CYBERNETICA
Institute of Information Security

Oblivious Sorting of Secret-Shared Data

Dan Bogdanov, Sven Laur, Riivo Talviste

T-4-19 / 2013

Dan Bogdanov[1], Sven Laur[2], Riivo Talviste[1,2].
[1] Cybernetica, Institute of Information Security,
[2] University of Tartu, Institute of Computer Science,

The research reported here was supported by:

1. the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS
2. European Social Fund Doctoral Studies and Internationalisation Programme DoRa
3. EU project UaESMC (contract no. FP7-284731).

Cybernetica research reports are available online at
`http://research.cyber.ee/`

Mailing address:
Cybernetica AS
Mäealuse 2
12618 Tallinn
Estonia

Oblivious Sorting of Secret-Shared Data

Dan Bogdanov, Sven Laur, Riivo Talviste

August 20, 2013

Abstract

In this research report we give an overview of methods for obliviously sorting data that has been protected using secret sharing. Oblivious sorting is an important primitive in privacy-preserving data analysis, as it can impose order on secret-shared data, simplifying the construction of data transformation and aggregation algorithms. In this work, we compare several published sorting methods with a novel implementation based on sorting networks. We evaluate the theoretical performance and discuss the practical implications of the different approaches.

1 Introduction

Secret sharing is a cryptographic primitive for securely sharing confidential information among several parties. The advantage of secret sharing over techniques like encryption is the existence of efficient *secure multiparty computation* (SMC) protocols that allow secret-shared data to be processed without reconstructing the original secrets. While there exist encryption schemes that allow such processing (e.g., homomorphic encryption schemes), they are less efficient than constructions based on homomorphic secret sharing.

There are several general approaches towards implementing sorting on secret-shared data. Each one is based on a different set of required primitives and has a different performance profile. Some algorithms have a low running time, but a high memory usage. One of our main goals of this report is to understand, what is the best algorithm for a given scenario.

Our benchmark results are measured on implementation developed for the Sharemind secure computation platform [Bog13]. The algorithms are implemented using the SecreC programming language [BLR13]. We implemented all discussed algorithms on the same platform to perform a fair comparison of the implementations.

1.1 Approach overview and related work

The first group of sorting algorithms we discuss is based on the blueprint provided in [LWZ11]. They propose oblivious shuffling as a general primitive for constructing oblivious data processing algorithms. In a follow-up work, Zhang proposes multiple constant-round sorting schemes for linear secret sharing schemes, including counting sort, arrayless bead sort and sorting key indexed data [Zha11]. These protocols assume a known range R for inputs, but [Zha11] also proposes to use radix sort on top of these protocols to deal with larger ranges.

Recently, Hamada et al. [HKI$^+$13] proposed a generic construction that can be used to convert any comparison-based sorting algorithm into data-oblivious algorithm without the overhead of

evaluating all the possible data flow branches. They have also developed an implementation and provide benchmarks.

The second group of sorting algorithms is based on sorting networks. One early implementation of sorting based on sorting networks is presented in [JKU11]. Their implementation was originally designed to find top k elements in a network anomaly detection scenario, similarly to the work of Burkhart et al [BD10].

The third group uses individual bit-level access on bitwise shared data. Counting sort [CLRS01, Edm08] is a sorting algorithm that can sort an array of integers in small range by first constructing a frequency table and then rearranging items in the array according to this table. To sort arrays of larger integers, counting sort can be used as a subroutine for radix sort [Hol89] that uses it to sort an array of integers one digit at a time.

In this report, we collect the presented algorithms and provide a systematized analysis of their properties. We also describe extensions for sorting matrix data by one or more columns and describe optimizations.

2 Constructions based on oblivious shuffling

Comparison-based sorting is one of the most famous classes of sorting algorithms. These algorithms use the comparison operation to determine the correct sequence between elements of a given array. For example, quicksort, merge sort and bubble sort belong to this class.

Comparison-based sorting algorithms are inherently data-dependent as the flow of the algorithm depends on the outcome of the comparison operation. Consequently, each comparison outcome should be a public value that is used to determine the next step. This is a problem for secret-shared data as the change in data flow might leak too much information about the shared values themselves. A simple solution would be to evaluate all the branches of the sorting algorithm and obliviously construct a mask vector that selects the correct output at the end, but this yields too much extra work and the algorithm would not be as efficient anymore.

However, if we *obliviously shuffle* the data before performing a comparison-based sort, there is no privacy leak. As we are comparing values in a randomly shuffled vector, any published comparison results are also random. This approach can also be used to obliviously implement other tasks like top-k queries and Hoare's selection algorithm [Hoa61].

3 Constructions based on sorting networks

A sorting network is an abstract structure that consists of several layers of comparators that change the positions of incoming values. These comparators are also called CompEx (compare-and-exchange) functions. A CompEx function takes two inputs, compares them according to the required condition and exchanges them, if the comparison result is true. The following mathematical representation shows a CompEx function for sorting numeric values in the ascending order.

$$\mathsf{CompEx}(x, y) = \begin{cases} (y, x), & \text{if } x > y \\ (x, y), & \text{otherwise.} \end{cases}$$

Basically, a sorting network is an arrangement of CompEx functions so that if the comparators of all the layers of the sorting network are applied on the input data array, the output data array will

Algorithm 1: Algorithm for sorting an array of integers.

Data: Input array $\mathcal{D} \in \mathbb{Z}_n^k$ and a sorting network $\mathcal{N} \in \mathbb{L}^m$.
Result: Sorted output array $\mathcal{D}' \in \mathbb{Z}_n^k$.

```
foreach 𝕃_i ∈ 𝒩 do
    foreach (x, y) ∈ 𝕃_i do
        (𝒟_x, 𝒟_y) ← CompEx(𝒟_x, 𝒟_y)
    end
end
```

be sorted according to the desired condition. For a more detailed explanation of sorting networks with examples, see [Knu98].

A sorting network is suitable for oblivious sorting, because it is static and independent of the incoming data. One network will sort all possible input arrays, making the approach inherently oblivious. Furthermore, sorting networks are relatively straightforward to implement using secure multiparty computation, as we only need a Min (minimum of two values) and Max (maximum of two values) operation to sort numeric data. Using these two operators, we can easily implement the CompEx function as a straight line program (one with no conditional branches). For an example of a CompEx function that sorts an array of numbers in an ascending order, see the following formula.

$$\mathsf{CompEx}(x, y) = (\mathsf{Min}(x, y), \mathsf{Max}(x, y))$$

We express a k-element array of n-bit integers as $\mathbb{Z}_n^k$. We will assume that the input and the output of the sorting network are in this form. We also need to represent the structure of that network. Intuitively, a sorting network consists of several layers of CompEx functions. The inputs of each CompEx function can be encoded with their indices in the array. Therefore, we will represent a layer $\mathbb{L}_i$ consisting of ℓ_i CompEx functions as follows.

$$\mathbb{L}_i = (\mathbb{N} \times \mathbb{N})^{\ell_i}$$

The complete sorting network, consisting of m layers, will then be written as $\mathbb{L}^m$. We also add one restriction to the network for efficiency and simplicity. We assume that no index appears more than once in each individual layer of the sorting network.

Algorithm 1 presents a general algorithm for evaluating a sorting network in this representation. Note that we can use the same array $\mathcal{D}$ for storing the results of the compare-exchange operation, because according to our assumption above, as a single layer does not use the same array index twice.

This algorithm is easy to implement securely, because we only have to provide a secure implementation of the CompEx operation. The rest of the algorithm is independent of the data, and can be implemented with public operations.

We intentionally omit guidance on how to implement a generator for sorting networks, as this is a well-researched area. We refer the reader to the classical work of Knuth as a starting point [Knu98].

4 Sorting bitwise shared data

In some secret sharing schemes accessing individual bit shares is less expensive than other operations. This holds for bitwise shared (XOR-shared) data. Inexpensive bit level access allows to use

other types of sorting algorithms.

Counting sort [CLRS01, Edm08] is a sorting algorithm that can sort an array of integers in small range by first constructing a frequency table and then rearranging items in the array according to this table (for an example of using counting sort on binary data see Algorithm 2). To sort arrays of larger integers, counting sort can be used as a subroutine for radix sort. Radix sort [Hol89] sorts an array of elements by rearranging them based on counting sort results on digits on the same positions. However, to use this radix sort on secret shared data, it must be data-oblivious. To accomplish that, it is sufficient to implement the underlying counting sort obliviously. As we are dealing with bitwise shared data, we use oblivious counting sort only on binary data. Algorithm 3 shows the full protocol of radix sort that uses oblivious binary counting sort as a subroutine.

Algorithm 2: Counting sort algorithm on binary array.

Data: Binary input array a.
Result: Array b with elements of a in increasing order.
`/* Count number of zeros. */`
$n \leftarrow a.length$; $n_0 \leftarrow n - sum(a)$
`/* Keep counters for currently processed zeros and ones. */`
$c_0 \leftarrow 0$; $c_1 \leftarrow 0$
`/* Put each element in right position. */`
foreach $i \in 1 \dots n$ **do**
  if $a[i] == 0$ **then**
    $c_0 = c_0 + 1$
    $b[c_0] = a[i]$
  else
    $c_1 = c_1 + 1$
    $b[n_0 + c_1] = a[i]$
  end
end
return b

As counting sort (and radix sort) do not use comparisons, they are not bound by the computational complexity lower bound of $\Omega(n \log n)$ for comparison-based sorting algorithms. Counting sort has a complexity of $\mathcal{O}(n)$ and radix sort on k-digit elements that uses counting sort as a subroutine, has a computational complexity of $\mathcal{O}(kn)$. Unfortunately, oblivious counting sort protocol also uses addition and multiplication operations, which are expensive protocols on bitwise shared data. Therefore, it is useful to convert input bits to additively shared data and work in this domain. Algorithm output can still be bitwise shared. Consequently, using subprotocol invocations, oblivious radix sort can be written as the following combination:

$$k \cdot (\mathsf{ShareConv}(n) + \mathsf{Mult}(n) + \mathsf{Shuffle}(n, 2) + \mathsf{Publish}(n)),$$

where k is the number of digits.

Algorithm 3: Radix sort using oblivious binary counting sort as subroutine.

Data: Input array $[\![a]\!]$. Each element has k digits.
Result: Sorted array $[\![a]\!]$.

```
/* Iterate over all digits starting with the least significant digit.      */
```
1 $n \leftarrow [\![a]\!].length$
2 **foreach** $m \in 0 \dots k-1$ **do**
```
    /* Construct a binary vector consisting of m-th digits.                */
```
3 $d \leftarrow ([\![a[1]]\!]_m, [\![a[2]]\!]_m, \dots, [\![a[n]]\!]_m)$
```
    /* Count number of zeros.                                              */
```
4 $[\![n_0]\!] \leftarrow n - sum([\![d]\!])$
```
    /* Keep counters for currently processed zeros and ones.               */
```
5 $[\![c_0]\!] \leftarrow 0$; $[\![c_1]\!] \leftarrow 0$
```
    /* Keep shared order vector.                                           */
```
6 $[\![ord]\!]$
```
    /* Put each element in right position.                                 */
```
7 **foreach** $i \in 1 \dots n$ **do**
8 $[\![c_0]\!] = [\![c_0]\!] + 1 - [\![d[i]]\!]$
9 $[\![c_1]\!] = [\![c_1]\!] + [\![d[i]]\!]$
```
    /* Obliviously update order vector.                                    */
```
10 $[\![ord[i]]\!] = (1 - [\![d[i]]\!]) * [\![c_0]\!] + [\![d[i]]\!] * ([\![n_0]\!] + [\![c_1]\!])$
11 **end**
12 Obliviously shuffle two row database $([\![a]\!], [\![ord]\!])$.
13 $ord \leftarrow declassify([\![ord]\!])$
14 Rearrange elements in $[\![a]\!]$ according to ord.
15 **end**
16 **return** $[\![a]\!]$

5 Optimization techniques

5.1 The use of vector operations

We now show how to optimize the implementation of the proposed algorithms using parallel operations on multiple values. Such SIMD (single instruction, multiple data) operations are very efficient on most secure multiparty computation paradigms. For example, secure multiparty computation protocols based on secret sharing can put the messages of many parallel operations into a single network message, saving on networking overhead.

In some cases, parallelizable naive protocols may work better in practice than sequential protocols with lower computational complexity. We propose a comparison-based sorting protocol where the most expensive operations—comparisons—are all done in one go. As the comparisons done this way cannot depend on the previous comparison results (as there are none), we have to compare every element with every other element in the array we want to sort. Computationally, this means that this naive algorithm works always in worst case time for comparion sort: $O(n^2)$. As with privacy-preserving comparison sorts proposed by Hamada et al. [HKI+13], our algorithm NaiveCompSort (see Algorithm 4) begins with obliviously shuffling the array, followed by comparisons and rearranging elements according to published comparison results.

Algorithm 4: Database sorting based on shuffle

Data: Shared table $[\![T]\!] = \{[\![t_1]\!], [\![t_2]\!], \ldots, [\![t_n]\!]\}$ of values to sort
Result: Shared table $[\![T']\!]$ having values from $[\![T]\!]$ sorted

1 Let $[\![T]\!] = \text{Shuffle}([\![T]\!])$
2 **for** $i < j \in \{1, 2, \ldots, n\}$ **do**
3 Let $[\![g_{i,j}]\!] = [\![t_i]\!] \overset{?}{\leq} [\![t_j]\!]$
4 Publish the values $[\![g_{i,j}]\!]$ and sort the values of $[\![T]\!]$ according to them, obtaining $[\![T']\!]$
5 **return** $[\![T']\!]$

Using SMC subprotocol invocations as building blocks, the proposed sorting algorithm can be written as

$$\mathsf{NaiveCompSort}(n) = \mathsf{Shuffle}(n) + \mathsf{Comp}(n(n-1)/2) + \mathsf{Publish}(n(n-1)/2),$$

whereas the version proposed by Hamada et al. [HKI+13] that uses regular quicksort algorithm after oblivious shuffling uses these subprotocol invocations (on average):

$$\mathsf{ObliviousQuicksort}(n) = \mathsf{Shuffle}(n) + O(n \log n)\mathsf{Comp}(1) + O(n \log n)\mathsf{Publish}(1).$$

As NaiveCompSort uses less individual subprotocol invocations (although with more data), it accumulates less network latency and works faster for smaller input vectors.

If we observe Algorithm 1, we see that it is trivial to make it use vector operations. Note that the outer loop of that algorithm can not be vectorized by replacing it with parallel operations. The intuitive reason is that every layer of the sorting network is directly dependent on the output of the previous layer and this does not allow multiple layers to be processed in parallel.

However, thanks to the assumption on the uniqueness of indices we made while describing the structure of the sorting network, we can trivially vectorize the inner loop. Indeed, we can replace the entire inner loop with two operations. One computes the maximum values of all input pairs and the other computes the minimal values.

5.2 Changing the share representation

Both comparison-based sorting algorithms and sorting networks rely on comparison operation. As comparison operator is a bitlevel operation [BNTW12], it is faster when implemented on bitwise shared data. We can make use of this fact by converting the additively shared input data into bitwise shared data and running the intended protocol that uses comparisons on this converted data. After the protocol has finished, the results should be converted back to additively shared data. However, the conversion from additively shared values to bitwise shared values is not free in the sense of communication complexity, so we can use more efficient comparisons only with the cost of adding conversion. Taking this into account, for protocols that make heavy use of parallel comparison operations (e.g. sorting networks and NaiveCompSort), the amount of communication time saved by using comparisons on bitwise shared data outweighs the communication time cost that we have to pay for the conversion.

6 Sorting more complex data structures

We often need to sort data in other form as arrays. For example, we may want to sort a table (matrix) of values according to a certain key column.

Oblivious radix sort and oblivious comparison based sorting algorithms — both the naive one in Algorithm 4 and the constructions proposed by Hamada et al. [HKI+13] — publish some intermediate values during execution. Thus, these protocols have to obliviously shuffle the data before publishing it to hide the trace from published values back to the original elements.

Such sorting protocols can be easily modified to support more complex data structures. Let us have a two-dimensional secret shared table with n rows and a column (indexed k) according to which we want to sort the rows in this table. First, we obliviously shuffle the order of rows in the whole table[1]. Next, we extract the k-th column from the table and pass it to the sorting protocol of our choice as an n-element vector. Additionally, we construct an n-element index vector $(1, 2, \ldots, n)$ and give it as an additional argument to the sorting protocol. Now, the sorting protocol behaves as usual, but in addition to swapping elements in the data vector, it also performs the same swaps on the index vector. Finally, after the sorting protocol has finished, we publish the output index vector and use it as a permutation to rearrange rows in the table.

As with publishing the individual comparison results in the sorting protocol, publishing the index vector also leaks information on how the elements were rearranged in the data vector. However, as the original table was obliviously shuffled, this leaks no information on the original placement of rows in the initial table.

Oblivious sorting with sorting networks, as described in Section 3, does not publish any intermediate values and thus does not need to obliviously shuffle the original data vector to hide relations between it and any published values. Consequently, when using sorting networks, we cannot use the idea of constructing a permutation vector and publicly rearranging the rest of the table according to it. Instead, we need to redefine the CompEx operation to work on the full data structure.

Let's consider the same case where we need to sort a table of integer values according to a certain column. Algorithm 1 still works perfectly, but we need to design a new kind of a compare-exchange function that evaluates the comparison condition on the value from the respective column, but performs the exchange on the whole table row.

Assume that our input data table is in the form of a matrix $\mathcal{D}_{i,j}$ where $i = 1 \ldots k$ and $j = 1 \ldots l$. Then, the CompEx needs to compare and exchange two input arrays $\mathcal{A}, \mathcal{B} \in \mathbb{Z}_n^k$ according to the comparison result from column c. Equation (1) shows the definition of such a function.

$$\mathsf{CompEx}(\mathcal{A}, \mathcal{B}, c) = \begin{cases} (\mathcal{B}, \mathcal{A}), & \text{if } \mathcal{A}_c > \mathcal{B}_c \\ (\mathcal{A}, \mathcal{B}), & \text{otherwise.} \end{cases} \tag{1}$$

A suitable oblivious implementation for a CompEx shown in Equation (1) on this structure is given in Algorithm 5. The algorithm uses two steps. First, it performs an oblivious comparison part of CompEx. In the given example, it evaluates a greater-than comparison. The main important thing here is that the result should be expressible as either 0 or 1 so that it can later be used in the oblivious exchange. The second step is to obliviously exchange the input data based on the result of the comparison.

This algorithm has the following assumptions for oblivious implementation.

[1] Note that this shuffling is already a part of comparison based sorting protocols like that of Hamada et al. and Algorithm 4. However, this extra step has to be added for radix sort.

Algorithm 5: Algorithm for obliviously comparing and exchanging two rows in a matrix.

Data: Two input arrays $\mathcal{A}, \mathcal{B} \in \mathbb{Z}_n^k$, column index $c \in \{1 \dots k\}$.
Result: Pair of arrays $(\mathcal{A}', \mathcal{B}') = \mathsf{CompEx}(\mathcal{A}, \mathcal{B}, c)$.
`/* Compute result of the condition */`
1 $b \leftarrow \begin{cases} 1, & \text{if } \mathcal{A}_c > \mathcal{B}_c \\ 0, & \text{otherwise.} \end{cases}$
`/* Exchange the vectors based on the condition */`
2 **foreach** $i \in 1 \dots k$ **do**
3 $\quad \mathcal{A}'_i = (1-b)\mathcal{A}_i + b\mathcal{B}_i$
4 $\quad \mathcal{B}'_i = b\mathcal{A}_i + (1-b)\mathcal{B}_i$
5 **end**

1. We can obliviously implement the comparison operation on input data so that the result is represented as a numeric zero or one.
2. We can subtract the comparison result from the constant 1.
3. We can cast the numeric zero-one result (or the subtraction result) to a type that can be multiplied with the input data type.
4. We can add two values of the input data type.

Fortunately, these assumptions hold for different secure computation paradigms. It is relatively easy to implement such an oblivious CompEx function with secure multiparty computation on different integer sizes. Moreover, Algorithm 5 can be parallelized the same way as Algorithm 1.

7 A comparison of oblivious sorting methods

Protocol	Rounds	Communication complexity
Arrayless bead sort [Zha11]	$\mathcal{O}(1)$	$\mathcal{O}(Rn)\mathsf{Comp}$
Quicksort [HKI+13] (average)	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$
Sorting network	$\sum_{i=1}^{m} \mathsf{Comp}(\ell_i) + \mathsf{Mult}(4\ell_i)$	
Naive comparison sort	$\mathsf{Shuffle}(n) + \mathsf{Comp}(n(n-1)/2) + \mathsf{Publish}(n(n-1)/2)$	
Radix sort	$k \cdot (\mathsf{ShareConv}(n) + \mathsf{Mult}(n) + \mathsf{Shuffle}(n, 2) + \mathsf{Publish}(n))$	

Table 1: Comparison of oblivious sorting algorithms. n is the number of elements to sort, k is number of digits and R is the range of numbers, where applicable. For sorting networks, m is the number of layers in the network and ℓ_i is the number of CompEx operations on i-th layer.

Here we will list the round and communication complexities of all the mentioned sorting protocols in one table for easier comparison. Where possible, for communication complexity, Table 1 does not use the usual $\mathcal{O}$ notation but combination of all subprotocol invocations. This allows us to show which parts of the protocol can be parallelized to save on network latency. For example,

mProtocol(n) means that SMC protocol Protocol is invoked m times with n parallel elements for each invocation.

References

[BD10] M. Burkhart and X. Dimitropoulos. Fast Privacy-Preserving Top-k Queries Using Secret Sharing. In *Proceedings of ICCCN 2010*, pages 1–7, 2010.

[BLR13] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-Polymorphic Programming of Privacy-Preserving Applications. Cryptology ePrint Archive, Report 2013/371, 2013. `http://eprint.iacr.org/`.

[BNTW12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[Bog13] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 8.2 Counting Sort, pages 168–170. MIT Press and McGraw-Hill, 2nd edition, 2001.

[Edm08] Jeff Edmonds. *How to Think about Algorithms*, chapter 5.2 Counting Sort (a Stable Sort), page 7275. Cambridge University Press, 2008.

[HKI+13] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *Information Security and Cryptology (ICISC 2012)*, volume 7839 of *LNCS*, pages 202–216. Springer, 2013.

[Hoa61] C. A. R. Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

[Hol89] Herman Hollerith. US395781 (A) - ART OF COMPILING STATISTICS. European Patent Office, 1889. `http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=395781`.

[JKU11] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure Multi-Party Sorting and Applications. Cryptology ePrint Archive, Report 2011/122, 2011. `http://eprint.iacr.org/`.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.

[Zha11] Bingsheng Zhang. Generic Constant-Round Oblivious Sorting Algorithm for MPC. In *Provable Security*, volume 6980 of *LNCS*, pages 240–256. Springer, 2011.