Bucket Oblivious Sort: An Extremely Simple Oblivious Sort

Gilad Asharov * T-H. Hubert Chan† Kartik Nayak‡ Rafael Pass§ Ling Ren¶ Elaine Shi‖

Abstract

We propose a conceptually simple oblivious sort and oblivious random permutation algorithms called bucket oblivious sort and bucket oblivious random permutation. Bucket oblivious sort uses $6n \log n$ time (measured by the number of memory accesses) and $2Z$ client storage with an error probability exponentially small in Z. The above runtime is only $3\times$ slower than a non-oblivious merge sort baseline; for 2^{30} elements, it is $5\times$ faster than bitonic sort, the de facto oblivious sorting algorithm in practical implementations.

1 Introduction

With the increased use of outsourced storage and computation, privacy of the outsourced data has been of paramount importance. A canonical setting is where a client with a small local storage outsources its encrypted data to an untrusted server. In this setting, encryption alone is not sufficient to preserve privacy. The access patterns to the data may reveal sensitive information.

Two fundamental building blocks for oblivious storage and computation [9, 12, 17] are oblivious sorting and oblivious random permutation. In these two problems, an array of n elements is stored on an untrusted server, encrypted under a trusted client's secret key. The client wishes to sort or permute the n elements in a *data-oblivious* fashion. That is, the sequence of accesses it makes to the server should not reveal any information about the n elements (e.g., their relative ranking). The client has a small amount of local storage, the access pattern to which cannot be observed by the server. This work presents simple and efficient algorithms to these two problems, named bucket oblivious sort and bucket oblivious random permutation.

1.1 State of the Affairs.

For oblivious sort, it is well-known that one can leverage sorting networks such as AKS [1] and Zig-zag sort [11] to obliviously sort n elements in $O(n \log n)$ time. Unfortunately, these algorithms are complicated and incur enormous constants rendering them completely impractical. Thus, almost all known practical implementations [17, 13, 14] instead employ the simple bitonic sort algorithm [5]. While asymptotically worse, due to the small leading constants, bitonic sort performs much better in practice.

Oblivious random permutation (ORP) can be realized by assigning a sufficiently long random key to each element, and then obliviously sorting the elements by the keys. To the best of our knowledge, this remains the most practical solution for ORP. It then follows that while $O(n \log n)$ algorithms exist in theory, practical instantiations resort to the $O(n \log^2 n)$ bitonic sort. There exist algorithms such as the Melbourne shuffle [15] that do not rely on oblivious sort; but they require $O(\sqrt{n})$ client storage to permute n elements. Other approaches include the famous Thorp shuffle [7] and random permutation networks [6], but none of these solutions are competitive in performance either asymptotically or concretely.

1.2 Our Results.

Let Z be a statistical security parameter that controls the error probability. Our bucket oblivious sort runs in $6n \log n$ time ($4n \log n$ for bucket ORP) and has an error probability around $e^{-Z/6}$ when the client can store $2Z$ elements locally. This is at most $3\times$ slower than the non-oblivious merge sort, and is at least $5\times$ faster than bitonic sort for $n = 2^{30}$ (cf. Table 1). Therefore, we recommend bucket oblivious sort and bucket ORP as attractive alternatives to bitonic sort in practical implementations.

The core of our algorithms is to assign each element to a random bin and then route the elements through a butterfly network to their assigned random bins. This part is inspired by Bucket ORAM [8]. In more detail,

*Bar-Ilan University. Part of the work was done while the author was a post-doctoral fellow at Cornell Tech supported a Junior Fellow award from the Simons Foundation, and while at J.P. Morgan AI Research.

†The University of Hong Kong. Partially supported the Hong Kong RGC under the grant 17200418.

‡Duke University. Part of the work was done while the author was at University of Maryland.

§Cornell Tech.

¶University of Illinois Urbana-Champaign. Part of the work was done while the author was at MIT.

‖Cornell University.

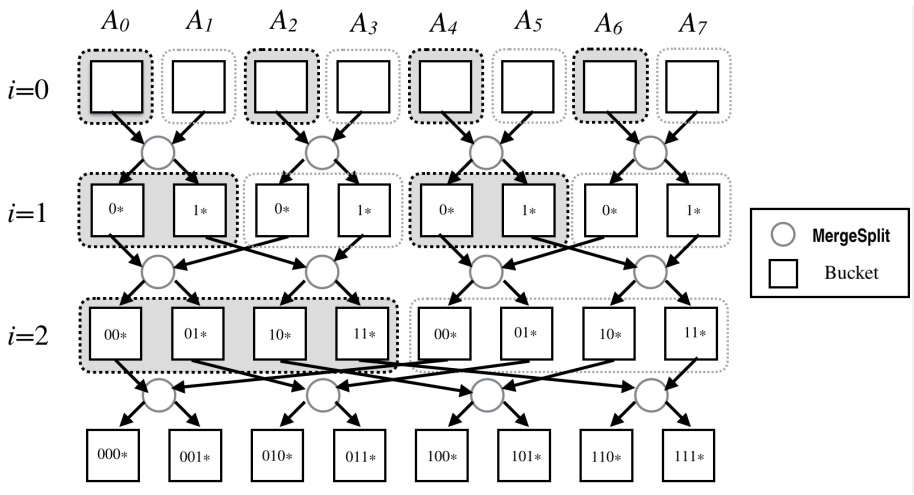

Figure 1: **Oblivious random bin assignment with 8 buckets.** The MergeSplit procedure takes elements from two buckets at level i and put them into two buckets at level $i+1$, according to the $(i+1)$-th most significant bit of the keys. At level i, every 2^i consecutive buckets are semi-sorted by the most significant i bits of the keys.

Algorithm	Oblivious	Client storage	Runtime	Error probability
Merge sort	No	$O(1)$	$2n \log n$	0
Bitonic sort	Yes	$O(1)$	$n \log^2 n$	0
AKS sort [1]	Yes	$O(1)$	$5.4 \times 10^7 \times n \log n$	0
Zig-zag sort [11]	Yes	$O(1)$	$8 \times 10^4 \times n \log n$	0
Randomized Shellsort [10]	Yes	$O(1)$	$24n \log n$	$\approx n^{-3}$
Bucket oblivious sort	Yes	$2Z$	$6n \log n$	$\approx e^{-Z/6}$
Bucket oblivious sort	Yes	$O(1)$	$\approx 2n \log n \log^2 Z$	$\approx e^{-Z/6}$

Table 1: **Runtime of bucket oblivious sort and classic non-oblivious and oblivious sort algorithms.** Bitonic sort requires $\frac{1}{4} n \log^2 n$ comparisons. The number of comparisons for AKS sort and zig-zag sort are cited from [11]. Runtime represents the number of memory accesses, which is four times the number of comparisons.

we divide the n elements into $B = 2n/Z$ buckets of size $Z/2$ each and add $Z/2$ dummy elements to each bucket. Now, imagine that these B buckets form the inputs of a butterfly network — for simplicity, assume B is a power of two. Each element is uniformly randomly assigned to one of the B output buckets, represented by a key of $\log B$ bits. The elements are then routed through the butterfly network to their respective destinations. Assuming the client can store two buckets locally at a time, at level i, the client simply reads elements from two buckets that are distance 2^i away in level i and writes them to two adjacent buckets in level $i+1$, using the i-th bit of each element's key to make the routing decision. We refer readers to Figure 1 for a graphical illustration.

The above algorithm is clearly oblivious, as the order in which the client reads and writes the buckets is fixed and independent of the input array. If no bucket overflows, all elements reach their assigned destinations. By setting Z appropriately, we can bound the overflow probability.

Our bucket oblivious sort and bucket ORP algorithms are derived from the above oblivious random bin assignment building block.

From oblivious random bin assignment to ORP and oblivious sort. To obtain a random permutation, we simply remove all dummy elements and randomly permute each bucket of the final layer. Since the client can hold Z elements, permuting each bucket can be done locally. We show that the algorithm is oblivious and gives a random permutation despite revealing the number of dummy elements in each destination bucket. To get oblivious sort, we can first perform ORP on the input array then apply any *non-oblivious, comparison-based* sorting algorithm (e.g., quick sort or merge sort). We show that the composition of ORP and non-oblivious sort results in an oblivious sort.

Dealing with small client storage. In Section 4.1, we extend our algorithms to support $O(1)$ client storage. We can rely on bitonic sort to realize the MergeSplit operation that operates on 4 buckets at a time, which would result in $O(n \log n \cdot \log^2 Z)$ runtime.

Locality. Algorithmic performance when the data is stored on disk has been studied in the external disk model (e.g., [16, 2, 18, 19]) and references within). Recently, Asharov et al. [3] extended this study to oblivious algorithms. We discuss how our algorithms can be made locality-friendly in Section 4.3.

2 Preliminaries

Notations and conventions. Let $[n]$ denote the set $\{1, \ldots, n\}$. Throughout this paper, we will use n to denote the size of the instance and use λ to denote the security parameter. For an ensemble of distributions $\{D_\lambda\}$ (parametrized with λ), we denote by $x \leftarrow D_\lambda$ a sampling of an instance from the distribution D_λ. We say two ensembles of distributions $\{X_\lambda\}$ and $\{Y_\lambda\}$ are $\epsilon(\lambda)$-statistically-indistinguishable, denoted $\{X_\lambda\} \overset{\epsilon(\lambda)}{\equiv} \{Y_\lambda\}$, if for any unbounded adversary $\mathcal{A}$,

$$\left| \Pr_{x \leftarrow X_\lambda} \left[\mathcal{A}(1^\lambda, x) = 1\right] - \Pr_{y \leftarrow Y_\lambda} \left[\mathcal{A}(1^\lambda, y) = 1\right] \right| \leq \epsilon(\lambda) \ .$$

Random-access machines. A RAM is an interactive Turing machine that consists of a memory and a CPU. The memory is denoted as $\mathsf{mem}[N, b]$, and is indexed by the logical address space $[N] = \{1, 2, \ldots, N\}$. We refer to each memory word also as a *block* and we use b to denote the bit-length of each block. The memory supports read/write instructions $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^b \cup \{\perp\}$. If $\mathsf{op} = \mathsf{read}$, then $\mathsf{data} = \perp$ and the returned value is the content of the block located in logical address addr in the memory. If $\mathsf{op} = \mathsf{write}$, then the memory data in logical address addr is updated to data. We use standard setting that $b = \Theta(\log N)$ (so a word can store an address).

Obliviousness. Intuitively, a RAM program M obliviously simulates a RAM program f if: (1) it has the same input/output behavior as f; (2) There exists a simulator $\mathsf{Sim}(|x|)$ that produces access pattern that is statistically close to the access pattern of $M(x)$, i.e., it can simulate all memory addresses accessed by M during the execution on x, without knowing x. In case the access pattern and the functionality are randomized, we have to consider the joint distribution of the simulator and the output of the RAM program or the functionality.

For a RAM machine M and input x, let $\mathsf{AccPtrn}(M(x))$ denote the distribution of memory addresses a machine M produces on an input x.

Definition 1. *A RAM algorithm M obliviously implements the functionality f with ϵ-obliviousness if the following hold:*

$$\left\{\mathsf{Sim}(1^\lambda), f(x)\right\}_{x \in \{0,1\}^\lambda} \overset{\epsilon(\lambda)}{\equiv} \left\{\mathsf{AccPtrn}(M(x)), M(x)\right\}_{x \in \{0,1\}^\lambda}$$

If $\epsilon(\cdot) = 0$, we say M is perfectly oblivious.

The two main functionalities that we focus on in this paper are the following:

Oblivious sort: This is a deterministic functionality in which the input is an array $A[1, \ldots, n]$ of memory blocks (i.e., each $A[i] \in \{0,1\}^b$, representing a key). The goal is to output an array $A'[1, \ldots, n]$ which is some permutation $\pi : [n] \to [n]$ of the array A, i.e., $A'[i] = A[\pi(i)]$, such that $A'[1] \leq \ldots \leq A'[n]$.

Oblivious permutation: This is a randomized functionality in which the input is an array $A[1, \ldots, n]$ of memory blocks. The functionality chooses a random permutation $\pi : [n] \to [n]$ and outputs an array $A'[1, \ldots, n]$ such that $A'[i] = A[\pi(i)]$ for every i.

3 Our Construction

We first present the oblivious random bin assignment algorithm (Section 3.1) and then use it to implement our bucket oblivious random permutation (Section 3.2) and bucket oblivious sort (Section 3.3).

3.1 Oblivious Random Bin Assignment.

The input to the oblivious random bin assignment algorithm is an array $\mathbf{X}$ of n elements. The goal is to obliviously and uniformly randomly distribute the elements into a set of bins. Each element is assigned to independent random bin, and elements are then routed into the bins obliviously.

The algorithm first chooses a bucket size Z, which can be set to the security parameter λ. Then, it constructs $B = \lceil 2n/Z \rceil$ buckets each of size Z. Without loss of generality, assume B is a power of 2 — if not, pad it to the next power of 2. Note that the algorithm introduces n dummy elements, and the output is twice the size of the input array.

Figure 1 gives a graphic illustration of the algorithm for 8 input buckets and Algorithm 3.1 gives the pseudocode. Each element in $\mathbf{X}$ is assigned a random key in $[0, B-1]$ which represents a destination bucket. Next, the algorithm repeatedly calls the MergeSplit subroutine to exchange elements between bucket pairs in $\log B$

Algorithm 3.1. (Oblivious Random Bin Assignment)

Input: an array $\mathbf{X}$ of size n

Choose a bucket size Z and let B be the smallest power of two that is $\geq 2n/Z$.

Define $(\log B + 1)$ arrays, each containing B buckets of size Z. Denote the j-th bucket of the i-th array $A_j^{(i)}$.

For each element in $\mathbf{X}$, assign a uniformly random key in $[0, B-1]$.

Evenly divide $\mathbf{X}$ into B groups. Put the j-th group into $A_j^{(0)}$ and pad with dummy elements to have size Z.

for $i = 0, \ldots, \log B - 1$ **do**

  for $j = 0, \ldots, B/2 - 1$ **do**

    $(A_{2j}^{(i+1)}, A_{2j+1}^{(i+1)}) \leftarrow \textsc{MergeSplit}(A_{j'+j}^{(i)}, A_{j'+j+2^i}^{(i)}, i)$ where $j' = \lfloor j/2^i \rfloor \cdot 2^{i+1}$

      ▷ Input: j-th pair of buckets with distance 2^i in $A^{(i)}$; Output: j-th pair of buckets in $A^{(i+1)}$

  end for

end for

Output: $A^{(\log B)} = A_0^{(\log B)} \| \ldots A_{B-1}^{(\log B)}$.

function $(A'_0, A'_1) \leftarrow \textsc{MergeSplit}(A_0, A_1, i)$

  A'_0 receives all real elements in $A_0 \cup A_1$ where the $(i+1)$-st MSB of the key is 0

  A'_1 receives all real elements in $A_0 \cup A_1$ where the $(i+1)$-st MSB of the key is 1

  If either A'_0 or A'_1 receives more than Z real elements, the procedure aborts with overflow

  Pad A'_0 and A'_1 to size Z with dummy elements and return (A'_0, A'_1)

end function

levels to distribute elements into their destination buckets. The operation $(A'_0, A'_1) \leftarrow \textsc{MergeSplit}(A_0, A_1, i)$ involves four buckets at the time, distributing the elements in the two input buckets A_0 and A_1 into two output buckets A'_0 and A'_1. A'_0 receives all the keys with $(i+1)$-th most significant bit (MSB) as 0 and A'_1 receives all the keys with $(i+1)$-th MSB as 1.

For now, assume the client can locally store two buckets. For each MergeSplit, it reads (and decrypts) the two input buckets, swaps elements in the two buckets according to the above rule, and writes to the two output buckets (after re-encryption). It is then easy to see that Algorithm 3.1 is oblivious since the order in which the client reads and writes the buckets is fixed and independent of the input array.

When no bucket overflows, all real elements are correctly put into their assigned bins. We now show that the probability of overflow is exponentially small in Z. Intuitively, this is because each bucket contains (in expectation) half dummy elements that serve as a form of "slack" to disallow overflow.

Lemma 3.1. *Overflow happens with at most* $\epsilon(n, Z) = 2n/Z \cdot \log(2n/Z) \cdot e^{-Z/6}$ *probability.*

Proof. Consider a bucket $A_b^{(i)}$ at level i. Observe that this bucket can receive real elements from 2^i initial buckets, each containing $Z/2$ real elements. For each such element, we have chosen an independent and uniformly random key; the element reaches $A_b^{(i)}$ only when the most significant i bits of its key match b, which happens with exactly 2^{-i} probability. A Chernoff bound shows that $A_b^{(i)}$ overflows with less than $e^{-Z/6}$ probability. Hence, a union bound over all levels and all buckets shows that overflow happens with less than $B \cdot \log B \cdot e^{-Z/6} = \epsilon(n, Z)$ probability. ∎

3.2 Bucket Oblivious Random Permutation.

After performing the oblivious random bin assignment, ORP can be simply achieved as follows: scan the array and delete dummy elements from each bin (note that within each bin it is guaranteed that the real elements appear before the dummy elements). Then obliviously permute each bin and finally concatenate all bins. We have:

Lemma 3.2. *Bucket ORP oblivious implement the permutation functionality except for* $\epsilon(n, Z)$ *probability.*

Proof. We first describe the simulator. The access pattern of the oblivious bin assignment algorithm is deterministic and the same for every input, where the overflow even is independent of the input itself. Therefore, it is easy to simulate the bin assignment. The simulator then pretends to simulate the randomly permuting of each bin. Then, the simulator chooses random loads $\vec{k} = (k_0, k_1, \ldots, k_{B-1})$, where k_i is the load of the real elements in the ith bin. This is done by simply throwing n elements into B bins ("in the head"). If there is some i for which $k_i > Z$ then the simulator aborts. The removal of the dummy elements is equivalent to the revealing of these loads.

Clearly, $\vec{k}$ are distributed the same as in the real execution. The only difference between the simulated access pattern and the real one is in the case where the algorithm aborts as a result of an overflow before the last level, which occurs with at most $\epsilon(n, Z)$ probability.

We next show that the output of the algorithm is a random permutation, conditioned on the access pattern. As we previously described, it is actually enough to condition on the vector of random loads $\vec{k} = (k_0, k_1, \ldots, k_{B-1})$. We show that given any such vector, all permutations are equally likely.

Fix a particular load $\vec{k} = (k_0, k_1, \ldots, k_{B-1})$. The algorithm works by first assigning the real elements into the bins, and then permuting within each bin. For every input, there are exactly $\binom{n}{k_0,\ldots,k_{B-1}}$ ways to distribute the real elements into the bins while achieving the vector of loads $\vec{k}$. Then, each bin is individually permuted, i.e., within each bin i, we have k_i different possible ordering. Overall, the total number of possible outputs with that load is then

$$\binom{n}{k_0, \ldots, k_{B-1}} \cdot k_0! \cdot \ldots \cdot k_{B-1}! = n!$$

That is, even conditioned on some specific loads $\vec{k} = (k_0, k_1, \ldots, k_{B-1})$, all permutations are still equally likely. Therefore, $\forall \pi$, $\Pr\left[\Pi = \pi \mid \vec{K} = \vec{k}\right] = \frac{1}{n!}$, and

$$\Pr[\Pi = \pi] = \sum_{\vec{k}} \Pr\left[\Pi = \pi \mid \vec{K} = \vec{k}\right] \cdot \Pr\left[\vec{K} = \vec{k}\right] = \frac{1}{n!}$$

Our algorithm fails to implement the ORP only when some bin overflows during the oblivious random bin assignment, which happens with $\epsilon(n, Z)$ probability by Lemma 3.1. □

3.3 Bucket Oblivious Sort. Once we have ORP, it is easy to achieve oblivious sort: just invoke any non-oblivious comparison-based sort after ORP.

Since the functionality is deterministic, it is enough to consider separately correctness and simulation. Correctness follows from directly from the correctness of the ORP and the non-oblivious sort. As for obliviousness, given any input array, one can easily simulate the algorithm by first randomly permuting the array and then running the comparison-based non-oblivious sort. The access patterns of a comparison-based sort depend only on the relative ranking of the input elements, which is independent of the input array once the array has been randomly permuted.

3.4 Efficiency. We analyze the efficiency of our algorithms and compare them to classic non-oblivious oblivious sorting algorithms in Table 1. We measure runtime using the number of memory accesses the clients needs to perform on the server.

For our algorithms, assuming the client can store $2Z$ elements locally, each $2n$-sized array is read and written once and there are $\log(2n/Z) < \log n$ of them. So oblivious bin assignment and bucket ORP run in (less than) $4n \log n$ time. Note that the last step of ORP, i.e., permuting each output bucket, can be incorporated with the last level of oblivious bin assignment. Bucket oblivious sort additionally invokes a non-oblivious sort, and thus runs in $6n \log n$ time. This is within $3\times$ of merge sort and beats bitonic sort when n is moderately large; for example, $5\times$ faster than bitonic for $n = 2^{30}$. For an overflow probability of 2^{-80} and most reasonable values of n, $Z = 512$ suffices.

4 Extensions

4.1 Extension to Constant Client Storage. We now discuss how to extend our algorithms to the case where the client can only store $O(1)$ elements locally.

Each MergeSplit can be realized with a single invocation of bitonic sort. Concretely, we first scan the two input buckets to count how many real elements should go to buckets A'_0 vs. A'_1, then tag the correct number of dummy elements going to either buckets, and finally perform a bitonic sort.

Next, we need to permute each output bucket obliviously with $O(1)$ local storage. This can be done as follows. First, assign each element in a bucket a uniformly random label of $\Theta(\log n)$ bits. Then, obliviously sort the elements by their random labels using bitonic sort. Since the labels are "short" (i.e., logarithmic in size), we may have collisions with n^{-c} probability for some constant c, in which case we simply retry. In expectation, it succeeds in $1 + o(1)$ trials.

Since we invoke $B/2$ instances of bitonic sort on $2Z$ elements at each level, the runtime is roughly $\log B \cdot B/2 \cdot 2Z \log^2(2Z)) \approx 2n \log n \log^2 Z$.

4.2 Better Asymptotic Performance. Our algorithms can also be extended to have better asymptotic performance. For this instantiation, we use a primitive called oblivious tight compaction. Oblivious tight compaction receives n elements each marked as either 0 or 1, and outputs a permutation of the n elements such that all elements marked 0 appear before the elements that are marked 1. It should not be hard to see that oblivious tight compaction can be used to achieve MergeSplit. Using the $O(1)$-client-storage and $O(n)$-time oblivious tight compaction construction from [4], bucket oblivious sort achieves $O(n \log n + n \log^2 Z)$ runtime and $O(1)$ client storage. Setting $Z = \omega(1) \log n$,

bucket oblivious sort achieves $O(n \log n)$ runtime, $O(1)$ client storage, and a negligible in n error probability.

4.3 Locality. Algorithmic performance when the data is stored on disk has been studied in the external disk model (e.g., [16, 2, 18, 19]) and references within). Recently, Asharov et al. [3] extended this study to oblivious algorithms. In this setting, an algorithm is said to have (p, ℓ) locality if it has access to p disks and accesses in total ℓ discontiguous memory regions in all disks combined. As an example, it is not hard to see that merge sort is a non-oblivious sorting algorithm that sorts an array of size n in $O(n \log n)$ and $(3, \log n)$-locality, whereas quick sort is not local for any reasonable p. This locality metric is motivated by the fact that real-world storage media such as disks support sequential accesses much faster than random seeks. Thus an algorithm that makes mostly sequential accesses would execute much faster in practice than one that makes mostly random accesses — even if the two have the same runtime in a standard word-RAM model.

Guided by this new metric, Asharov et al. [3] consider how to design oblivious algorithms and ORAM schemes that achieve good locality. Since sorting is one of the most important building blocks in the design of oblivious algorithms, inevitably Asharov et al. [3] show a locality-friendly sorting algorithm. Concretely, they show that there is a specific way to implement the bitonic sort meta-algorithm, such that the entire algorithm requires accessing $O(\log^2 n)$ distinct memory regions (i.e., as many as the depth of the sorting network) require only 2 disks to be available — in other words, the algorithm achieves $(2, O(\log^2 n))$-locality.

We observe that our algorithm, when implemented properly, is a locality-friendly oblivious sorting algorithm. Our algorithm outperforms Asharov et al. [3]'s scheme by an almost logarithmic factor improvement in locality. To achieve this, the crux is to implement all n/Z instances of MergeSplit in the same layer of the butterfly network while accessing a small number of discontiguous regions. Specifically, the MergeSplit operation works on 4 buckets at a time, while reading two buckets from the input layer, and writing to two consecutive buckets in the output layer. Moreover, the different invocations of MergeSplit on the same layer deal with consecutive buckets. By carefully distributing the buckets among the different disks, and by using bitonic sort while implementing the MergeSplit operation, we conclude:

Corollary 4.1. *There exists a statistically oblivious sort algorithm which, except with $\approx e^{-Z/6}$ probability, completes in $O(n \log n \log^2 Z)$ work and with $(3, O(\log n \log^2 Z))$ locality.*

References

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $0(n \log n)$ sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9. ACM, 1983.

[2] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory (extended abstract). In *ACM Symposium on the Theory of Computing (STOC '97)*, pages 540–548, 1997.

[3] Gilad Asharov, T-H Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Locality-preserving oblivious RAM. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 214–243. Springer, 2019.

[4] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: optimal oblivious RAM. *Cryptology ePrint Archive*, 2018.

[5] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.

[6] Artur Czumaj. Random permutations using switching networks. In *STOC*, pages 703–712. ACM, 2015.

[7] Artur Czumaj and Berthold Vöcking. Thorp shuffling, butterflies, and non-markovian couplings. In *ICALP (1)*, volume 8572 of *Lecture Notes in Computer Science*, pages 344–355. Springer, 2014.

[8] Christopher W Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious RAM. *Cryptology ePrint Archive*, 2015.

[9] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[10] Michael T Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1262–1277. SIAM, 2010.

[11] Michael T Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n$ time. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 684–693. ACM, 2014.

[12] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.

[13] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *Symposium on Security and Privacy*. IEEE, 2015.

[14] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC:

Parallel secure computation made easy. In *Symposium on Security and Privacy*. IEEE, 2015.

[15] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming*, pages 556–567. Springer, 2014.

[16] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[17] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Symposium on Security and Privacy*. IEEE, 2013.

[18] Jeffrey Scott Vitter. External memory algorithms and data structures. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[19] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.