Bureaucratic Protocols for Secure Two-Party Sorting, Selection, and Permuting

Guan Wang
Syracuse University
Syracuse, NY, USA 13244
gwang07@syr.edu

Tongbo Luo
Syracuse University
Syracuse, NY, USA 13244
toluo@syr.edu

Michael T. Goodrich
University of California, Irvine
Irvine, CA, USA 92697
goodrich@ics.uci.edu

Wenliang Du
Syracuse University
Syracuse, NY, USA 13244
wedu@syr.edu

Zutao Zhu
Syracuse University
Syracuse, NY, USA 13244
zutao@syr.edu

ABSTRACT

In this paper, we introduce a framework for secure two-party (S2P) computations, which we call *bureaucratic* computing, and we demonstrate its efficiency by designing practical S2P computations for sorting, selection, and random permutation. In a nutshell, the main idea behind bureaucratic computing is to design data-oblivious algorithms that push all knowledge and influence of input values down to small black-box circuits, which are simulated using Yao's garbled paradigm. The practical benefit of this approach is that it maintains the zero-knowledge features of secure two-party computations while avoiding the significant computational overheads that come from trying to apply Yao's garbled paradigm to anything other than simple two-input functions. [1]

Categories and Subject Descriptors

K.6 [**Management of Computing and Information Systems**]: Miscellaneous

General Terms

Algorithms, Security

Keywords

Secure two-party computation, sorting, oblivious algorithms, bureaucratic protocols

1. INTRODUCTION

As people become more and more concerned about their privacy, privacy-preserving techniques have been receiving increasing attention in the research community, with secure two-party (S2P) computations being one of the primary solutions that have been studied to address this concern. S2P computations allow two parties, say Alice and Bob, to evaluate a public known function, $f(X, Y)$, on private data, with X belonging to Alice and Y belonging to Bob, without disclosing any information on their inputs other than what can be inferred from the result, $f(X, Y)$ [14, 30].

One of the most theoretically elegant and general techniques for achieving S2P computations is Yao's garbled circuit paradigm [30]. In using this paradigm, Alice and Bob compile $f(X, Y)$ into a Boolean circuit, which they then cooperatively evaluate using cryptographically masked versions of their respective inputs, X and Y. The generality of this technique is derived from the fact it can be applied to any function that can be converted into a Boolean circuit, which implies that it works for any function that can be computed by an algorithm[2].

Bringing Yao's garbled circuit paradigm out of the realm of theoretical computer science, the Fairplay system [25] is a beautiful software system that implements Yao's garbled circuit paradigm to compile Boolean circuits from program specifications and then simulate the resulting circuits using cryptographic masking of the inputs shared by the two parties. Unfortunately, as the popularity of the Fairplay system is growing, people are discovering that Yao's garbled circuit approach incurs major computational costs for anything other than basic functions, like MIN, MAX, ADD, and compare-exchange. Thus, the practical usefulness of Yao's garbled circuit paradigm, and, hence, the Fairplay system, is diminished for more complex computations. For example, using the Fairplay system's implementation of Yao's garbled circuit paradigm, we can turn Quicksort into a circuit, and achieve sorting of an array between Alice and Bob, where each array element s_i is shared by Alice and Bob (e.g., $s_i = a_i \oplus b_i$, where a_i is known to Alice and b_i is known to Bob). We know that Quicksort has an $O(n \log n)$ average-case complexity, and is considered as one of the most efficient sorting algorithms. Nevertheless, we have observed that the running time of Quicksort is $\Theta(n^2)$ if it is implemented using

[1]This work has partially supported by Awards No. 0430252 and 0618680 from the United States National Science Foundation.

ASIACCS'10 April 13–16, 2010, Beijing, China.

[2]Recall that the formal definition of an algorithm states that it must terminate on every possible input.

a garbled circuit for S2P with Fairplay. This performance, of course, completely negates the efficiency of using Quicksort. Even the simple Bubblesort can achieve $\Theta(n^2)$ in the S2P environment. Moreover, such examples are more than just academic exercises, since sorting plays a role in several S2P protocols.

1.1 The Benefit of Bureaucracy

To help understand better what causes an efficient algorithm in the non-S2P environment to become inefficient in the S2P environment, let us consider the following two simple algorithms. Both algorithms calculate the number of ones in the first n positions in a binary sequence, S, which we assume are indexed from 1 to n:

```
CountOne_1(S,n):
    if (n == 1) then
        return S[1]
    else
        i = n
        if (S[i] == 1) then
            return 1 + CountOne_1(S,n-1)
        else
            return CountOne_1(S,n-1)
        end_if
    end_if

CountOne_2(S,n):
    countOne = 0
    for i = 1 to n do
        countOne = S[i] + countOne
    end_for
    return countOne
```

Note that both algorithms have running times that are $O(n)$, and that `CountOne_1` has the added benefit that the number of additions it performs will on average be half the number done by `CountOne_2`. Therefore, these algorithms might seem to be equal candidates for conversion to an S2P environment, with `CountOne_1` likely to be somewhat faster than `CountOne_2` in practice (since it uses tail recursion and makes half as many calls on average to an addition circuit), but this is not the case.

In a typical S2P environment, the array S is shared by two parties, Alice and Bob, with each bit of the array being the exclusive-or of two bits a and b, where a is known to Alice and b is known to Bob. Alice and Bob want to find out the number of ones in the array S without letting the other know the actual contents of S. Using Yao's garbled circuit paradigm, we can easily turn the above algorithms into S2P protocols. However, the running time of the resultant protocols will be quite different.

For the first algorithm, the outcome of the second `if-else` (and the fact that one of its branches performs an addition and the other doesn't) cannot be disclosed to anybody during the computation; otherwise, that party will know the value of $S[i]$, which is considered as private information disclosure in typical S2P computation circumstance because no party is supposed to know any real value in array S. To disguised the outcome of the `if-condition`, therefore, the circuit has to execute both branches of the second `if-else`. Unfortunately, this increases the complexity of the algorithm from $\Theta(n)$ to $\Theta(2^n)$, since it requires the duplication of the circuit that counts the number of ones in the first $n-1$ positions of S. That is, compiling `CountOne_1` for S2P converts an efficient linear-time computation into an exponential-sized circuit that Alice and Bob must then painfully evaluate using masked versions of their respective data values. The second algorithm does not have the problem, however, since it has no conditions that depend on any values of S, so the complexity of the resulting circuit is $O(n)$.

The dramatic difference in complexity between the S2P versions of these two algorithms is caused by the fact that the second algorithm is *data-oblivious*, while the first one is not. Recall that an algorithm is data-oblivious [21] if the control flow of the algorithm does not depend on any data values. In the case of `CountOne_2`, the only part of the algorithm that needs to "know" the actual values of S is the low-level summation operator. The higher-level portions of the algorithm needs not "know" anything about the low-level summation operations other than the fact that they are performing their respective tasks correctly.

In addition to being a graphic demonstration of the inefficiencies that can come from wholesale applications of Yao's garbled circuit paradigm, the above example also serves as a motivation for an alternative approach to S2P computations, which we call *bureaucratic* computing. In the bureaucratic approach, all the high-level steps of an algorithm to evaluate the function, $f(X, Y)$, of interest to Alice and Bob, are done in a data-oblivious way; that is, the high-level steps are non-adaptive with respect to the data values, X and Y. All the low-level computations that depend on the data values are isolated to simple "black-box" functions, like SUM, MIN, MAX, compare-exchange, etc., which can be efficiently simulated in a zero-knowledge fashion using Yao's garbled circuit technique. In simulating a bureaucratic algorithm in the S2P framework, Alice and Bob collectively perform each high-level step using their cryptographically masked variables, and they only engage in the simulations of Boolean circuits for the low-level steps that involve their inputs being "fed" into one of the low-level black-box functions.

1.2 Our Results and Contributions

In this paper, we demonstrate the utility of the bureaucratic approach to S2P computations through case studies involving three problems—sorting, random permutation and selection (i.e., finding the k^{th} smallest or largest element). These problems have been extensively studied in the non-S2P environment, and many efficient algorithms for them are taught in undergraduate computer science classes (e.g., see [10, 16]). However, as we have discussed, the time complexities in the non-S2P environment and in the S2P environment are not necessarily the same. To the best of our knowledge, there has been little previous attention to methods for developing practical algorithms for these problems in the S2P environment, for which we are advocating the bureaucratic approach in this paper.

Specifically, we study two bureaucratic algorithms for sorting from the perspective of S2P computations: one algorithm leverages research from sorting networks and the other uses a randomized-algorithms approach. Both algorithms use compare-exchange as a low-level primitive, with all other (high-level) steps being done in a data-oblivious manner. Incidentally, previous studies of sorting networks were motivated by a totally different reason (hardware design), but we find out that the results of sorting networks are nevertheless applicable to our problem, since they give rise to

efficient bureaucratic algorithms. Using existing results on sorting networks, we can achieve an $O(n \log^2 n)$ time complexity for S2P sorting, with a constant factor that is only 0.25. We also study another interesting algorithm that lowers the time complexity to $O(n \log n)$, with a constant 5. This algorithm is also data-oblivious, but it is probabilistic, guaranteeing with very high probability that its final result is sorted. As we show, this algorithm also leads to efficient S2P implementations.

Building upon these bureaucratic S2P sorting algorithms, we propose some immediate applications: in particular, to selection algorithms and random permutation algorithms for the S2P environment.

The contribution of this paper is two-fold: (1) We have developed a new paradigm, called bureaucratic computing, for solving S2P computation problems. (2) We have developed efficient algorithms for conducting sorting, selection, and random permuting in the secure two-party computation environment.

2. BUREAUCRATIC COMPUTING

2.1 The Framework

A standard S2P computation involves two parties, each having his/her own secret inputs. Without loss of generality, we assume that each party's input is an array. These two participants want to run a protocol to evaluate a function (e.g. vector dot product) or run an algorithm (e.g. sorting) on their joint inputs. At the end of the protocol, neither party can learn anything other than what can be derived from the results and their own private inputs.

As we have pointed out in Section 1, although a wholesale application of Yao's garbled circuit paradigm can solve general S2P computation problems theoretically, the resultant protocols might not be practically applicable, even if the Yao's circuit is built upon an algorithm that is optimal in the non-S2P environment.

We propose a new paradigm, called *bureaucratic computing*. It consists of two levels of computations. The low-level computations depend on the data values from both parties. The computations must be *oblivious*, in the sense that no participant should be able to infer anything about the intermediate results of the computation or other party's data. We can use Yao's garbled circuit technique to build S2P circuits for these low-level computations. We call these circuits *components* in this paper.

The high-level computations in our bureaucratic computing paradigm consists of the procedures (or algorithms) of how to invoke the low-level components. These computations must be *data oblivious*, meaning that the algorithms are non-adaptive with respect to the data values; namely, regardless of what the results of the low-level computations are, the behavior of the high-level computations stays the same. This data-oblivious property is very important; otherwise, a participant can infer the other party's private inputs from the observable behavior of the algorithms. Data-obliviousness guarantees that the behavior of an algorithm is independent from its inputs.

2.2 Low-Level Components

Low-level components serves as building blocks to S2P computations. In our bureaucratic computing paradigm, these components should be oblivious; more specifically, they should have the following properties: first, outputs of the components cannot be revealed to any participant. Because the outputs of the low-level components are usually intermediate results, not the final results, S2P computations do not allow them to be disclosed (recall that in S2P, participants can only learn whatever can be derived from the final results and their own private inputs, and nothing else). Second, the inputs of the low-level component might be intermediate results from other components; because of the way how the other components' outputs are protected, nobody (Alice or Bob) knows the actual inputs.

Therefore, when building Yao's garbled circuit for low-level computation components, we adopt the circuit layout depicted in Figure 1.

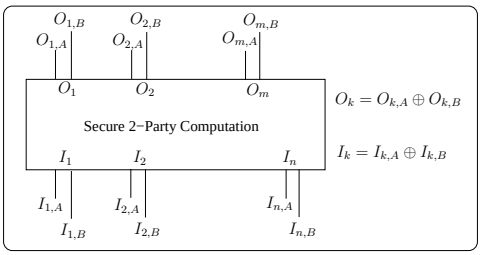


Figure 1: Circuit Layout

Outputs: To protect intermediate results in Yao's garbled circuit, all intermediate results are distributed to Alice and Bob using a secret sharing scheme, i.e., the actual output is split into two pieces, each going to one party. Nobody can derive any useful information about the secret outputs based on their own share alone; they need to put both pieces together to reveal the actual secrets. Since in S2P computation, we only have two participants, we use a simple secret sharing method, exclusive-or (`xor`). Namely, each secret r is split into two random pieces a and b, such that $r = a \oplus b$, where a is disclosed to Alice and b is disclosed to Bob.

Therefore, each output pin of our circuit (we may have many output pins depending on the actual computations) consists of two outputs, one goes to Alice and the other goes to Bob. This way, nobody knows the actual output of this circuit. If this circuit is the last step of the entire algorithm or function, Alice and Bob can disclose their secret shares to each other to obtain the final results. They cannot do so if the output is not the final results, due to privacy concerns.

Inputs: If our circuit is used as a component, the inputs of the circuit are likely outputs from other components. Therefore, according to the ways how outputs are protected, each input pin of our circuit also consists of two inputs, one from Alice and the other from Bob; the actual input is the xor of these two inputs. More specifically, in our circuit design, we assume that Alice holds her shares $A = \{a_1, a_2, \ldots, a_n\}$, and Bob holds his shares $B = \{b_1, b_2, \ldots, b_n\}$. The real secret values $R = A \oplus B$, i.e., $R = \{r_i = a_i \oplus b_i \mid i = 1,\ldots, n\}$, where r_i's are never known to any single party.

The above setting is different from the setting of traditional S2P problems. In the traditional setting, Alice and Bob have their own private inputs, and they want to do a joint computation on their joint inputs. The traditional setting can be considered as a special case of our setting. For example, if Alice's private inputs are $r_1, \ldots, r_k$,

while Bob's private inputs are $r_{k+1}, \ldots, r_n$, we can consider that Alice has $A = \{r_1, \ldots, r_k, 0, \ldots, 0\}$, and Bob has $B = \{0, \ldots, 0, r_{k+1}, \ldots, r_n\}$, as we know that any number xor with 0 stays the same.

2.3 Secure Compare-Swap Component

We have developed a useful component that is called *Secure Compare-Swap (SCS)*; it serves as the low-level component for our bureaucratic protocols for sorting, selection, and random permutation problems. According to bureaucratic computing paradigm, we use Yao's garbled circuit technique to implement this component. The layout of this component follows the general circuit layout depicted in Figure 1.

The functionality of the SCS circuit is to compare two input numbers L and R, such that the larger one is output from the `Max` pin, while the smaller one is output from the `Min` pin. The actual input L is the `xor` of L_A and L_B (R is the `xor` of R_A and R_B), where L_A and R_A belong to Alice, while L_B and R_B belong to Bob. Neither Alice nor Bob knows L or R.

Outputs of the SCS component are most likely intermediate results, so they should not disclose any information about the inputs. Therefore, we need this primitive to be not only secure (i.e. disclosing no information about the value of the inputs), but also *oblivious*, i.e., from the output, nobody should be able to tell which number (L or R) becomes the `Max` output and which one becomes the `Min` output. To put it in another way, nobody should know whether the "swap" has occurred within the circuit or not.

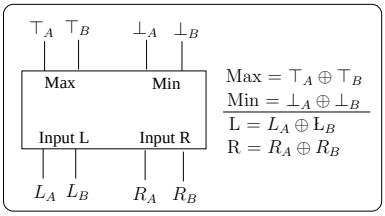


Figure 2: Secure Compare-Swap (SCS)

It should be noted that every time the SCS circuit is used, the random numbers generated to disguise the outputs must be fresh, and cannot be constants. Therefore, although we use the same component multiple times in a protocol, the outputs are independently disguised, and there is no information disclosure due to the repeated use of the component.

The SCS circuit can be easily built from the scratch or using Fairplay [25].

2.4 High-Level Computations

The high-level computations in the bureaucratic computing are concerned about how to assemble the low-level components together to solve a specific problem. To prevent partial information disclosure, the high-level computations should be data-oblivious.

There are two different ways to achieve data-obliviousness. One is to add redundant steps to turn a non-data-oblivious algorithm into an data-oblivious one. For example, an algorithm might need need to use $a[i]$ at a specific step, and the value i depends the results of previous steps. This algorithm is not data-oblivious. In S2P computation, letting either party learn the value of i is unacceptable. It is not difficult to hide the value of i using Yao's garbled circuit technique. The problem is if i is unknown, how can the participants know which element of the array should be used at this step. To solve this problem, a standard technique is to "pretend to" use the entire array, but making sure that only the value of $a[i]$ actually affects the outcome [25]. The computations of other array elements are redundant and are for disguise purpose. These redundant computations increase the time complexity of non-data-oblivious algorithms.

An alternative to achieve data-obliviousness in high-level computations is to directly use a *data-oblivious algorithm*. In a data-oblivious algorithm, if $a[i]$ needs to be used at a specific step, the value i must be independent from the results of previous steps, i.e., i is known even before the whole computation. Therefore, disclosing the value i poses no risk. For example, `bubblesort` is a data-oblivious algorithm. Figure 3 depicts the steps of `bubblesort` for three numbers. Regardless of what the inputs are, the steps depicted in the figure are always the same. Because there is no need to add redundant computations, converting a data-oblivious algorithms to S2P computations does not change the time complexity of the algorithm.

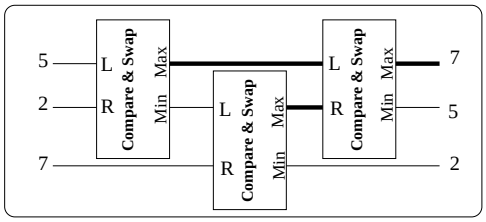


Figure 3: Example of `bubblesort` of 3 numbers

Regardless of what approach we take, the end results are data-oblivious algorithms. Therefore, the problem that we should solve is to develop optimal data-oblivious algorithms and use them for the high-level computations in the bureaucratic computing. In the subsequent sections, we will focus on developing such algorithms.

2.5 From Oblivious Algorithm to S2P Circuit

Once we have a data-oblivious algorithm for high-level computation and the necessary low-level components, converting the algorithm to a S2P circuit is straightforward. To help readers understand the procedure, we use the `bubblesort` of three numbers as an example.

The comparison sequence of `bubblesort` for three numbers are depicted in Figure 3. This sequence is data-oblivious, so the sequence itself (i.e. the high-level steps) reveals nothing about the contents of inputs. However, the low-level "compare & swap" component does reveal information about the inputs because the behavior of the component depends on the inputs. To ensure that no information about the "compare & swap" step is disclosed, we replace this component with the SCS component (Figure 2), which uses Yao's garbled circuit technique to hide the behavior inside the component. Moreover, the component is reusable like a function of a program, so the whole bureaucratic computing is quite scalable.

Since it is straightforward, we will not discuss how to convert data-oblivious algorithms into S2P circuit again in the rest of this paper. We will only focus on the discussion of the date-oblivious algorithms themselves.

3. SECURE TWO-PARTY SORTING

3.1 The S2P Sorting Problem

Sorting is a fundamentally important procedure that serves as a critical subroutine to many solutions. Therefore, to provide solutions to many interesting S2P problems, it is very important to be able to conduct sorting in the S2P setting, where the actual array to be sorted consists of the private inputs from two different parties, and no private inputs should be disclosed to the other party.

We would like to build an efficient S2P circuit for sorting using the bureaucratic computing paradigm. We call this circuit the *S2P sorting circuit*. We would like the S2P sorting circuit to not only serve as a complete solution itself, but also serve as a S2P sub-function to other more sophisticated problems. Therefore, the layout of our circuit follows what we depicted in Figure 1. We formally define the requirement of our S2P sorting circuit in the following:

DEFINITION 3.1. *(S2P Sorting Circuit) The objective of this sorting circuit is to sort the input $I = \{I_1, \ldots, I_n\}$. However nobody knows the actual input array; instead, each participant has a secret share of this array. Namely, Alice has $\{I_{1,A}, \ldots, I_{n,A}\}$, while Bob has has $\{I_{1,B}, \ldots, I_{n,B}\}$, where $I_k = I_{k,A} \oplus I_{k,B}$, for $k = 1, \ldots, n$.*

The actual output of the circuit is a sorted array $O = \{O_1, \ldots, O_n\}$, where $O_1 \leq O_2 \leq \ldots \leq O_n$. No party should learn these actual outputs; instead, Alice learns $\{O_{1,A}, \ldots, O_{n,A}\}$ and Bob learns $\{O_{1,B}, \ldots, O_{n,B}\}$, where $O_k = O_{k,A} \oplus O_{k,B}$, for $k = 1, \ldots, n$.

Because this is a S2P computation, from the evaluation of this S2P sorting circuit, nobody should be able to derive any useful information about the input array I and the output array O, other than what they have already learned before the evaluation of this circuit [3].

3.2 Challenges and Approaches

As we known, sorting has been extensively studied in the non-S2P setting; many sorting algorithms have been proposed, such as `quicksort`, `mergesort`, `bubblesort`, etc. The lower bound on comparison-based sorting algorithm is $O(n \log n)$, which is achieved by a number of sorting algorithms. We have studied the common sorting algorithms that achieve $O(n \log n)$ time complexity (either in worst case or in average case). Unfortunately, these algorithms are not data-oblivious; none of them can beat the `bubblesort` asymptotically (i.e., they are no better than $O(n^2)$) in secure sorting cases where time complexity may change (see the countone example in introduction).

Recall that mainstream comparison-based algorithms, such as `quicksort`, `mergesort`, `heapsort`, etc, must know where the compare-swap occurs, so that they can perform further sorting based on the previous result. Iterating in such a manner, however, is considered to disclose the mapping information in our secure two-party sorting case. Because the positions that compare-swap occurs are dependent on the input data. In other words, those positions are variables in each iteration. Directly using such sorting algorithms will put us in a paradox, because compare-swap positions are key part in sorting, however, we must somehow leave Alice and Bob unknown about them in the whole sorting process.

[3]If the output is the final result, then both parties are supposed to learn the output. In this case, they will disclose their private shares of the outputs to each other.

Our Approaches: We have identified two types of sorting algorithms that are efficient and data-oblivious. One type of algorithms come from the literature of *sorting networks*, which have been studied extensively for the purpose of efficient hardware design. The best practical results can achieve $O(n \log^2 n)$ time complexity. Not satisfying with this asymptotic result, we have developed a new algorithm called `Randomized Shellsort`; it achieves $O(n \log n)$ time complexity. This algorithm is a probabilistic sorting algorithm, i.e., it can sort any array with very high probability.

3.3 The S2P Sorting Networks

Data-oblivious sorting algorithms have been extensively studied in the literature. The studies were motivated by a totally different reason: a data-oblivious sorting algorithm always makes the same comparisons, regardless of the input. That is, the behavior of the algorithm is independent of the input. This property is useful for hardware design (i.e. design a hardware sorting module or a switching network), for parallel computing, and for sorting that uses external memory (such as disks and tapes), all of which can benefit if the algorithm is data-oblivious [4].

The data-oblivious sorting algorithms are usually called *sorting networks* in the literature. A sorting network is an abstract mathematical model of a network of wires and comparator modules that is used to sort a sequence of numbers. Each comparator connects two wires and sort the values by outputting the smaller value to one wire, and a larger value to the other. By arranging these comparators properly, every input element will be sent to its final position in the sorted sequence. Figure 3 gives an example of sorting networks for three numbers.

Although sorting networks were not motivated by secure two-party computation, S2P computation gives sorting networks a brand new life: sorting networks are data-oblivious, which is exactly the property that we need for bureaucratic computing. Therefore, we can leverage the optimal results developed from the sorting network community to build efficient S2P sorting circuits. Here optimal means to sort n number of input, using least number of comparators (or least number of SCS circuits for S2P sorting circuits).

Efficient Sorting Networks. The asymptotically best sorting network, discovered by Ajtai, Komlós, and Szemerédi, achieves $O(n \log n)$ for n inputs [2]. This network is called AKS network, and it is asymptotically optimal. A simplified version of the AKS network was described by Paterson [26]. While an important theoretical discovery, the AKS network has little or no practical application because of the large linear constant hidden by the Big-O notation.

Currently, optimal sorting networks for $n \leq 8$ are known exactly and are presented in [21] along with the most efficient sorting networks to date for $9 \leq n \leq 16$. These results are listed in Table 1.

For input size $n > 16$, no optimal network have been discovered. However, several sorting networks with size $O(n \log^2 n)$ have been proposed, such as `odd-even mergesort` and `bitonic sort` [4]. These networks are often used in practice. These algorithms are attached in Appendix C. Among these three sorting networks, the `odd-even mergesort` has the smallest constants in the Big-O notation. For example, to sort

Input Size n	1	2	3	4	5	6	7	8
# of Comparators	0	1	3	5	9	12	16	19
Input Size n	9	10	11	12	13	14	15	16
# of Comparators	25	29	35	39	45	51	56	60

Table 1: Best results for sorting networks

n inputs, the actual number of comparators needed by the `odd-even mergesort` is $\frac{1}{4}n\log^2 n - \frac{1}{4}n\log n + n - 1$.

3.4 The S2P Randomized Shellsort

As we discussed in the previous subsection, practical sorting networks have $O(n\log^2 n)$ time complexity. In this subsection, we discuss an asymptotically better sorting algorithm, called *randomized* Shellsort.This algorithm achieves $O(n\log n)$ time complexity and correctly sorts with very high probability.

Shellsort. Shellsort defines a gap sequence (also called offset sequence) $H = \{h_1, \ldots, h_m\}$, where m is a predefined value. The performance of Shellsort depends on this gap sequence. A number of good sequences have been proposed through empirical studies, including the geometric sequence (i.e. $n/2$, $n/4$, ..., 1) [29], the Ciura sequence [9], and the Fibonacci sequence [28]. In this paper, we only use the geometric sequence.

For each sequence number h_i, Shellsort divides the input array (of size n) into $\frac{n}{h_i}$ regions, and use the insertion sort algorithm to sort the array consisting of the j-th element from each region, i.e., $A[j]$, $A[j+h_i]$, ..., $A[j+m*h_i]$. for $j = 0, \ldots, h_i - 1$. This step is called `h-sort`. The psudocode of Shellsort is given in Figure 4.

```
Shellsort
Input: The n-element array A that to be sorted
Output: The sorted array A
foreach h ∈ {h_1, ..., h_m} do
    for i = 0 to h − 1 do
        Sort A[i], A[i + h], A[i + 2h], ..., using insertion sort;
        /* This inner loop is h-sort */
    end
end
```

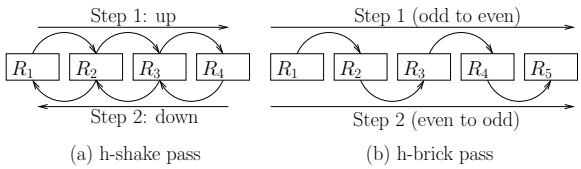
Figure 4: The original Shellsort Algorithm

Unfortunately, the `h-sort` step, i.e. the insertion sort, is not data-oblivious, and cannot be efficiently converted into Yao's garbled circuit. We need to replace the insertion sort with an efficient data-oblivious algorithm.

Shake and Brick pass Three interesting operations have been studied in the context of Shellsort: `h-bubble` pass, `h-shake` pass, and `h-brick` pass. All these operations are *data-oblivious*, suitable for our bureaucratic computing.

An `h-bubble` pass move from left to right through the array, compare-exchanging each element with the one h to its right. This is like one iteration of the bubble sort. An `h-shake` pass is an `h-bubble` pass followed by a similar pass in the opposite direction, from right to left through the array, compare-exchanging each element with the one h to its left [19]. Figure 5(a) gives the concrete example of a `h-shake` pass. The `h-shake` pass pushes the large numbers toward the right and small numbers towards the left.

`H-brick` pass is another interesting operation [28]. Within this pass, elements in positions i, $i+2h$, $i+4h$, $i+6h$, ... are compare-exchanged with items in positions $i+h$, $i+3h$, $i+5h$, $i+7h$, ..., respectively; then items in positions $i+h$, $i+3h$, $i+5h$, $i+7h$, ..., are compare-exchanged with those in positions $i+2h$, $i+4h$, $i+6h$, $i+8h$, ..., respectively. Figure 5(b) shows how brick pass works. `H-brick` pass helps larger elements quickly jump to the right regions while smaller elements quickly jump to the left.

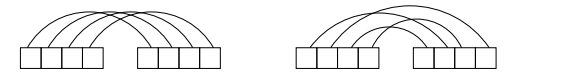


(a) h-shake pass (b) h-brick pass

Figure 5: H-Shake Pass and H-Brick Pass

Empirical results [18,23] indicate that replacing the `h-sort` in Shellsort by an `h-shake` pass or an `h-brick` pass gives an algorithm that nearly always sorts when the increment sequence is geometric. The imprecise phrase "nearly always sorts" indicates a probabilistic sorting method. That is, the method might leave some items unsorted.

Randomized Shellsort Algorithm. To further improve the sorting results, consider a *randomized region comparison* concept. To conduct compare-and-swap on two regions R_1 and R_2 (both of size L), `h-shake` pass and `h-brick` pass always conduct the operation on elements that are exactly h distance apart (see Figure 6(a)). We change this deteriministic behavior to randomized behavior, i.e., we first construct a random permutation of set $\{1, 2, \ldots, L\}$; assume that the permutation result is the set $\{i_1, i_2, \ldots, i_L\}$. For the k-th element in R_1, do a compare-swap operation with the element i_k in R_2. See Figure 6(b).



(a) Original Region Comparison (b) Randomized Region Comparison

Figure 6: Region comparisons

Combining the advantages of `h-shake` pass, `h-brick` pass, and the randomized region comparison, the Randomized Shellsort algorithm is shown in Figure 7. In this algorithm, we choose the geometry gap sequence $\{n/2, n/4, n/8, \ldots, 1\}$. For each of the gap value h in this sequence, the algorithm runs 6 loops. The first two loops are actually one `h-shake` pass (one loop from left to right, and the other from right to left). The next four loops are several `h-brick` passes: each pass compares with the region $3h$, $2h$, and h gap away, respectively. The comparison between two regions always use randomized region comparison.

Correctness Theorem. Although our algorithm is probabilistic, it achieves complete sorting with very high probability. The proof of this property is quite complicated, and is beyond the scope of this paper. We refer readers to [17] for more details. Thus, we only state the following theorem of correctness, for which we provide a reference in the non-anonymous version of this paper. In Section 6, we use empirical studies to demonstrate the validity of this theorem.

```
Randomized Shellsort
Input: The n-element array A that to be sorted
Output: The sorted array A
Offset = {n/2, n/4, ..., 1};
foreach h ∈ Offset do
    for i ← 0; i < n − h; i ← i + h do
        compareRegions(a, i, i + h, h);
    end
    for i ← n − h; i > h; i ← i − h do
        compareRegions(a, i − h, i, h);
    end
    for i ← 0; i < n − 3 ∗ h; i ← i + h do
        compareRegions(a, i + 3 ∗ h, h, h);
    end
    for i ← 0; i < n − 2 ∗ h; i ← i + h do
        compareRegions(a, i + 2 ∗ h, h, h);
    end
    for i ← 0; i < n; i ← i + 2 ∗ h do
        compareRegions(a, i, i + h, h);
    end
    for i ← h; i < n − h; i ← i + 2 ∗ h do
        compareRegions(a, i, i + h, h);
    end
end

procedure compareRegions(a, start1, start2, offset)
1: array target ← 0 to offset - 1
2: RandomPermute(target);
3: for i ← 0 to offset - 1
4:     CompareSwap(a, start1 + i, start2 + target(i));
```

Figure 7: The Randomized Shellsort Algorithm

THEOREM 1. *The Randomized Shellsort algorithm can correctly sort any input array of size n with very high probability. The term "very high probability" means that the failure probability is at most $O(n^{-\alpha})$, for some constant $\alpha > 1$, where failure means that the number of inverse ordered element pair within the output array is at least one.*

Time Complexity Analysis The time complexity of the Randomized Shellsort algorithm is the following (the proof is in Appendix A):

THEOREM 2. *To sort n inputs, the number of compare-swap operations in the Randomized Shellsort is the following:*

$$T(n) = 5n \log n - \frac{15}{2}n + 8. \quad (1)$$

3.5 A Further Improvement

Although Randomized Shellsort can reduce the time complexity to about $5n \log n$, in practice, this cost might still be quite expensive, due to its constant. This is because unlike non-S2P settings, each comparison in the S2P setting is quite expensive. Therefore, it is quite desirable for S2P computation if we can further reduce the cost.

In practice, there may be some applications that accept "almost sorted" array as "sorting" result, as long as the number of miss-placed elements is small and their offsets (compared to their actual places in the completed sorted array) is not too far off. This observation can help us reduce the number of loops in the Randomized Shellsort algorithm.

From our empirical studies, we have discovered the following properties in the Randomized Shellsort algorithm: (1) If we only keep the first two loops (i.e. the `h-shake` pass), the number of misplaced elements is quite small (less than 4.5% for $n \leq 2^{16}$. (2) Among these miss-placed elements, the majority of them is miss-placed by one position. Fortunately, these missing-one mistakes can be corrected by a single `bubble` pass (i.e., one iteration in `bubblesort`), which takes n comparisons. Based on these two properties, we modified the Randomized Shellsort algorithm, and developed a new algorithm called *Fast Randomized Shellsort*. The algorithm is depicted in Figure 8.

```
Fast Randomized Shellsort
Input: The n-element array A that to be sorted
Output: The sorted array A
Offset = {n/2, n/4, ..., 1};
foreach h ∈ Offset do
    for i ← 0; i < n − h; i ← i + h do
        compareRegions(a, i, i + h, h);
    end
    for i ← n − h; i > h; i ← i − h do
        compareRegions(a, i − h, i, h);
    end
end
for i ← 0; i < n − 1; i ← i + 1 do
    CompareSwap(a, i, i + 1);
end
```

Figure 8: The Fast Randomized Shellsort Algorithm

The time complexity of this algorithm is improved from $5n \log n$ to $2n \log n$, an improvement of over 60%. This improvement is achieved at a small cost of sorting quality. We will provide detailed evaluations in Section 6.

4. SECURE TWO-PARTY SELECTION

4.1 The S2P Selection Problem

Selection is another important problem in computer science. A selection algorithm is an algorithm for finding the k-th smallest (or largest) number in an array. Selection algorithms are widely used in many applications. It is important to be able to conduct selection in the S2P environment, where the actual input array consists of the private inputs from two different parties.

We would like to build an efficient S2P circuit for selection. Similar to what we did to S2P sorting circuit, we would like this circuit to not only serve as a complete solution itself, but also serve as a component of the solutions to other more sophisticated problems. Therefore, the layout of our circuit follows what we depicted in Figure 1, except that the output only consists of two pins, which are the two secret shares of the actual output, the k-th smallest number. We formally define the S2P selection circuit in the following:

DEFINITION 4.1. *(S2P Selection) The input of the S2P selection circuit is an array $I = \{I_1, \ldots, I_n\}$. Nobody knows the actual input array; instead, each participant has a secret share of this array. Namely, Alice has $\{I_{1,A}, \ldots, I_{n,A}\}$, while Bob has has $\{I_{1,B}, \ldots, I_{n,B}\}$, where $I_k = I_{k,A} \oplus I_{k,B}$, for $k = 1, \ldots, n$.*

The actual output of the circuit is denoted as O, which is the k-th smallest number in I (k is public to both participants). No party should learn these actual value of O; instead, Alice learns O_A and Bob learns O_B, where $O = O_A \oplus O_B$.

4.2 Challenges

It is well-known that selection can be achieved in linear time for general k [5, 21]. Similar to `quicksort`, these linear time algorithms use a pivot to partition the input array, and then conduct the recursion on one of the partitions. As we have already discussed before, the recursion (i.e. subsequent comparisons) after partition depends on the partition results, which depend on the actual input. Therefore, just like `quicksort`, these linear time algorithms are not data-oblivious. If we use Fairplay to build a Yao's garbled circuit directly from these algorithms, the time complexity will be increased to $O(n^2)$.

Sorting-Based Approach. A naive solution is to directly apply the S2P sorting circuit on the input, and then output the k-th smallest number as the result. This way, we can achieve the $O(n \log^2 n)$ time complexity using sorting networks or $O(n \log n)$ using our proposed probabilistic sorting algorithms.

However, we have observed that using sorting to achieve selection basically requires us to do more work than what is actually needed. In the selection problem, we are only interested in ensuring that the k-th smallest element is in its correct position; whether other elements are in correct positions is not important. When we use sorting algorithms, we have to do extra work by putting the other $n-1$ elements in their correct places; this becomes overhead. The challenge is whether we can reduce the amount of overhead.

Another observation that we make is that in the selection problem, many applications might not demand that the final output is strictly the k^{th} smallest; small errors are often tolerable. For example, if the requirement is to find the median, but the result turns out to be $(\frac{n}{2}-1)$-th smallest, the results are acceptable to many applications. If we are building S2P selection circuits for this type of applications, we should be able to simplify our circuits by sacrificing a little bit of accuracy. Based on this motivation, we have modified our Fast Randomized Shellsort algorithm for selection, and reduced the running time by 50%. Asymptotically, our new algorithm runs in $\Theta(n \log k)$ time, when selecting the k^{th} smallest element in n inputs.

4.3 A fast selection algorithm

The objective of our algorithm is to construct a series of compare-swap steps, after which the k^{th} smallest element (called the *target*) of the input array A is put in $A[k]$. To avoid the overhead of a complete sorting, we would like to achieve the following goals: (1) Elements smaller than the target should be located to the left of the target. (2) Element larger than the target should be located to the right of the target. We are not concerned about the actual positions of the element other than the target.

We divide the input array into $\lfloor \frac{n}{k} \rfloor$ groups. The group containing the k^{th} position is called the target group. We refer positions left to the target the "left part", and right ones the "right part". To achieve the goals above, we first go through a `bubble` pass from the rightmost group to the target group, after which, smaller elements on the right side of the target group will be moved to the left. Second, we run another `bubble` pass from the first group to the group next to the target group. This step guarantees to move larger elements of the left part into the right part of the k^{th} position. Third, after these two passes, we make region comparisons between the elements in the target group (only those to the right of the k^{th} position) and the elements in other groups on the right side. [4] In the last step, we compare the two groups on both side of the target. This pass provides more opportunities for smaller elements to jump directly into the left part. Figure 9 illustrates the above steps.

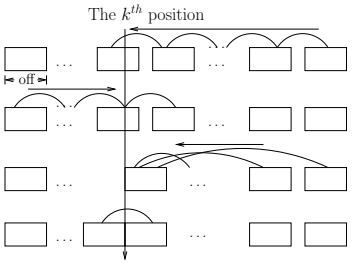


Figure 9: Steps in the Fast Selection Algorithm

We repeat the above four steps for a different offset value h that is half of the previous offset value, until $h = 1$. At this time, all elements that are smaller or equal to our target have been move to the left part with very high probability. We simply add one more `bubble` pass to extract the target. The algorithm is depicted in Figure 10.

```
Fast Oblivious Selection
Input: The n-element array A and k.
Output: The k^th smallest element in A
Offset = {k, k/2, k/4, ..., 1};
foreach offset h ∈ Offset do
    group ←k/h;
    for i ← n − h; i ≥ group*h; i ← i − h do
        compareRegions(a, i − h, i, h, rand);
    end
    for i ← 0; i ≤ group*h; i ← i + h do
        compareRegions(a, i, i + h, h, rand);
    end
    for i ← n; i ≥ k+h; i ← i − h do
        compareRegions(a, i − h, k, h, rand);
    end
    compareRegions(a, k-h, k, h, rand);
end
for i ← 1; i ≤ k; i ← i + 1 do
    CompareSwap(a, i, i + 1);
end
for i ← n; i > k; i ← i − 1 do
    CompareSwap(a, i, i − 1);
end
Output a_k
```

The `compareRegions` and `CompareSwap` functions are the same as the Randomized Shellsort.

Figure 10: The Fast Selection Algorithm

This algorithm can find the k-th smallest element with high probability. Our empirical studies show that the error rate is very low. More details will be given in Section 6. The time complexity of this selection algorithm is given in the following:

THEOREM 3. *To find the k-th smallest element in an array of size n, the number of compare-swap operations $T_{n,k}$ is bounded by the following inequality:*

$$(2n-k)\log k + n - 4k + 4 \leq T_{n,k} \leq (2n-k)\log k + 3n + k - 2.$$

[4] We assume that $k \leq \frac{n}{2}$. If $k > \frac{n}{2}$, the region comparisons will be conducted on the left side. The goal is to conduct the comparisons on the longer side.

The proof is given in Appendix B. The theorem indicates that the time complexity of the algorithm is $\Theta(n \log k)$.

5. S2P RANDOM PERMUTING

5.1 The S2P Random Permuting Problem

A random permutation is a random ordering of a set of elements. It often serves as a building block to many randomized algorithms, such as those in coding theory, cryptography, and simulation. A good random permuting algorithm should be able to generate all of the $n!$ permutations with an equal probability.

We would like to build a S2P random permuting circuit. The layout of the circuit is exactly the same as the layout depicted in Figure 1. Namely, the inputs of the circuit are shared by Alice and Bob (nobody knows the actual inputs). The output of the circuit is a random permutation of the inputs. Neither Alice nor Bob knows how the inputs are permuted.

5.2 Random Permuting Algorithms

Like sorting networks, there are permuting networks. Goldstein and Leibholz proposed a method in [15]. Its cost is $(p-1) \cdot 2^p + 1$, for 2^p items. Although this algorithm is data-oblivious, the distribution of the results is not uniform. Namely, the information of the inputs is disclosed to certain degree.

Another well-known random permuting algorithm is referred to as Knuth shuffle [20]. It has been proven that Knuth shuffle generates all $n!$ permutations with uniform distribution, as long as the random number is uniformly generated. To shuffle an array with size n, the algorithm takes $n-1$ round. In each round (say round i), the algorithm generates a random number $r \in \{1, \ldots, n-i\}$, and exchange the i-th element of the array with the r-th element.

Knuth shuffle runs in $O(n)$ time, but unfortunately, it is not a data-oblivious algorithm, because the sequence of comparisons depends on the value of r. To use Knuth shuffle to build a S2P permuting circuit, the value of r cannot be disclosed to anybody; otherwise how the inputs are permuted is partially disclosed. Introducing decoys to hide r increases the time complexity to $O(n^2)$.

5.3 A Sorting-Based Permuting Algorithm

Permuting can be achieved using sorting. The idea is quite simple: in order to randomly permute an array, we expand each element (say a_i) of the array into a tuple (r_i, a_i), where r_i is a randomly generated number. We then sort this array of tuples using the random numbers as the key. After sorting, the elements (a_i's) of the original array are randomly permuted.

The random numbers r_i's are generated jointly by Alice and Bob, i.e., each of them generate their own shares, the actual random numbers used for sorting is the `xor` of their shares. Nobody knows the actual random numbers. Our S2P sorting circuit allows the sorting without each party knowing the actual inputs.

It has been proven that if there are no duplicates in the generated random numbers, the permutations is uniformly random [20].

6. EVALUATION

This section gives a comprehensive empirical study of our S2P sorting and selection algorithms. Because our permuting algorithm is based on sorting, we will not describe its evaluation results. All the experiments are run on Intel(R) Pentium(R)-D machines with 3.00 GHz CPU and 4GB of physical memory. Our implementation is in Java. We use Fairplay [25] to build our compare-swap primitive.

6.1 S2P Sorting Circuit

In this section, we evaluate the performance of S2P circuits based on sorting networks and Randomized Shellsort. Figure 11(a) and 11(b) show the performance of S2P sorting circuits on a local machine and a local area network, respectively. All the evaluated circuits can finish S2P sorting within reasonable amount time, except Shellsort, because Shellsort's inner loop is not data-oblivious and its circuit runs out of memory when $n > 64$.

From these two figures, we can find out that when the array size is not too large, the Odd-Even Mergesort performs the best. Although this algorithm has $O(n \log^2 n)$ time complexity, it has a small constant (0.25). That is why it is even better than the randomized Shellsort algorithm (with time complexity $(n \log n)$). However, Randomized Shellsort becomes better when n is large (see Figure 11(c)). In this figure, we uses the number of compare-swap as the Y-axis, because the running time for the evaluation of one compare-swap primitive is constant (0.766 second for a local machine and 0.901 second over a LAN).

Randomized Shellsort is a probabilistic algorithm, which achieves sorting with very high probability. To evaluate this, we have conducted experiments using randomly generated array of various size. We ran the algorithm [5] over one million times for each input size, ranging from 2 to 2^{20}. We have found no sorting error in the results.

6.2 Fast Randomized Shellsort

We evaluate the improvement achieved by our Fast Randomized Shellsort. Figure 11(c), the asymptotic plot, demonstrates the savings compared to Randomized Shellsort. This saving is quite significant; it is achieved by allowing a small amount of sorting errors. To see how much the sacrifice is, we conducted our evaluation on randomly generated arrays of different sizes. We run our experiment $10,000$ times for each size, and plot the average results in Figure 12.

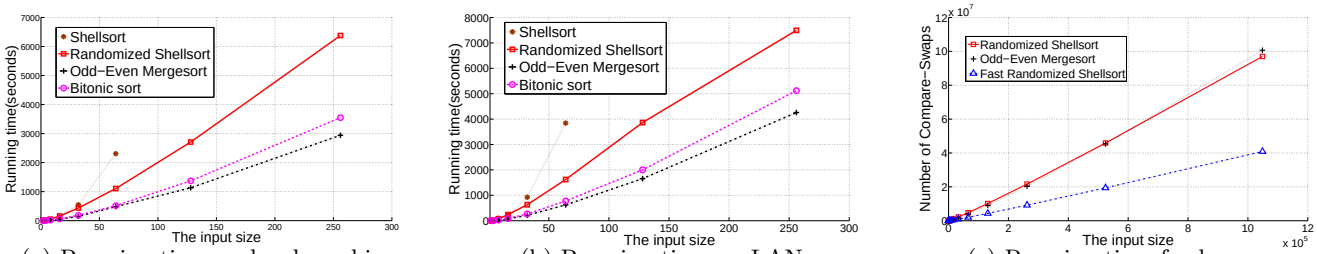
From Figure 12, we can see that the error rates (i.e., the portion of elements that are not placed in correct positions after sorting) are very low, i.e., although the whole array is not completely sorted, only a small percent of elements are in wrong positions. Moreover, among all mis-located elements, most of them are only off by one position (i.e., error distance = 1), and very small number of elements are off by two positions. No element in our experiments is off by more than two positions.

6.3 S2P Selection Circuit

Figure 13 compares our selection algorithm with the sorting-based selection. The improvement over the running time is quite significant. This improvement is achieved at the cost of a accuracy.

To evaluate how accurately our algorithm can find the k-th smallest item. We conducted experiments on randomly

[5]To save time, we conduct this experiment only in the non-S2P setting, as the results of this experiment does not change even if it is conducted in S2P settings.



(a) Running time on local machine (b) Running time on LAN (c) Running time for large n

Figure 11: Performance of S2P sorting algorithms

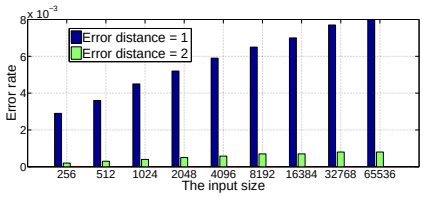


Figure 12: Error rate of Fast Randomized Shellsort

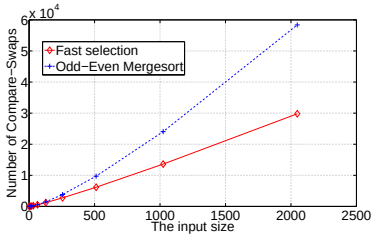


Figure 13: Performance of S2P selection algorithms

generated arrays of various sizes and with various k values. The error rates are plotted in Figure 14.

Our selection algorithm is quite accurate. Most of the errors are just off by one (i.e., instead of finding the k-th largest number, the algorithm returns the $(k-1)$-th or $(k+1)$-th largest number). The rate of this type of errors is quite small (less than 0.014%). Errors that are off by more than one position are much less.

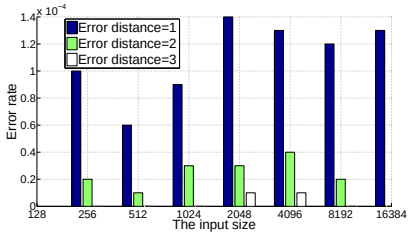


Figure 14: Error rate of the Fast Selection algorithm

7. RELATED WORK

Bureaucratic computing reveals the inherent relationship between secure two-party computation and oblivious algorithms. We briefly review these two fields in this section.

Applications of S2P can be categorized into two directions. One focuses on developing generic solution using Yao's garbled circuit [13, 14, 30]. Thanks to Fairplay [25], we can easily build Yao's circuit for a specific problem. However, this general approach is not scalable [1,24]. For example, directly compiling the `bubblesort` algorithm into Yao's circuit generates a circuit that contains $O(n^2)$ compare-swap gates, each of which requires an expensive oblivious-transfer protocol. The cost of such a generic circuit is quite large with a big n. Another direction aims at designing S2P schemes for a given function or a specific algorithm [1,3,6,8,11,24]. They take the approach of reducing the generic circuit construction into many invocations of simpler secure functions of less inputs, or sacrificing partial secret information to gain efficiency. The concept of "data obliviousness of an algorithm" has never been studied in those papers.

Our goal is different. We would like to answer which algorithm is the fastest for a specific S2P computation problem, such as sorting. Bureaucratic framework ensures that the costly effort of applying the generic garbled circuit is spent only on the atomic two-input functions.

This framework roots in the data obliviousness of algorithms. Knuth's book [21] is one of the first places to mention "oblivious algorithm" in details. In [12, 27], the authors propose "cache-oblivious" algorithms, which are appealing in hardware applications. The motivation is to design algorithms that are independent to hardware parameters, such as cache size and cache-line length. This line of work has little relationship to ours; we study data obliviousness.

Sorting network is an important research topic in hardware and parallelization research [4]. We exam most popular two of them, odd-even merge and bitonic sort. Several studies based on [4] have improved efficiency and simplicity for selection and sorting [7, 22]. Sorting networks are suitable for bureaucratic computing since they are data-oblivious. Shellsort and its variants is another widely-studied technique [19, 28]. These variants are mainly for improving performance. The best known time complexity for Shellsort variant is beyond $O(n \log n)$ [28]. To the best of our knowledge, our Randomized Shellsort is the first variant that achieves $O(n \log n)$, with a provable very high probability. Moreover, it is, by design, data-oblivious and appropriate for bureaucratic computing.

This paper is also (arguably) the first one to propose efficient S2P sorting, selection, and permutation algorithms. Although there is a secure k^{th}-ranked computation method in [1], their setting is different from ours, in the sense that it allows partial information (i.e., intermediate results) disclosure. Our solution ensures zero knowledge disclosure.

We have already summarized the work related to random permuting in Section 5. We will not repeat it here.

8. CONCLUSION

We propose a bureaucratic computing framework for algorithm design for S2P computation, and justify its efficiency on S2P sorting, selection and permutation problems. Our system is reasonably fast and will be open source. With the increasing privacy concerns, we believe that the direction to find efficient algorithms for bureaucratic computing is important. This paper makes an important initial step towards this direction. More efficient algorithms will emerge to solve other interesting S2P problems.

9. REFERENCES

[1] G. Aggarwal, N. Mishra, and B. Pinkas. Secure computation of the k^{th}-ranked element. In *In Advances in Cryptology-Proc. of Eurocrypt'04*, pages 40–55, 2004.

[2] M. Ajtai, J. Komlos, and E. Szemeredi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[3] M. J. Atallah and W. Du. Secure multi-party computational geometry. In *WADS2001: 7th International Workshop on Algorithms and Data Structures*, pages 165–179, Providence, Rhode Island, USA, August 8-10 2001.

[4] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*, pages 307–314, 1968.

[5] M. Blum, R. W. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection, 1973.

[6] R. Canetti, Y. Ishai, R. Kumar, M. K. Reiter, R. Rubinfeld, and R. N. Wright. Selective private function evaluation with applications to private statistics (extended abstract). In *Proceedings of Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.

[7] G. Chen and H. Shen. A bitonic selection algorithm on multiprocessor system. *J. of Comput. Sci. Technol.*, 4:315–322, 1989.

[8] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, Milwaukee, WI USA, October 23-25 1995.

[9] M. Ciura. Best increments for the average case of shellsort. In *International Symposium on Fundamentals of Computation Theory*, Riga, Latvia, 2001.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[11] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. Strauss, and R. Wright. Secure multiparty computation of approximations. In *Twenty Eighth International Colloquium on Automata, Language and Programming*, 2001.

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms (extended abstract). In *In Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–397. IEEE Computer Society Press, 1999.

[13] O. Goldreich. *The Foundations of Cryptography*, volume 2. 2004.

[14] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 218–229, 1987.

[15] L. J. Goldstein and S. W. Leibholz. On the synthesis of signal switching networks with transient blocking, 1967.

[16] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. Wiley, 2001.

[17] Michael T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–16. SIAM, 2010.

[18] J. Incerpi. A study of the worst case of shellsort. *Ph.D. thesis, Brown University, Dept. of Computer Science*, 1994.

[19] J. Incerpi and R. Sedgewick. Practical variations of shellsort. *Information Processing Letters*, 79:223–227, 2001.

[20] D. E. Knuth. Seminumerical algorithms. *The Art of Computer Programming*, 2.

[21] D. E. Knuth. Sorting and searching. *The Art of Computer Programming*, 3.

[22] T. Leighton, Y. Ma, and T. Suel. On probabilistic networks for selection, merging, and sorting. In *SPAA'95*, pages 106–118, Santa Barbara, CA, USA, 1995.

[23] P. Lemke. The performance of randomized shellsort-like network sorting algorithms. In *SCAMP working paper*, Institute for Defense Analysis, Princeton, NJ, USA, 1994.

[24] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Advances in Cryptology - Crypto2000, Lecture Notes in Computer Science*, volume 1880, 2000.

[25] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *In USENIX Security Symposium*, pages 287–302, 2004.

[26] M. S. Paterson. Improved sorting networks with o(log n) depth. *Algorithmica*, 5:75–92, 2005.

[27] H. Prokop. Cache-oblivious algorithms. Technical report, M.I.T, 1999.

[28] R. Sedgewick. Analysis of shellsort and related algorithms. In *ESA 96: Fourth Annual European Symposium on Algorithms*, pages 25–27, 1996.

[29] D. L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.

[30] A. C. Yao. How to generate and exchange secrets. In *Proceedings 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

APPENDIX

A. PROOF OF THEOREM 2

Here is the proof for Theorem 2, the time complexity of Randomized Shellsort.

PROOF. In every round, we have six passes. When offset equals to f, the number of compare-swap operations that

each pass takes is $(\frac{n}{f}-1)*f$, $(\frac{n}{f}-1)*f$, $(\frac{n}{f}-3)*f$, $(\frac{n}{f}-2)*f$, $(\frac{n}{2f}*f$, and $(\frac{n}{2f}-1)*f$, respectively. Note that the third, fourth, and sixth inner loop are not running at the first round. We have the total number of compare-swaps as the following:

$$\begin{aligned} T(n) &= \sum_{i=1}^{\log n}((\frac{n}{f_i}-1)f+(\frac{n}{f_i}-1)f+(\frac{n}{2f_i})f_i) \\ &+ \sum_{i=2}^{\log n}((\frac{n}{f_i}-3)f_i+(\frac{n}{f_i}-2)f_i+(\frac{n}{2f_i}-1)f_i). \end{aligned}$$

Since we are using geometry sequence for our offset, f_i is $n/2^i$, we have

$$T(n) = 5n\log n - \frac{15}{2}n + 8.$$

This concludes our proof. □

B. PROOF OF THEOREM 3

We have the time complexity of the Fast selection algorithm. The proof of Theorem 3 is the following:

PROOF. Similar to the proof above, we list the number of compare-swaps each inner loop takes. When offset is f, they are $\lfloor\frac{k}{f}\rfloor f$, $\lceil\frac{n-k}{f}\rceil f$, $\lfloor\frac{n-k-f}{f}\rfloor f$, and f, respectively. The summation of them gives the expression of $T_{n,k}$:

$$T_{n,k} = \sum_{i=0}^{\lfloor\log k\rfloor}(\lfloor\frac{k}{f_i}\rfloor f_i + \lceil\frac{n-k}{f_i}\rceil f_i + \lfloor\frac{n-k-f_i}{f_i}\rfloor f_i + f_i) + n \tag{2}$$

Since f_i is $k/2^i$, we have

$$\begin{aligned} T_{n,k} &\leq \sum_{i=0}^{\lfloor\log k\rfloor}(\frac{k}{f_i}f_i + (\frac{n-k}{f_i}+1)f_i + \frac{n-k-f_i}{f_i}f_i + f_i) + n \\ &= \sum_{i=0}^{\lfloor\log k\rfloor}(2n-k+f_i) + n \\ &\leq (2n-k)\log k + 3n + k - 2 \end{aligned}$$

We can have the lower bound in the same way. □

C. RELATED ALGORITHMS

Figure 15 is the algorithm of Odd-Even Mergesort; while Figure 16 is Bitonic sort algorithm.

```
Odd-Even Mergesort
Input: The n-element array A that to be sorted
Output: The sorted array A
if n > 1 then
    Odd-Even Mergesort A_0, A_1, ..., A_{n/2-1}
    Odd-Even Mergesort A_{n/2}, A_{n/2+1}, ..., A_n
    Odd-Even Merge(A)
end

procedure Odd-Even Merge(a)
1: if n > 2
2:   Odd-Even Merge(a_0, a_2, ..., a_{n-2})
3:   Odd-Even Merge(a_1, a_3, ..., a_{n-1})
4:   CompareSwap(a, i, i+1) for all i ∈ {1, 3, 5, 7, ..., n-3}
5: else
6:   CompareSwap(a, 0, 1)
```

Figure 15: Odd-Even Mergesort Algorithm

```
Bitonic sort
Input: The n-element array A that to be sorted, start, end, direction
Output: The sorted array A_start ... A_end
n ← end − start if n > 1 then
    m ← n/2
    Bitonic sort(A_start, ..., A_m, ASCENDING)
    Bitonic sort(A_{m+start}, ..., A_m, DESCENDING)
    Bitonic Merge(A_start, ..., A_end, direction)
end

procedure Bitonic Merge(a, low, a.length, direction)
1: n ← a.length
2: if n > 1
3: m ← n/2
4: for(i ← low; i < low + m; i ← i + 1
5:   CompareSwap(a, i, i + m, direction)
6: Bitonic Merge(a, low, m, direction)
7: Bitonic Merge(a, low + m, m, direction)
```

Figure 16: Bitonic sort Algorithm