb{Z}_Q$ is $\Theta(\log n)$.
▷ $\Theta(m \log n)$ com, $O(1)$ rnd
2: $[\![\boldsymbol{b}_1[i]]\!], \ldots, [\![\boldsymbol{b}_\ell[i]]\!] \leftarrow$ Bit-Decomposition($[\![\boldsymbol{k}[i]]\!]$) for $i \in \{1, \ldots, n\}$.
▷ $O(\ell^2 \log \ell m^2 n)$ com, $O(1)$ rnd
3: $[\![\boldsymbol{b}_j[i]]\!]^{(Q)} \leftarrow$ Bit-Field-Conversion($[\![\boldsymbol{b}_j[i]]\!]$) for $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, \ell\}$.
▷ $O(\ell^2 m^3 n + \ell m^3 n \log n)$ com, $O(1)$ rnd
4: Parties agree on a shared vector $[\![\boldsymbol{h}]\!]^{(Q)}$ s.t. $\boldsymbol{h} = (1, \ldots, n)^{\mathrm{T}}$. ▷ $O(\ell mn)$ com, $O(1)$ rnd
5: For simplicity, we use $[\![\boldsymbol{h}_1]\!]^{(Q)}$, $[\![\boldsymbol{b}'_1]\!]^{(Q)}$, $[\![\boldsymbol{b}_{\ell+1}]\!]^{(Q)}$ and $[\![\boldsymbol{b}'_{\ell+1}]\!]^{(Q)}$ as aliases of $[\![\boldsymbol{h}]\!]^{(Q)}$, $[\![\boldsymbol{b}_1]\!]^{(Q)}$, $[\![\boldsymbol{D}]\!]$ and $[\![\boldsymbol{D}']\!]$, respectively.
6: **for** $j = 1$ **to** ℓ **do**
7: Locally construct a matrix representation $[\![\boldsymbol{B}_j]\!]^{(Q)}$ of $[\![\boldsymbol{b}'_j]\!]^{(Q)}$.
8: $[\![\boldsymbol{c}_j]\!]^{(Q)} \leftarrow$ Dest-Comp($[\![\boldsymbol{B}_j]\!]^{(Q)}$). ▷ $O(m^2 n \log n)$ com, $O(1)$ rnd
9: $\boldsymbol{h}'_j, [\![\boldsymbol{c}'_j]\!]^{(Q)} \leftarrow$ Reveal-Sort($[\![\boldsymbol{h}_j]\!]^{(Q)}, [\![\boldsymbol{c}_j]\!]^{(Q)}$). ▷ $O(\mathcal{M} m^2 n \log n)$ com, $O(\mathcal{M})$ rnd
10: Pick up $[\![\boldsymbol{b}_{j+1}]\!]^{(Q)}$ and $[\![\boldsymbol{h}]\!]^{(Q)}$.
11: $\boldsymbol{c}''_j, [\![\boldsymbol{b}'_{j+1}]\!]^{(Q)} \| [\![\boldsymbol{h}_{j+1}]\!]^{(Q)} \leftarrow$ Reveal-Sort($[\![\boldsymbol{c}'_j]\!]^{(Q)}, [\![\boldsymbol{b}_{j+1}]\!]^{(Q)} \| [\![\boldsymbol{h}]\!]^{(Q)}$), where $\boldsymbol{x} \| \boldsymbol{y}$ represents horizontal concatenation of column vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. ▷ $O(\mathcal{M} m^2 n \log n)$ com, $O(\mathcal{M})$ rnd
▷ $O(\mathcal{M} e \ell m^2 n + \mathcal{M} m^2 n \log n)$ com, $O(\mathcal{M})$ rnd (if $j = \ell$)
12: **end for**
13: **return** $[\![\boldsymbol{b}'_{\ell+1}]\!]^{(Q)}$.

Thus, $\mathcal{S}$ perfectly simulates the view of adversaries. □

Complexity Communication and round complexities for each step are denoted as comments in Algorithm 3. Therefore, binary key sort exhibits $O(\mathcal{M} e \ell m^2 n + \mathcal{M} m^2 n \log n)$ communications in $O(\mathcal{M})$ rounds.

4.2 Oblivious Radix Sort

Now, we extend binary key sort and propose oblivious radix sort, whose input keys are not limited to being binary. We begin by defining the functionality f^{Radix} for oblivious radix sort.

Definition 11 (Oblivious radix sort). *On inputting* $([\![\boldsymbol{k}]\!]_{\mathcal{P}_i}, [\![\boldsymbol{D}]\!]_{\mathcal{P}_i})$ *from each* $\mathcal{P}_i \in \mathbb{U}$*, it reveals* $\boldsymbol{k}$ *and* $\boldsymbol{D}$ *with the revealing algorithm of* Π_{SS}*, stably sorts* $(\boldsymbol{k}[1], \boldsymbol{D}[1, 1], \ldots, \boldsymbol{D}[1, e]), \ldots, (\boldsymbol{k}[n], \boldsymbol{D}[n, 1], \ldots, \boldsymbol{D}[n, e])$ *to* $(\boldsymbol{k}'[1], \boldsymbol{D}'[1, 1], \ldots, \boldsymbol{D}'[1, e]), \ldots, (\boldsymbol{k}'[n], \boldsymbol{D}'[n, 1], \ldots, \boldsymbol{D}'[n, e])$ *such that* $\boldsymbol{k}'[i] \leq \boldsymbol{k}'[i + 1]$ *for* $i \in \{1, \ldots, n - 1\}$*, and generates* $[\![\boldsymbol{D}']\!]$ *with the dealing algorithm of* Π_{SS}*. Finally, it outputs* $[\![\boldsymbol{D}']\!]_{\mathcal{P}_i}$ *to each* $\mathcal{P}_i \in \mathbb{U}$.

Because binary key sort is stable, we can construct a sorting algorithm for arbitrary keys by using the idea of radix sort. That is, we can construct a sorting algorithm for arbitrary keys by applying bit-decomposition to the keys and iteratively applying binary key sort to the bitwise keys. However, this naive strategy requires $\Theta(\ell)$ overhead since the j-th significant bits of the keys are permuted $j - 1$ times before they are used as the sort key for the binary key sort.

We can use the pointer technique to avoid this overhead if we do not have to be concerned about privacy. Namely, we only permute small pointers that indicate the addresses of the data instead of the data. We construct a secure variant of the pointer technique by using the reveal sort algorithm again.

The proposed algorithm is shown as Algorithm 4. We first apply the bit-decomposition protocol to the key and convert the fields of the bitwise keys. Before applying the binary key sort, we prepare a shared vector $[\![\boldsymbol{h}]\!]^{(Q)}$, which stores the original positions of each row. This vector $[\![\boldsymbol{h}]\!]^{(Q)}$ is used like a pointer. Then we iteratively conduct the following procedure. At the beginning of the j-th iteration, we are given the j-th least significant bitwise key $[\![\boldsymbol{b}'_j]\!]^{(Q)}$ and an array $[\![\boldsymbol{h}_j]\!]^{(Q)}$, which stores original positions. The $[\![\boldsymbol{b}'_j]\!]^{(Q)}$ and $[\![\boldsymbol{h}_j]\!]^{(Q)}$ are stably sorted by $j-1$ less significant bits of the keys. We first compute $[\![\boldsymbol{c}_j]\!]^{(Q)}$, which indicates the destination positions after stable sort by j less significant bits of the keys. Then we reorder the computed positions $[\![\boldsymbol{c}_j]\!]^{(Q)}$ to the original order with the reveal sort, and pick up the $(j+1)$th bitwise keys $[\![\boldsymbol{b}_{j+1}]\!]^{(Q)}$ and original positions $[\![\boldsymbol{h}]\!]^{(Q)}$. Finally, we apply the reveal sort to reorder the $(j+1)$th bitwise keys $[\![\boldsymbol{b}_{j+1}]\!]^{(Q)}$ and original positions $[\![\boldsymbol{h}]\!]^{(Q)}$ according to the computed destination positions $[\![\boldsymbol{c}'_j]\!]^{(Q)}$ and we obtain $(j+1)$th bitwise key $[\![\boldsymbol{b}'_{j+1}]\!]^{(Q)}$ and original positions $[\![\boldsymbol{h}_{j+1}]\!]^{(Q)}$ sorted by j less significant bits of the keys.

Correctness We claim that the condition "$\boldsymbol{b}'_{j+1}$ is a stably sorted vector of $\boldsymbol{b}_{j+1}$ according to $\boldsymbol{b}_j, \ldots, \boldsymbol{b}_1$ at the end of the j-th iteration," which we call C_j, holds for every $j \in \{1, \ldots, \ell\}$. We first assume that the condition "$\boldsymbol{b}'_j$ is a stably sorted vector of $\boldsymbol{b}_j$ according to $\boldsymbol{b}_{j-1}, \ldots, \boldsymbol{b}_1$ at the beginning of the j-th iteration," which we call D_j, holds for a $j \in \{1, \ldots, \ell\}$. Then, Dest-Comp($\boldsymbol{b}'_j$) computes the positions $\boldsymbol{c}_j$ after we stably sort them by $\boldsymbol{b}'_j$. $\boldsymbol{c}_j$ represents the positions of rows after the data are sorted by lower j bits since $\boldsymbol{b}'_j$ is stably sorted by $\boldsymbol{b}_{j-1}, \ldots, \boldsymbol{b}_1$. In addition, $\boldsymbol{h}_j[i]$ represents the initial position of $\boldsymbol{b}'_j[i]$ for $i \in \{1, \ldots, n\}$. Therefore, we correctly find the next corresponding key. Thus, if D_j holds, C_j also holds. Since D_{j+1} is immediate from C_j, and D_1 holds, C_ℓ also holds. That is, $\boldsymbol{b}'_{\ell+1}$ is stably sorted.

Security Since Algorithm 4 consists of secure subprotocols, the only possibility of information leakage would be the vectors revealed by Reveal-Sort($\cdot$). However, all the revealed vectors are equal to $(1, \ldots, n)^{\mathrm{T}}$. Therefore, Algorithm 4 leaks no additional information.

Theorem 4. *Algorithm 4 privately reduces f^{Radix} to f^{BD}, f^{BFC}, f^{DC}, and f^{RS}.*

Proof. The view of adversaries consists of their inputs $[\![\boldsymbol{k}]\!]_{\mathbb{I}}$ and $[\![\boldsymbol{D}]\!]_{\mathbb{I}}$, random tapes, $[\![\boldsymbol{b}_j]\!]_{\mathbb{I}}$, $[\![\boldsymbol{b}_j]\!]_{\mathbb{I}}^{(Q)}$, $[\![\boldsymbol{c}_j]\!]_{\mathbb{I}}^{(Q)}$, $\boldsymbol{h}'_j$, $[\![\boldsymbol{c}'_j]\!]_{\mathbb{I}}^{(Q)}$, $\boldsymbol{c}''_j$, $[\![\boldsymbol{b}'_{j+1}]\!]_{\mathbb{I}}^{(Q)}$, and $[\![\boldsymbol{h}_{j+1}]\!]_{\mathbb{I}}^{(Q)}$. The output consists of $[\![\boldsymbol{D}']\!]_{\mathbb{I}}$. Note that adversaries have no view of the subprotocols Bit-Decomposition($\cdot$), Bit-Field-Conversion($\cdot$), Dest-Comp($\cdot$), or Reveal-Sort($\cdot$) since the execution of these protocols is substituted with the oracle invocation of functionalities f^{BD}, f^{BFC}, f^{DC}, and f^{RS}, respectively.

We construct the simulator $\mathcal{S}$ as follows. Inputs and outputs are the same as those of adversaries, and $\mathcal{S}$ selects random tapes uniformly at random. For $\boldsymbol{h}'_j$ and $\boldsymbol{c}''_j$, $\mathcal{S}$ sets $(1, \ldots, n)^{\mathrm{T}}$ as the simulated values for $\boldsymbol{h}'_j$ and $\boldsymbol{c}''_j$. This perfectly simulates $\boldsymbol{h}'_j$ and $\boldsymbol{c}''_j$

Table 4. Running time of sorting schemes in seconds. n represents number of input values. "N/A" means that execution did not finish in $3,600$ seconds.

Sorting scheme	$n = 10$	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
Batcher's merge sort [2]	1.164	3.623	20.736	121.214	1100.698	N/A	N/A
Randomized shellsort [15]	5.259	77.193	819.018	N/A	N/A	N/A	N/A
Oblivious keyword sort [30]	0.242	2.601	198.996	N/A	N/A	N/A	N/A
Quicksort [17]	0.161	0.452	1.512	9.124	72.793	964.619	N/A
Proposed algorithm	0.143	0.171	0.488	2.290	16.827	196.316	2648.398

since the correctness of f^{RS} and f^{DC} guarantees that these vectors are always equal to $(1, \ldots, n)^{\mathrm{T}}$. For $[\![\boldsymbol{b}_j]\!]_{\mathbb{I}}$, $[\![\boldsymbol{b}_j]\!]_{\mathbb{I}}^{(Q)}$, $[\![\boldsymbol{c}_j]\!]_{\mathbb{I}}^{(Q)}$, $[\![\boldsymbol{c}'_j]\!]_{\mathbb{I}}^{(Q)}$, $[\![\boldsymbol{b}'_{j+1}]\!]_{\mathbb{I}}^{(Q)}$, and $[\![\boldsymbol{h}_{j+1}]\!]_{\mathbb{I}}^{(Q)}$, except for $[\![\boldsymbol{b}'_{\ell+1}]\!]_{\mathbb{I}}^{(Q)}$, $\mathcal{S}$ selects uniformly random numbers and sets them as the simulated values for these shares. Since these shares are output shares of either Bit-Decomposition$(\cdot)$, Bit-Field-Conversion$(\cdot)$, Dest-Comp$(\cdot)$, or Reveal-Sort$(\cdot)$, and these subprotocols output uniformly random shares, the above simulation is perfect.

Thus, $\mathcal{S}$ perfectly simulates the view of adversaries. □

Complexity Communication and round complexities for each step are denoted as comments in Algorithm 4. The iteration is conducted ℓ times where ℓ represents the bit-length of the field size. Therefore, Algorithm 4 exhibits $O(\ell^2 \log \ell m^2 n + \ell^2 m^3 n + \mathcal{M}\ell m^2 n \log n + \mathcal{M}e\ell m^2 n)$ communications in $O(\mathcal{M}\ell)$ rounds. If we assume that the number of parties m is constant and the field bit-length ℓ is constant, Algorithm 4 exhibits $O(n \log n)$ communications in $O(1)$ rounds.

5 Evaluation

5.1 Complexity Analysis

We first evaluated our oblivious radix sort algorithm from an asymptotic perspective. The complexities of our algorithm and existing sorting protocols are summarized in Table 1. When the number of parties m and the bit length of the field ℓ are assumed to be constant, our oblivious radix sort exhibits a very efficient complexity of $O(1)$ rounds and $O(n \log n)$ communications. Thus, our oblivious radix sort is asymptotically very efficient in the above setting.

5.2 Experimental Results

To demonstrate practical efficiency of our oblivious radix sort, we implemented our oblivious radix sort and existing sorting algorithms for comparison. The AKS sorting network [1] was not implemented since this algorithm is not of practical interest. We implemented sorting algorithms on $(2, 3)$-Shamir's secret-sharing scheme with corruption tolerance $t = 1$. For better performance, protocols that are secure against a semi-honest adversary are implemented. We implemented the comparison protocol proposed by Damgård et al. [9] as a building block of existing sorting algorithms. The oblivious

radix sort algorithm used the shuffling protocol proposed by Laur et al. [21] and the bit-decomposition protocol proposed by Damgård et al. [9]. The bit-field-conversion protocol was not implemented since the size of the field was large enough to deal with the number of input values n in our experiment. Our implementation of the randomized shellsort and Batcher's merge sort algorithms is based on circuit representations. That is, we replaced the comparators in the original algorithms with comparator protocols constructed by combining comparisons, multiplications, and additions. All values are elements over a finite field $\mathbb{Z}_P = \{0, 1, \ldots, P-1\}$, where P is a prime number $4294967291 = 2^{32} - 5$ and satisfies $2^{31} < P < 2^{32}$. That is, the values are 32-bit words.

Experiments were conducted on three laptop machines with an Intel Core i7-2640M 2.8-GHz CPU and 8 GB of physical memory. These three machines were connected to a 1-Gbps LAN. The implementation was written in C++, and g++ 4.6.3 was used for compiling. The running times of the sorting algorithms are given in Table 4, and an intuitive graph is shown in Fig.1. The results show that our oblivious radix sort outperforms existing sorting algorithms in the above setting.

6 Conclusion

We proposed a simple and efficient oblivious stable sorting algorithm for MPC. The complexities of the proposed algorithm can be considered as $O(n \log n)$ communications and $O(1)$ rounds if we consider only the asymptotic dependency of the number of input values n.

We also showed the practical efficiency of the proposed algorithm. The feasibility of our sorting algorithm was demonstrated by means of an implementation on an MPC scheme based on $(2, 3)$-Shamir's secret-sharing scheme with corruption tolerance $t = 1$. Our implementation sorted 1 million 32-bit word secret-shared values in 197 seconds, and outperformed other existing sorting algorithms on the above MPC scheme.

References

1. Ajtai, M., Komlós, J., Szemerédi, E.: An O(n log n) sorting network. In: STOC. pp. 1–9. ACM (1983)
2. Batcher, K.E.: Sorting networks and their applications. In: AFIPS Spring Joint Computing Conference. pp. 307–314 (1968)
3. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: Simon [28], pp. 1–10
4. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: ESORICS. LNCS, vol. 5283, pp. 192–206. Springer (2008)
5. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Financial Cryptography. LNCS, vol. 5628, pp. 325–343. Springer (2009)
6. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.A.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: USENIX Security Symposium. pp. 223–240. USENIX Association (2010)

7. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: Simon [28], pp. 11–19
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge, MA, second edn. (2001)
9. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: TCC. pp. 285–304 (2006)
10. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: TCC. LNCS, vol. 6597, pp. 144–163. Springer (2011)
11. Dodis, Y., Yampolskiy, A., Yung, M.: Threshold and proactive pseudo-random permutations. In: TCC. Lecture Notes in Computer Science, vol. 3876, pp. 542–560. Springer (2006)
12. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In: PODC. pp. 101–111. ACM (1998)
13. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)
14. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC. pp. 218–229. ACM (1987)
15. Goodrich, M.T.: Randomized shellsort: A simple oblivious sorting algorithm. In: SODA. pp. 1262–1277 (2010)
16. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: ICALP (2). Lecture Notes in Computer Science, vol. 6756, pp. 576–587. Springer (2011)
17. Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: ICISC. LNCS, vol. 7839, pp. 202–216. Springer (2012)
18. Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: NDSS (2012)
19. Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. IACR Cryptology ePrint Archive 2011, 122 (2011)
20. Knuth, D.E.: Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition), chap. 5. Addison-Wesley Professional (1998)
21. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: ISC. LNCS, vol. 7001, pp. 262–277. Springer (2011)
22. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - secure two-party computation system. In: USENIX Security Symposium. pp. 287–302 (2004)
23. Ning, C., Xu, Q.: Multiparty computation for modulo reduction without bit-decomposition and a generalization to bit-decomposition. In: ASIACRYPT. pp. 483–500 (2010)
24. Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In: PKC. pp. 343–360 (2007)
25. Obana, S., Araki, T.: Almost optimum secret sharing schemes secure against cheating for arbitrary secret distribution. In: ASIACRYPT. LNCS, vol. 4284, pp. 364–379. Springer (2006)
26. Ogata, W., Kurosawa, K.: Optimum secret sharing scheme secure against cheating. In: EUROCRYPT. LNCS, vol. 1070, pp. 200–211. Springer (1996)
27. Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)
28. Simon, J. (ed.): Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA. ACM (1988)
29. Wang, G., Luo, T., Goodrich, M.T., Du, W., Zhu, Z.: Bureaucratic protocols for secure two-party sorting, selection, and permuting. In: ASIACCS. pp. 226–237 (2010)
30. Zhang, B.: Generic constant-round oblivious sorting algorithm for MPC. In: ProvSec. LNCS, vol. 6980, pp. 240–256. Springer (2011)