Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms

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

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

Abstract. Sorting is one of the most important primitives in various systems, for example, database systems, since it is often the dominant operation in the running time of an entire system. Therefore, there is a long list of work on improving its efficiency. It is also true in the context of secure multi-party computation (MPC), and several MPC sorting protocols have been proposed. However, all existing MPC sorting protocols are based on less efficient sorting algorithms, and the resultant protocols are also inefficient. This is because only a method for converting data-oblivious algorithms to corresponding MPC protocols is known, despite the fact that most efficient sorting algorithms such as quicksort and merge sort are not data-oblivious. We propose a simple and general approach of converting non-data-oblivious comparison sort algorithms, which include the above algorithms, into corresponding MPC protocols. We then construct an MPC sorting protocol from the well known efficient sorting algorithm, quicksort, with our approach. The resultant protocol is *practically* efficient since it significantly improved the running time compared to existing protocols in experiments.

Keywords: Multi-party protocol, sorting, comparison sort, secret sharing, unconditional security.

1 Introduction

With the growth in information technology, the use of personal data is also increasing. Therefore, awareness concerning privacy issues has been growing, and systems that use sensitive data without breaching privacy are needed. Secure multi-party computation (MPC) is a technique that enables the creation of such secure systems, and frameworks, such as FairplayMP [3], Sharemind [6], SEPIA [7], TASTY [17], and VIFF [13], have been implemented. MPC protocols allow a set of participants (*parties*) to compute a function privately. That is, when a function is represented as $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$, each party with its private input x_i obtains only the output y_i and nothing else. In a typical MPC framework, input and output values are in secret-shared form. Namely, x_i and y_i are the shares of input and output values, respectively. Although any function can be computed securely by using a circuit representation of the function [4,15], it is not easy to design practically efficient MPC protocols for complex algorithms, such as database operations. Therefore, proposals have been made to

© Springer-Verlag Berlin Heidelberg 2013

construct specific and efficient MPC protocols as building blocks, e.g., computing bit-decomposition and comparison [10,25], and modulo reduction [24].

Sorting is one of the most important primitives in various systems, for example, database systems, since it is frequently conducted and comparatively time-consuming. The importance of a sorting algorithm is known, and there is a long list of work on improving its efficiency. To obtain a practically efficient sorting algorithm, researchers not only investigated computational complexity but also experimental performance. Although computational complexity is a good asymptotic metric of efficiency, sometimes an inferior (in the sense of computational complexity) sorting algorithm exceeds the experimental performance of superior ones. For example, quicksort is more popular than merge sort since quicksort often performs better even though its computational complexity is worse than that of the merge sort algorithm. One of the most famous classes of sorting is *comparison sorts*. A comparison sort determines the sorted order based only on comparisons between the input elements. Comparison sorts include a number of well-known and efficient sorting algorithms, such as quicksort, shell sort, heapsort, and merge sort.

In the context of MPC protocols, sorting is also a very important primitive. MPC sorting protocols are often required in various database operations and have many applications such as cooperative IDS [20], oblivious RAM [12] and private set intersection [19]. Therefore, a number of MPC sorting protocols has been proposed [16,3,20,32]. However, they are based on less efficient sorting algorithms, and the resultant protocols are also inefficient. One of the main causes is the obstacle in constructing MPC protocols.

1.1 Obstacle for Using Well-known Algorithms

We say that an algorithm is *data-dependent* if the control flow of the algorithm depends on data values, and an algorithm that is not data-dependent is said to be *data-oblivious*. Generally speaking, there is a large obstacle when one constructs a practically efficient MPC protocol from a well-known algorithm. That is, MPC protocols should be data-oblivious while most efficient algorithms are not. Furthermore, how to convert data-dependent algorithms to data-oblivious algorithms is not known.

To illustrate this obstacle during the conversion from data-dependent algorithms to MPC protocols, let us consider the following two algorithms. Both algorithms receive a sequence of values $a_1, \ldots, a_m \in \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, where p is a prime, as input, and the output is the number of non-zero values in $a_1, \ldots, a_m$. [1]

```
CountNonZero1($a_1, \ldots, a_m$):
1: $c = 0$.
2: for $i = 1$ to $m$ do
3:     $c = c + ((a_i)^{p-1} \mod p)$.
4: return $c$.
```

```
CountNonZero2($a_1, \ldots, a_m$):
1: $c = 0$.
2: for $i = 1$ to $m$ do
3:     if $a_i \neq 0$ then
4:         $c = c + 1$.
5: return $c$.
```

[1] $(a_i)^{p-1} \mod p = \begin{cases} 0 \text{ if } a_i = 0 \\ 1 \text{ otherwise} \end{cases}$ holds by Fermat's little theorem.

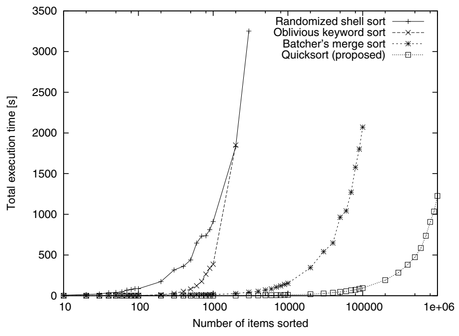

Fig. 1. Running time of four compared sorting implementations. Number of elements on x-axis is on log-scale.

The running time of the first algorithm is $O(m \log p)$ since $(a_i)^{p-1} \mod p$ is computed with $O(\log p)$ multiplications over $\mathbb{Z}_p$ by using the exponentiation by squaring technique, and that of the second algorithm is $O(m)$. Therefore, the second algorithm seems more efficient than the first one.

Next, let us consider the case when we convert these algorithms to MPC protocols. For the first algorithm, we need only minor modifications: We replace the values $a_1, \ldots, a_m$ and c with secret-shared values (or values in other forms depending on the MPC environment), and replace operations applied to these values, such as additions and multiplications, with corresponding MPC subprotocols. [2]

The resulting protocol requires only $O(m \log p)$ invocations of subprotocols. For the second algorithm, it is not enough to apply the same modifications as the first one since the second algorithm has an **if** condition, and the result of the **if** condition discloses the information that $a_i = 0$ or not. Even if the result is hidden, the branch of subsequent processes discloses the information. To avoid these disclosures naively, we have to execute both cases of the **if** condition. Therefore, the resulting protocol requires $\Omega(2^m)$ invocations of subprotocols.

This significant difference between the complexities of the converted protocols is due to the fact that the first algorithm is data-oblivious while the second algorithm is data-dependent. Thus, the naive method used to convert data-oblivious algorithms to MPC protocols does not work when the algorithm is data-dependent.

Above obstacle also occurs in the area of sorting. Therefore, all existing MPC sorting protocols are based on specific sorting algorithms, which are data-oblivious but less efficient. This is one of the main causes of the large gap on efficiency between MPC sorting protocols and well known sorting algorithms.

1.2 Contributions

In this paper we show that in the areas of comparison sort one can efficiently convert data-dependent algorithms to MPC protocols with a simple approach. Furthermore, we

[2] We have no need for applying expensive exponentiation protocols since p is a public constant value.

propose a practically efficient MPC sorting protocol from the well known sorting algorithm quicksort. Note that we discriminate *protocol* and *algorithm* such that the former is used in a multi-party sense and the other is in an ordinal one, and say an algorithm or protocol is *practically efficient* if it not only has less computational complexity but also delivers good experimental results.

When trying to convert comparison sort algorithms to MPC protocols, an obstacle to conversion for data-dependent algorithms occurs: The next pair of elements to be compared depends on the outcome of previous comparisons in sorting algorithms. Therefore, well known and practically efficient comparison sort algorithms, such as quicksort, have not been applied to MPC protocols.

To overcome the above obstacle, we use a simple approach of *shuffling before sorting*. That is, the parties first shuffle the input (in an MPC sense) and then use a comparison sort algorithm, e.g., quicksort or merge sort, with minor modifications on the shuffled secret-shared values. Roughly speaking, although the data-dependent comparison leaks the order of compared elements, the order is randomized by the shuffling and has no relation to the inputs of the protocol. Therefore, we can straightforwardly construct MPC sorting protocols from comparison sort algorithms after shuffling.

We next show that our approach can construct a practically efficient MPC sorting protocol. We concretely construct an MPC sorting protocol from the quicksort algorithm with our approach. Our protocol uses $O(m \log m)$ comparisons in $O(\log m)$ rounds on average, which are comparable to other existing protocols. We describe a precise complexity comparison in Sect. 4. Furthermore, we implement the proposed quicksort protocol and other existing sorting protocols [2,31,32] on $(2,3)$-Shamir's secret-sharing scheme with corruption tolerance $t = 1$. This setting is reasonable since our aim is to produce a practically efficient sorting protocol and the performance of MPC protocols does not scale well based on the number of parties. As a result, our proposed quicksort protocol sorts 32-bit words and $1,000,000$ secret-shared values in $1,227$ seconds, while existing sorting protocols cannot sort within $3,600$ seconds. We describe an intuitive graph in Fig. 1 and precise experimental results in Sect. 4.

1.3 Related Work

Some circuit-based sorting algorithms are known as sorting networks. Since sorting networks are constructed in a circuit style and circuit-based algorithms are data-oblivious, they can be efficiently applied to MPC protocols. Ajtai et al. proposed an asymptotically optimal sorting network known as the AKS sorting network, which exhibits a complexity of $O(m \log m)$ comparisons, where m is the number of input shares [1]. However, this algorithm is not practical since its constant factor is very high. On the other hand, Batcher's merge sort [2] is more efficient unless m is quite large [21]. This algorithm exhibits a complexity of $O(m \log^2 m)$ comparisons with a lower constant factor.

Goodrich proposed a data-oblivious sort called randomized shell sort [16]. Similar to sorting networks, data-oblivious sorts are also efficiently applied to MPC protocols. Although randomized shell sort returns a wrong output with low probability, it exhibits a complexity of $O(m)$ rounds and $O(m \log m)$ comparisons.

Wang et al. reported experimental results of some sorting algorithms [31]. Their implementation is based on the MPC system Fairplay [23]. The running times of Batcher's merge sort [2] and randomized shell sort [16] for 256 input values are approximately $3,000$ and $6,200$ seconds, respectively.

Jónsson et al. studied a general technique to hide the number of input values for sorting protocols [20]. They also implemented Batcher's merge sort [2] and other sorting protocols on the MPC system Sharemind [6]. Their implementation is optimized using a technique called vectorization, and the vectorized Batcher's merge sort sorts $16,384$ secret shared values in 210 seconds.

Zhang proposed data-oblivious sorting algorithms [32] based on bead sort. All of Zhang's algorithms exhibit complexities of constant rounds and $O(Rm)$ or $O(m^2)$ comparisons depending on the algorithm, where R represents the range of input values. Since these algorithms are data-oblivious, we can convert them to multi-party sorting protocols by using a circuit-based technique while keeping their complexities.

2 Preliminaries

2.1 Assumptions and Notations

We focus on secret-sharing-based MPC. For simplicity, n parties $P_1, \ldots, P_n$ are connected by secure channels. All values used in secret-sharing schemes belong to a field K. We use $[\![s]\!]_{P_i}$ to denote a *share* for P_i where a *secret value* is $s \in K$. Let $\mathbb{Q}$ be a coalition of parties and $[\![s]\!]_{\mathbb{Q}}$ denote a set of shares $\{[\![s]\!]_{P_i} \mid P_i \in \mathbb{Q}\}$. When $\mathbb{U}$ represents all parties, we simply denote $[\![s]\!]_{\mathbb{U}}$ as $[\![s]\!]$ and call it *shared values*. We call some elements related to secret-sharing scheme as follows;

- s: secret value,
- $[\![s]\!]_{P_i}$: share (for a party P_i),
- $[\![s]\!] = [\![s]\!]_{\mathbb{U}} = \{[\![s]\!]_{P_1}, \ldots, [\![s]\!]_{P_n}\}$: shared value.

We use $[i]$ to denote a set $\{1, 2, \ldots, i\}$.

2.2 Security Model

We consider unconditional, perfect security against a semi-honest adversary with static corruption of at most t. This means that the adversary can execute unbounded computation, must follow a protocol, and can corrupt at most t parties only before the protocol is conducted. More technically, 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 $\mathbb{I} = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\} \subset \mathbb{U}$ to denote the parties that are corrupted. Due to space limitation, the formal definition of the security against a semi-honest adversary with static corruption appears in Appendix A.

2.3 Complexity Metrics in MPC

We use two metrics, *round complexity* and *the number invocations of the comparison protocol*, 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. Because the comparison protocol is a dominant factor of the complexity of communications, we measure the amount of data transmitted by the parties with the number of invocations of the comparison protocol.

2.4 Secret-Sharing Scheme

We focus on a class of secret-sharing schemes called (k, n)-threshold. This means that the shares are shared by the n parties in such a way that any coalition of k or more parties can together reconstruct the secret, but no coalition fewer than k parties can. Shamir's secret-sharing scheme [28] belongs to this class. We assume that the corruption tolerance t satisfies $t < \min(k, n - k)$. We say $[\![s]\!]$ is *uniformly random* if it is uniformly randomly chosen from the set of possible shared values whose secret value is s.

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. The revealing algorithm takes at least k shares and outputs the secret value s.

2.5 Shuffling, Comparison, and Reveal Protocols

We introduce some existing MPC protocols used as building blocks of our protocol.

Our protocols 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 [8]. In this paradigm, secret values are preliminary shared with a secret-sharing scheme to all parties that participate in MPC protocols. Then MPC protocols take secret-shared values as inputs from each party and output the result in secret-shared form. The result is finally recovered by the revealing algorithm of the secret-sharing scheme.

Comparison protocol. The comparison protocol [10,25] receives two shared values and outputs a shared value of the comparison result of the inputs. More precisely, the comparison protocol accepts $[\![a]\!]_{P_i}, [\![b]\!]_{P_i}$ from each $P_i \in \mathbb{U}$ as input and outputs $[\![c]\!]_{P_i}$ to each $P_i \in \mathbb{U}$ such that $c = 1$ if $a \leq b$ and $c = 0$ otherwise. We assume that K is totally ordered and denote this protocol as "$[\![c]\!] \leftarrow [\![a \leq b]\!]$". We formally define the comparison protocol with the following function $f^{\mathsf{CMP}}_{\Pi_{\mathsf{SS}}}$.

$f^{\mathsf{CMP}}_{\Pi_{\mathsf{SS}}}$: *On inputting* $[\![x]\!]_{P_i}$ *and* $[\![y]\!]_{P_i}$ *from each* $P_i \in \mathbb{U}$*, it reveals* x *and* y *with the revealing algorithm of* Π_{SS}*, sets* $z = 1$ *if* $x \leq y$ *and* $z = 0$ *otherwise, and generates* $[\![z]\!]$ *with the dealing algorithm of* Π_{SS}*. Finally, it outputs* $[\![z]\!]_{P_i}$ *to each* $P_i \in \mathbb{U}$*.*

The comparison protocol proposed by Nishide and Ohta [25] exhibits the complexity of $O(1)$ rounds and $O(\ell)$ invocations of multiplication protocols where ℓ is the bit-length of K.

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

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

We formally define the shuffling protocol with the following function $f^{\mathsf{Shuffle}}_{\Pi_{\mathsf{SS}}}$.

$f^{\mathsf{Shuffle}}_{\Pi_{\mathsf{SS}}}$: *On inputting* $([\![a_1]\!]_{P_i}, \ldots, [\![a_m]\!]_{P_i})$ *from each* $P_i \in \mathbb{U}$*, it reveals* $a_1, \ldots, a_m$ *with the revealing algorithm of* Π_{SS}*, selects a permutation* $\pi : [m] \to [m]$ *uniformly* *at random, sets* $b_i = a_{\pi(i)}$ *for* $i \in [m]$*, and generates* $[\![b_1]\!], \ldots, [\![b_m]\!]$ *with the dealing algorithm of* Π_{SS}*. Finally, it outputs* $([\![b_1]\!]_{P_i}, \ldots, [\![b_m]\!]_{P_i})$ *to each* $P_i \in \mathbb{U}$*.*

Laura et al. proposed efficient shuffling protocols [22]. One of their protocols exhibits the complexity of $O(2^n/\sqrt{n})$ rounds and $O(2^n n^{3/2} m \log m)$ communications. When the number of parties is constant, it exhibits $O(1)$ rounds and $O(m \log m)$ communications. We use this protocol as the shuffling protocol.

Reveal protocol. The reveal protocol accepts $[\![x]\!]_{P_i}$ from each $P_i \in \mathbb{U}$ and outputs x to each $P_i \in \mathbb{U}$. This protocol just has a role of the reveal algorithm in a multi-party setting. A run of this protocol is denoted as

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

We formally define the reveal protocol with the following function $f_{\Pi_{\mathsf{SS}}}^{\mathsf{Reveal}}$.

$f_{\Pi_{\mathsf{SS}}}^{\mathsf{Reveal}}$: *On inputting* $[\![x]\!]_{P_i}$ *from each* $P_i \in \mathbb{U}$*, it reveals* x *with the revealing algorithm of* Π_{SS} *and outputs* x *to each* $P_i \in \mathbb{U}$*.*

The reveal protocol can be easily constructed in a semi-honest model by distributing all shares among all parties. Even in the malicious model it can be constructed by using secret-sharing schemes *secure against cheating* [27,26].

3 MPC Sorting Protocols

In this section, we propose an approach of constructing efficient sorting protocols, and then we apply our approach to the quicksort algorithm. For simplicity, we split the construction of our quicksort protocol with two steps: we begin by describing the construction with restricted inputs and later show how to remove this restriction. We also discuss further extensions of our approach.

We assume that the following protocols can be executed on Π_{SS}; shuffling, comparison, and reveal. For example, Shamir's secret-sharing scheme satisfies this condition.

3.1 Our Approach of Constructing Efficient Sorting Protocols

To construct an efficient sorting protocol, it is natural to try to construct an MPC sorting protocol that emulates practically efficient sorting algorithms. However, this approach has to solve a certain problem; When trying to convert well-known sorting algorithms to MPC protocols, the problem with most practically efficient sorting algorithms is that they are data-dependent. On the other hand, if an MPC protocol changes its behavior according to the input, it might violate privacy. Therefore, all existing sorting protocols use less efficient data-oblivious sorting algorithms. Consequently, we have to fill the gap between data-dependency and data-obliviousness to construct MPC sorting protocols from well-known sorting algorithms.

Sorting algorithms which determine the sorted order based only on comparisons between the input elements are called comparison sorts. Comparison sorts include a number of practically efficient sorting algorithms, such as quicksort, shell sort, heapsort, insertion sort, and merge sort. However, comparison sorts are essentially data-dependent since the next pair of elements to be compared depends on the outcome of previous comparisons. Therefore no comparison sort, including well known quicksort algorithm, has been applied to MPC protocols.

To solve the above problem, we use a simple approach of *shuffling before sorting*. Our approach consists of the following modifications to the original comparison sort algorithm.

1. We apply the shuffling protocol to the inputs at the first step.
2. We execute as the same to the original (data-dependent) comparison sort algorithm, except to replace the comparison operation with a continuous execution of comparison and reveal protocols.

In the execution of the protocol, the revealed result of comparison seems to leak ordinal information. However, the ordinal information is randomized by the shuffling at the first step, so it leaks no information about true inputs. This approach is quite simple and effective for constructing practically efficient sorting protocols. Our approach is also quite general since, to our knowledge, all of practically efficient comparison sort algorithms can be converted to MPC protocols with our approach.

3.2 Quicksort Protocol

Now, we concretely construct an MPC sorting protocol, which we call quicksort protocol, from the quicksort algorithm with our approach. Note that we assume that the secret values of inputs are distinct here and discuss the unrestricted input case in the following subsection.

The sorting function is defined as follows.

$f_{\Pi_{\mathsf{SS}}}^{\mathsf{Sorting}}$: *On inputting* $([\![a_1]\!]_{P_i}, \ldots, [\![a_m]\!]_{P_i})$ *from each* $P_i \in \mathbb{U}$, *it reveals* $a_1, \ldots, a_m$ *with the reveal algorithm of* Π_{SS}, *sorts* $(a_1, \ldots, a_m)$ *to* $(b_1, \ldots, b_m)$ *such that* $b_i \leq b_{i+1}$ *for* $i \in [m-1]$, *and generates* $[\![b_1]\!], \ldots, [\![b_m]\!]$ *with the dealing algorithm of* Π_{SS}. *Finally, it outputs* $([\![b_1]\!]_{P_i}, \ldots, [\![b_m]\!]_{P_i})$ *to each* $P_i \in \mathbb{U}$.

We describe our quicksort protocol constructed by applying our approach in Protocol 1. Next we discuss the property of our quicksort protocol.

Correctness. Our quicksort protocol has two differences compared to the original quicksort algorithm. The first difference is comparison; however, this has no effect on execution since the replicated protocols simply emulate the original. The second difference is an additional shuffling step inserted at the beginning of our quicksort protocol. Since the secret values of the input shared values are distinct, the order of the secret values of the output is unique. Therefore, the first shuffling step does not affect the results.

Security. Roughly speaking, the shuffling and comparison protocols are secure, and the swapping operation is just a local computation. Therefore, the only possible information leakage is the revealed results from the comparisons. However, the results of each comparison have no relation to the input. This is because the input shared values are shuffled in the first step by the shuffling protocol. We formally claim the following theorem.

Theorem 1. *Protocol 1 t-privately reduces* $f_{\Pi_{\mathsf{SS}}}^{\mathsf{Sorting}}$ *to* $f_{\Pi_{\mathsf{SS}}}^{\mathsf{Shuffle}}$, $f_{\Pi_{\mathsf{SS}}}^{\mathsf{CMP}}$, *and* $f_{\Pi_{\mathsf{SS}}}^{\mathsf{Reveal}}$.

The proof of the theorem appears in Appendix B.

Protocol 1. Quicksort protocol

Notation: $[\![b_1]\!], \dots, [\![b_m]\!] \leftarrow \mathsf{Quicksort}([\![a_1]\!], \dots, [\![a_m]\!])$
Input: Shared values $[\![a_1]\!], \dots, [\![a_m]\!]$.
Output: Shared values $[\![b_1]\!], \dots, [\![b_m]\!]$ where $b_1 \le \cdots \le b_m$.

```
1: Unless this is a recursively called execution, apply the shuffling protocol to $[\![a_1]\!], \dots, [\![a_m]\!]$.
2: if $1 < m$ then
3:     $p, [\![e_1]\!], \dots, [\![e_m]\!] \leftarrow \mathsf{Partition}([\![a_1]\!], \dots, [\![a_m]\!])$.
4:     $[\![b_1]\!], \dots, [\![b_{p-1}]\!] \leftarrow \mathsf{Quicksort}([\![e_1]\!], \dots, [\![e_{p-1}]\!])$.
5:     Let $[\![b_p]\!] = [\![e_p]\!]$.
6:     $[\![b_{p+1}]\!], \dots, [\![b_m]\!] \leftarrow \mathsf{Quicksort}([\![e_{p+1}]\!], \dots, [\![e_m]\!])$.
7: else
8:     Let $([\![b_1]\!], \dots, [\![b_m]\!]) = ([\![a_1]\!], \dots, [\![a_m]\!])$.
9: return $[\![b_1]\!], \dots, [\![b_m]\!]$.
```

Notation: $p, [\![e_1]\!], \dots, [\![e_m]\!] \leftarrow \mathsf{Partition}([\![a_1]\!], \dots, [\![a_m]\!])$
Input: Shared values $[\![a_1]\!], \dots, [\![a_m]\!]$.
Output: Position p and shared values $[\![e_1]\!], \dots, [\![e_m]\!]$.

```
1: Let $i = 0$.
2: for $j = 1$ to $m - 1$ do
3:     $[\![c]\!] \leftarrow [\![a_j \le a_m]\!]$.
4:     $c \leftarrow \mathsf{Reveal}([\![c]\!])$.
5:     if $c = 1$ then
6:         Let $i = i + 1$.
7:         Swap $[\![a_i]\!]$ and $[\![a_j]\!]$.
8: Let $p = i + 1$.
9: Swap $[\![a_p]\!]$ and $[\![a_m]\!]$.
10: Let $([\![e_1]\!], \dots, [\![e_m]\!]) = ([\![a_1]\!], \dots, [\![a_m]\!])$.
11: return $p, [\![e_1]\!], \dots, [\![e_m]\!]$.
```

Complexity. There are only two subprotocols that matter in terms of complexity. One is the shuffling protocol and the other is the comparison protocol. As described previously, we use the shuffling protocol proposed by Laura et al. [22], which exhibits a complexity of $O(1)$ rounds and $O(m \log m)$ communications when the number of parties n is constant. Since the quicksort algorithm requires $\Omega(m \log m)$ invocations of comparison, we have no need to take into account the complexity of the shuffling protocol.

The number of invocations of the comparison protocol is exactly the same as that of comparisons in the original quicksort algorithm. With a naive implementation, therefore, our quicksort protocol exhibits a complexity of $O(m \log m)$ rounds and $O(m \log m)$ comparisons.

We can improve the round complexity of the main part of the proposed quicksort protocol to $O(\log m)$ by setting the invocations of the comparison protocols to be parallel. First, we claim that the depth of the recursive calls is $\Theta(\log m)$ on average. Since our quicksort protocol shuffles the input in the first step, the input to the main part of the quicksort protocol is uniformly randomized. When the input is assumed to be uniformly randomized, the depth of the recursive calls for the quicksort algorithm is known to be $\Theta(\log m)$ on average [9]. Additionally, we can easily confirm that we can make the invocations of the subprotocol Partition parallel at each depth. Thus, through parallel implementation, our quicksort protocol exhibits a complexity of $O(\log m)$ rounds and $O(m \log m)$ comparisons on average.

3.3 Sorting Duplicated Values

When there are duplicate inputs in its secret values, our quicksort protocol may leak information regarding the input. For example, if the protocol invokes two comparison protocols $[\![a \le b]\!]$ and $[\![b \le a]\!]$ s.t. $a = b$, the results of the comparisons reveal the existence of a pair of shared values with identical secret values. Another example is the case when all the values are same. In this case, the results of comparisons are all true, and this implies many values are same with high probability.

We can easily address this problem, for example, by the following steps. Let m be the number of input shares and add $\lceil \log_2 m \rceil$ bits, which we call a tie breaker, to every input share in the least significant positions. Then, we execute the protocol treating the modified input as the input. The above modification gives the identical shared values strict order; therefore, solving the problem. Furthermore, depending on how we make the tie breaker, we can give the proposed quicksort protocol certain features. If we shuffle the tie breaker, the duplicated values are uniformly and randomly ordered. To generate a sorting protocol while retaining the original order of the duplicated items (such a sorting operation is called *stable*), we arrange the tie breakers in ascending order.

3.4 Further Extensions

Beyond sorting, our approach must be applied to many other data-dependent algorithms. We illustrate a *selection algorithm* which is for finding the k-th smallest number in a list. This includes finding the minimum, maximum, and median elements often executed in the database operation. For example, we can obtain the median MPC protocol that exhibits $O(\log m)$ rounds and $O(m)$ comparisons in the average case from Hoare's algorithm [18] and also obtain the protocol that exhibits the same rounds and comparisons even in the worst case from Blum's algorithm [5].

Our approach seems to be secure even in the malicious model if the shuffling, reveal, and comparison protocols are also secure in the malicious model. However, we are interested in constructing a practically efficient MPC protocol, and to our knowledge, there is no secret-sharing scheme providing practically efficient shuffling, reveal, and comparison protocols simultaneously. Therefore, we only give the proof in the semi-honest model in this paper.

4 Evaluation

In this section, we evaluate our quicksort protocol. We compare this protocol with other existing sorting protocols both asymptotically and experimentally. As a result, we show that our quicksort protocol exhibits a comparable computational complexity and significantly improved the running time in an experiment.

Table 1. Complexities of sorting protocols. m and R represent the number of the input values and the range of input values, respectively.

Sorting protocol	Rounds		Invocations of comparison	
	Average	Worst	Average	Worst
AKS sorting network [1]	$O(\log m)$	$O(\log m)$	$O(m \log m)$	$O(m \log m)$
Randomized shell sort [16]	$O(m)$	$O(m)$	$O(m \log m)$	$O(m \log m)$
Batcher's merge sort [2]	$O(\log^2 m)$	$O(\log^2 m)$	$O(m \log^2 m)$	$O(m \log^2 m)$
Oblivious arrayless bead sort [32]	$O(1)$	$O(1)$	$O(Rm)$	$O(Rm)$
Oblivious keyword sort [32]	$O(1)$	$O(1)$	$O(m^2)$	$O(m^2)$
Quicksort (proposed)	$O(\log m)$	$O(m)$	$O(m \log m)$	$O(m^2)$

Table 2. Performance of sorting protocols. m represents the number of the input values. The "N/A" means that the execution did not finish in 3, 600 seconds.

Sorting protocol	$m = 10$	$m = 10^2$	$m = 10^3$	$m = 10^4$	$m = 10^5$	$m = 10^6$
Randomized shell sort [16]	6.356[s]	86.355[s]	911.376[s]	N/A	N/A	N/A
Oblivious keyword sort [32]	0.335[s]	3.392[s]	387.128[s]	N/A	N/A	N/A
Batcher's merge sort [2]	1.331[s]	4.139[s]	14.285[s]	152.168[s]	2070.890[s]	N/A
Quicksort (proposed)	0.247[s]	0.488[s]	1.410[s]	9.859[s]	93.674[s]	1226.267[s]

4.1 Complexity Analysis

We first evaluated our quicksort protocol from an asymptotic perspective. As described in Sect. 3, this protocol exhibits a complexity of $O(\log m)$ rounds and $O(m \log m)$ comparisons on average, where m is the number of the input values. We summarize the complexities of ours and existing sorting protocols in Table 1 by taking into account parallelism.

As mentioned repeatedly, we are interested in practically efficient protocols; therefore, we stress the average case rather than the worst case. Our quicksort protocol requires $O(m \log m)$ comparisons on average, which is asymptotically optimal for comparison sorts. Our quicksort protocol is superior to randomized shell sort [16] and Batcher's merge sort [2] in either rounds or comparisons. The AKS sorting network [1] has the same complexity on average. The oblivious arrayless bead sort [32] exhibits $O(1)$ rounds and $O(Rm)$ comparisons where R is the range of secret values. This algorithm is quite efficient when R is small, e.g., the secret value belongs to $\{0, 1\}$. However, when R is large, e.g., $R = 2^{32}$, it becomes quite inefficient. The oblivious keyword sort [32] has a comparable complexity to ours. This exhibits a complexity of constant rounds that is superior to ours but $O(m^2)$ comparisons that is inferior on average.

4.2 Experimental Results

As the quicksort often outperforms other sorting algorithms with $O(m \log m)$ comparisons in practice [30], the experiment results are very important for practical use. We implemented our quicksort protocol and existing sorting protocols, such as the randomized shell sort [16], the oblivious keyword sort [32] and Batcher's merge sort [2], for comparison. The AKS sorting network [1] was not implemented since this algorithm is not of practical interest. We also did not implement the oblivious arrayless bead sort [32]. This is because in many applications such as Oblivious RAM the range of numbers, R, is large; therefore, this sort protocol becomes quite inefficient.

We implemented sorting protocols on (2, 3)-Shamir's secret-sharing scheme with corruption tolerance $t = 1$. This is because MPC protocols generally do not scale well as the number of participants increases, and such MPC protocols executed by a few participants can be building blocks of ones executed by many participants [11]. For better performance, we implemented component protocols secure against a semi-honest adversary. This implies all the implemented sorting protocols are also secure against such an adversary. We implemented the comparison protocol proposed by Damgård et al. [10] as a building block of all sorting protocols. The quicksort protocol additionally uses the shuffling protocol proposed by Laura et al. [22]. Our implementation of the randomized shell sort and Batcher's merge sort protocols are based on circuit representations. That is, we replaced the comparators in the original algorithms to comparator protocols constructed by comparisons, multiplications, and additions. We implemented all of them on C++ and compiled by g++ 4.6.1. All values are in $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, where p is a prime number 4294967291 and satisfies $2^{31} < p < 2^{32}$, that is, 32-bit words.

We then timed how long the running time of these protocols is. All the experiments were conducted on three laptop machines with an Intel Core i5 2540M 2.6-GHz CPU and 8 GB of physical memory. These three machines were connected to a 1-Gbps LAN. The running times of the sorting protocols are shown in Fig. 1, and detailed times in some cases are summarized in Table 2 where m is the number of input shared values.

As expected, our quicksort protocol allowed us to consider large inputs size. The results show that our quicksort protocol is much faster than randomized shell sort and oblivious keyword sort, and about ten to twenty times faster than Batcher's merge sort. Consequently, the proposed quicksort protocol significantly improved the running time of the existing sorting protocols. In other words, our quicksort protocol is practically efficient.

5 Conclusion

We proposed a simple and general approach, shuffling before sorting, for converting data-dependent but efficient comparison sort algorithms to MPC sorting protocols. We then constructed a quicksort protocol from the quicksort algorithm with our approach. The resultant protocol is practically efficient since it has comparable computational complexity and significantly improved the running time compared to existing protocols in experiments.

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-David, A., Nisan, N., Pinkas, B.: Fairplaymp: a system for secure multi-party computation. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM Conference on Computer and Communications Security, pp. 257–266. ACM (2008)
4. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: [29], pp. 1–10
5. Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. J. Comput. Syst. Sci. 7(4), 448–461 (1973)
6. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008)
7. 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)
8. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: [29], pp. 11–19
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
10. 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: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 285–304. Springer, Heidelberg (2006)
11. Damgård, I., Ishai, Y.: Constant-round multiparty computation using a black-box pseudorandom generator. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 378–394. Springer, Heidelberg (2005)
12. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 144–163. Springer, Heidelberg (2011)
13. Geisler, M.: Cryptographic Protocols: Theory and Implementation. PhD thesis, University of Aarhus (2010)
14. Goldreich, O.: The Foundations of Cryptography. Basic Applications, vol. 2. Cambridge University Press (2004)
15. 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)
16. Goodrich, M.T.: Randomized shellsort: A simple oblivious sorting algorithm. In: SODA, pp. 1262–1277 (2010)
17. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: Tasty: tool for automating secure two-party computations. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM Conference on Computer and Communications Security, pp. 451–462. ACM (2010)
18. Hoare, C.A.R.: Algorithm 65: find. Commun. ACM 4(7), 321–322 (1961)
19. Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: NDSS (2012)
20. Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. IACR Cryptology ePrint Archive 2011, 122 (2011)
21. Knuth, D.E.: Art of Computer Programming, 2nd edn. Sorting and Searching, vol. 3, ch. 5. Addison-Wesley Professional (1998)
22. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 262–277. Springer, Heidelberg (2011)
23. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - secure two-party computation system. In: USENIX Security Symposium, pp. 287–302 (2004)

24. Ning, C., Xu, Q.: Multiparty computation for modulo reduction without bit-decomposition and a generalization to bit-decomposition. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 483–500. Springer, Heidelberg (2010)
25. Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 343–360. Springer, Heidelberg (2007)
26. Obana, S., Araki, T.: Almost optimum secret sharing schemes secure against cheating for arbitrary secret distribution. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 364–379. Springer, Heidelberg (2006)
27. Ogata, W., Kurosawa, K.: Optimum secret sharing scheme secure against cheating. In: Maurer, U.M. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 200–211. Springer, Heidelberg (1996)
28. Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)
29. Simon, J. (ed.): Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC, Chicago, Illinois, USA, May 2-4. ACM (1988)
30. Skiena, S.S.: The Algorithm Design Manual, 2nd edn. Springer Publishing Company, Incorporated (2008)
31. 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)
32. Zhang, B.: Generic constant-round oblivious sorting algorithm for MPC. In: Boyen, X., Chen, X. (eds.) ProvSec 2011. LNCS, vol. 6980, pp. 240–256. Springer, Heidelberg (2011)

A Formal Definition of the Security

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 P_i during the protocol execution of ρ on inputs $\mathbf{x}$ as $\textsc{view}^{\rho}_{P_i}(\mathbf{x}) = (x_i, r_i; \mu_1, \ldots, \mu_\ell)$ where r_i is P_i's random tape, and μ_j is the j-th message that P_i received in the protocol execution. We also denote the output of P_i as $\textsc{output}^{\rho}_{P_i}(\mathbf{x})$.

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

Definition 1 ([14]). *Let $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be a probabilistic n-ary functionality, ρ be a protocol,* $\textsc{view}^{\rho}_{\mathbb{I}}(\mathbf{x}) = (\textsc{view}^{\rho}_{P_{i_1}}(\mathbf{x}), \ldots, \textsc{view}^{\rho}_{P_{i_t}}(\mathbf{x}))$, *and*

$$\textsc{output}^{\rho}(\mathbf{x}) = (\textsc{output}^{\rho}_{P_1}(\mathbf{x}), \ldots, \textsc{output}^{\rho}_{P_n}(\mathbf{x})).$$

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*

$$\{(\mathcal{S}(\mathbb{I}, \mathbf{x}_{\mathbb{I}}, f_{\mathbb{I}}(\mathbf{x})), f(\mathbf{x}))\} \equiv \left\{(\textsc{view}^{\rho}_{\mathbb{I}}(\mathbf{x}), \textsc{output}^{\rho}(\mathbf{x}))\right\}.$$

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 2 ([14]). *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 by 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.

B Proof of Theorem 1

Let $[\![b'_1]\!], \dots, [\![b'_m]\!]$ be the shuffled (and renewed) shared values in the Step 1 of Quicksort(). The view of adversaries consists of their inputs $[\![a_1]\!]_{\mathbb{I}}, \dots, [\![a_m]\!]_{\mathbb{I}}$, random tapes, $[\![b'_1]\!]_{\mathbb{I}}, \dots, [\![b'_m]\!]_{\mathbb{I}}$, $[\![c]\!]_{\mathbb{I}}$, and c. The output consists of $[\![b_1]\!]_{\mathbb{I}}, \dots, [\![b_m]\!]_{\mathbb{I}}$. Note that the adversaries have no view of the subprotocols Shuffle$(\cdot)$, $[\![\cdot \le \cdot]\!]$, and Reveal$(\cdot)$ since the execution of these protocols are substituted with the oracle invocation of functionalities $f^{\mathsf{Shuffle}}_{\Pi_{\mathsf{SS}}}$, $f^{\mathsf{CMP}}_{\Pi_{\mathsf{SS}}}$, and $f^{\mathsf{Reveal}}_{\Pi_{\mathsf{SS}}}$, 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 $[\![b'_1]\!]_{\mathbb{I}}, \dots, [\![b'_m]\!]_{\mathbb{I}}$ and c, let $\pi' : [m] \to [m]$ be a permutation that satisfies $[\![b_i]\!] = [\![b'_{\pi'(i)}]\!]$ $(i \in [m])$. There exists exactly one such permutation since $\{b_1, \dots, b_m\}$ is distinct and $([\![b_1]\!], \dots, [\![b_m]\!])$ is a permutated sequence from $([\![b'_1]\!], \dots, [\![b'_m]\!])$ by the swap operations executed in Step 7 or Step 9 of Partition$(\cdot)$. Once π' is perfectly simulated, $[\![b'_i]\!]_{\mathbb{I}}$ is also perfectly simulated by setting $[\![b_{\pi'^{-1}(i)}]\!]_{\mathbb{I}}$ as the simulated shares and c is also perfectly simulated by setting the value

$$c' = \begin{cases} 1 \text{ if } \pi'^{-1}(i) \le \pi'^{-1}(j) \\ 0 \text{ otherwise} \end{cases}$$

when $[\![b'_i \le b'_j]\!]$ is executed. Now we claim that $\mathcal{S}$ perfectly simulates π' by selecting just a uniformly random permutation. By the correctness of the shuffling and quicksort protocols, $b'_i = a_{\pi_r(i)}$ and $b_i = a_{\pi_s(i)}$ $(i \in [m])$ hold for a fixed (according to $a_1, \dots, a_m$) permutation $\pi_s : [m] \to [m]$ and a uniformly random permutation $\pi_r : [m] \to [m]$. $\pi_s = \pi_r \circ \pi'$ holds and this implies $\pi' = \pi_r{}^{-1} \circ \pi_s$. Therefore, π' is uniformly random.

As for $[\![c]\!]_{\mathbb{I}}$, $\mathcal{S}$ picks $|\mathbb{I}|$ uniformly random numbers and sets them as the simulated values for $[\![c]\!]_{\mathbb{I}}$. Since $[\![c]\!]_{\mathbb{I}}$ is the output shares of Reveal$(\cdot)$, the above simulation is perfect.

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