Flexway O-Sort: Enclave-Friendly and Optimal Oblivious Sorting

Tianyao Gu[1], Yilei Wang[2], Afonso Tinoco[1], Bingnan Chen[3], Ke Yi[3], and Elaine Shi[1]

[1]Carnegie Mellon University, Oblivious Labs Inc.
{tianyaog, atinoco}@andrew.cmu.edu, runting@gmail.com
[2]Alibaba Cloud
fengmi.wyl@alibaba-inc.com
[3]Hong Kong University of Science and Technology
{bchenba, yike}@cse.ust.hk

Abstract

Oblivious algorithms are being deployed at large scale in real world to enable privacy-preserving applications such as Signal's private contact discovery. Oblivious sorting is a fundamental building block in the design of oblivious algorithms for numerous computation tasks. Unfortunately, there is still a theory-practice gap for oblivious sort. The commonly implemented bitonic sorting algorithm is not asymptotically optimal, whereas known asymptotically optimal algorithms suffer from large constants.

In this paper, we construct a new oblivious sorting algorithm called flexway o-sort, which is asymptotically optimal, concretely efficient, and suitable for implementation in hardware enclaves such as Intel SGX. For moderately large inputs of 12 GB, our flexway o-sort algorithm outperforms known oblivious sorting algorithms by 1.32× to 28.8× when the data fits within the hardware enclave, and by 4.1× to 208× when the data does not fit within the hardware enclave. We also implemented various applications of oblivious sorting, including histogram, database join, and initialization of an ORAM data structure. For these applications and data sets from 8GB to 32GB, we achieve 1.44 $\sim$ 2.3× speedup over bitonic sort when the data fits within the enclave, and 4.9 $\sim$ 5.5× speedup when the data does not fit within the enclave.

1 Introduction

Oblivious algorithms [24, 23, 49, 53, 36, 32, 4, 43, 7] allow a trusted client (e.g., a secure hardware enclave) to securely outsource data to an untrusted storage (e.g., memory or disk), such that as the client accesses the data, the addresses visited leak nothing about the secret data or query. Oblivious algorithms have been deployed at a large scale in the real world. For example, Signal, the encrypted messenger app, relies on an oblivious key-value store to enable private contact discovery [48]. Specifically, users send their encrypted contacts to the Signal server running in a hardware enclave (also called TEE, short for Trusted Execution Environment), and the enclave securely fetches matching entries from an oblivious key-value store, such that the access patterns do not leak which users' entries are matched.

Oblivious sorting [2, 27, 4, 35, 43] is known to be a particularly important building block in the design of oblivious algorithms. Prior works [36, 7, 28, 43, 38, 18, 51, 13] showed that we can obliviously realize numerous computation tasks using oblivious sorting, such as histogram, page rank, database joins and group-by queries, list ranking, tree computations with Euler tour, tree contraction, graph algorithms such as breadth-first search, connected components, and minimum spanning tree/forest. More generally, any computational task represented as a streaming Map-Reduce algorithm or in the GraphLab programming model has an efficient oblivious implementation using oblivious sorting [28, 36, 38]. Oblivious sorting is also employed in Oblivious RAM (ORAM) [23] (a generic compiler that compiles any program to an oblivious counterpart), either for initialization, or as a load balancer for parallelization [11, 16].

Although oblivious sorting has been explored in various prior works, the practical state of the art is still unsatisfactory. Bitonic sort [6], due to its simplicity, is a go-to scheme in many implementations [55, 18, 37, 3]; however, its asymptotical work $O(N \log^2 N)$ is suboptimal, where N denotes the input size. On the other hand, the well-known AKS sorting network [2] and subsequent improvements [40, 27] achieve an optimal $O(N \log N)$ work, but their construction suffer from astronomically large constants. To bridge this significant theory-practice gap, some recent works [26, 4, 43, 47] proposed oblivious sort constructions that achieve either $O(N \log N)$ or $O(N \log N \mathsf{poly} \log \log N)$ work; however, their practical performance are still far worse than bitonic sort.

In this paper, we explore the following important question:

Can we acheive *concretely efficient*, *enclave-friendly* oblivious sorting with *optimal* $O(N \log N)$ work?

1.1 Our Results and Contributions

Our work takes a step forward in designing practically efficient and asymptotically optimal oblivious sorting algorithms. Moreover, we show that our oblivious sorting algorithm is enclave-friendly from both an algorithmic perspective and with empirical implementation and evaluation. We summarize our contributions below.

Concretely efficient and optimal oblivious sorting. We propose a new oblivious sorting algorithm called *flexway oblivious sort* (or *flexway o-sort* for short) which achieves $(2.23 + o(1))N \log N$ work (measured in terms of the number of compare-and-exchange operations).

Enclave-friendliness. Our flexway o-sort algorithm is optimal in both of the following scenarios:

1. [**EPC $\geq$ data**]: when the input array fits within the enclave's protected memory region (also called EPC, short for Enclave Page Cache);
2. [**EPC $<$ data**]: when the input array does not fit within the EPC memory.

In the former setting, we want to optimize the *computational overhead* (also called *work*) of the algorithm. In the latter setting, we also need to minimize the number of *page swaps*. Specifically, when the enclave fetches (encrypted) data from insecure memory or disk, it needs the operating system's help to perform a page swap. The page swap is a heavy-weight operation since it involves context switching as well as encrypting and decrypting the pages being swapped in and out. Existing benchmarking results [51] showed that the cost of a 4 KB page swap can be 66$\times$ more expensive than moving a 4 KB page within the EPC memory. The vast majority of oblivious algorithms [24, 23, 4, 31, 36, 49, 53] in the literature focus on optimizing work but not the page swap overhead. Earlier works [51] showed that if we directly employ such an algorithm for hardware enclaves, then page swaps will likely become the dominant overhead when the data does not fit within the EPC memory. Also, it is known that $\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$ number of page swaps is necessary for any (even non-oblivious) sorting algorithm [1, 19].

Although recent enclave technologies such as SGX v2 and TDX has supported larger secure memory [17, 12], external-memory efficiency remains crucial for deploying oblivious sorting for several reasons:

- Limited RAM Space: Although Intel's server-grade processors now support up to 1TB of EPC, many privacy-preserving systems are also limited by the amount of physical memory. For a private blockchain, e.g., high RAM requirements would significantly hinder participation.
- Resource Allocation: Sorting are often performed as maintenance tasks. In Signal's usecase, e.g., if oblivious sorting is applied to scale the database, it should not consume memory needed by the latency-sensitive query service.
- Enclave Creation Time: The creation time of an SGX enclave increases linearly with the EPC size; e.g., creating a 64GB EPC enclave takes about 150 seconds in SGX V2 [17]. Therefore, it is desirable to minimize the EPC size for faster enclave creation.
- Communication in Distributed Clusters: when sorting is performed on a distributed clusters, page swap overheads translate into communication costs between nodes.

To the best of our knowledge, our flexway o-sort is the **first concretely efficient algorithm that achieves optimality in both dimensions**. Specifically, our flexway o-sort algorithm achieves $(2.23 + o(1))N \log N$ work and $(3 + o(1))\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$ number of page swaps where B denotes the page size, and M denotes the size of the EPC memory.

Open-source implementation. We implemented our algorithm and evaluated their concrete performance. The core algorithm implementation (counting both oblivious sorting algorithms) has 1,600 lines of code. Although our implementation uses Intel SGX, the algorithm design should work for any common hardware enclave architecture. Our implementation has been made open source at `https://github.com/odslib/oblsort`.

Evaluation. We show that our algorithm achieves significant speedup over prior oblivious sorting algorithms for both the [EPC $\geq$ data] and [EPC $<$ data] scenarios. For an array of 12 GB size, our speedup over prior algorithms is depicted in the following table where for the [EPC $<$ data] scenario, we adopt an EPC size of 128 MB which is the same as SGXv1:

Scheme	**Our speedup**	
	EPC $\geq$ data	EPC $<$ data
Randomized Shellsort	28.8×	208×
Bitonic (non-recursive impl.)	5.49×	38.4×
Bitonic (recursive impl.)	1.32×	4.10×
Multi-way bucket o-sort[43]	14.3×	12.4×

We also implemented various applications that rely on oblivious sorting, including histogram, ORAM initialization, and database join. We measure the end-to-end application performance when using our flexway o-sort, and compare it with (recursive) bitonic sort as a baseline. For these applications, we achieve $1.44 \sim 2.3\times$ speedup when the data fits within the enclave, and $4.9 \sim 5.5\times$ speedup when the data does not fit within the enclave.

Distribution o-sort. We also provide the distribution o-sort algorithm in Section B, which is a variant of our flexway o-sort algorithm that achieves optimal constant for the number of page swaps. Table 1 shows the performance of both our flexway o-sort and distribution o-sort algorithms in comparison with prior work.

Additional result. As a byproduct, we also construct an *oblivious shuffler*, which randomly permutes an input array without leaking the permutation. Earlier works showed that the oblivious shuffler is also a versatile primitive in oblivious algorithms and ORAM schemes [45, 46, 30]. We give more detailed evaluation results on oblivious shuffler in Section 5.4.

1.2 Technical Highlights

Starting point: multi-way bucket o-sort. Ramachandran and Shi [43] constructed multi-way bucket o-sort, which is asymptotically optimal in both work and page swaps, but unfortunately suffers from astronomical constants. Their construction reduces the task of oblivious sorting N elements to an oblivious p-way $\mathsf{MergeSplit}$, which can be viewed as a special sorting algorithm where the keys come from a small domain $\{0, 1, \ldots, p-1\}$. More precisely, a p-way $\mathsf{MergeSplit}$ accomplishes the following:

- **Input:** p bins each of size Z. Each bin contains *real elements* each marked with a key from $\{0, 1, \ldots, p-1\}$ and *fillers*. The frequency of each distinct key in the input is bounded by Z.
- **Output:** Route the real elements to p destination bins depending on their key, and each output bin is padded with fillers to its capacity Z.

Ramachandran and Shi [43] showed that if we can achieve oblivious $\mathsf{MergeSplit}$ in $O(n \log n)$ cost where $n = pZ$, $p = \log N$, and $Z \in \mathsf{poly} \log N$, then we can get oblivious sorting optimal in both work and number of page swaps.

Table 1: **Comparison with sorting algorithms from prior works.** For the multi-way bucket o-sort [43], C_{AKS} and C_{SPMS} denote the large constants associated with the AKS sorting network [2] and the SPMS sorting algorithm [14]. For all the algorithms listed in the table, the work is dominated by the element-wise exchanges. A page read plus a page write is counted as one page swap.

Algorithm	**Work (# Exchanges)**	**Page Swaps**
Theoretically optimal [1, 19]	$\Omega(N \log N)$	$\Omega(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$
	Prior works	
Bitonic (non-recursive) [6]	$(\frac{1}{4} + o(1))N \log^2 N$	$(\frac{1}{2} + o(1))\frac{N}{B} \log^2 N$
Bitonic (recursive) [6, 18]	$(\frac{1}{4} + o(1))N \log^2 N$	$(\frac{1}{2} + o(1))\frac{N}{B} \log^2 \frac{N}{M}$
Randomized Shell Sort† [26]	$24N \log N$	$(24 + o(1))N \log \frac{N}{M}$
Bucket o-sort [4]	$(1 + o(1))N \log N \log^2 \log N$	$(4 + o(1))\frac{N}{B} \log \frac{N}{B}$
Multi-way bucket o-sort [43]	$C_{\text{AKS}} \cdot N \log N$	$C_{\text{SPMS}} \cdot \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$
	Our results	
Standard tall cache assumption $B \geq \log^2 N$ and $M \geq B^2$:		
Flexway o-sort	$(2.23 + o(1))N \log N$	$(3 + o(1))\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{N}{B}$
Strong tall cache assumption $B = \log^c N$ and $M \in B^{\omega(1)}$:		
Flex-way o-sort	$(\max(c, 1) + 1.23 + o(1))N \log N$	$(2 + o(1))\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{N}{B}$
Distribution o-sort	$\frac{1}{2}(c + 1 + o(1))N \frac{\log N \log \log N}{\log \log \log N}$	$(1 + o(1))\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + 1.5\frac{N}{B}$

†With non-negligible failure probability.

Our key idea. We observe that Ramachandran and Shi's framework still works if we use a slightly smaller $p \in \Theta(\sqrt{\log N})$. Our main contribution is to construct a concretely efficient $\Theta(\sqrt{\log N})$-way MergeSplit algorithm that achieves $O(m \log m)$ work. To get this, we need some new algorithmic tricks. Notably, we show that if one can construct an oblivious Euler tour algorithm for a graph with p vertices, we can achieve p-way MergeSplit efficiently. To solve the oblivious Euler tour problem efficiently, we applied a *packing* trick. Specifically, as long as $p \in O(\sqrt{\log N})$, we can pack the adjacency matrix of the graph in $O(1)$ memory words. This allows us to obliviously access any entry in the adjacency matrix in $O(1)$ cost under a standard word-RAM model [21], and hence obtain an Euler tour obliviously in linear time. We refer the readers to Section 3.4 for the details of our novel MergeSplit algorithm.

Other constant-factor optimizations. Simply plugging in our new MergeSplit into the multi-way bucket o-sort framework allows us to significantly improve the concrete performance relative to the original multi-way bucket o-sort [43]; however, it is not enough for achieving the tight constants claimed earlier. To get the promised result with $(2.23 + o(1))N \log N$ work and $(3 + o(1))\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$ page swaps, we introduce various additional optimizations.

Notably, while previous works use a uniform number of ways throughout the butterfly, we use a flexway butterfly construction that allows different number of ways in different layers (hence the name *flexway o-sort*). This approach ensures that the algorithm can adapt to all input sizes with only $(1 + o(1))\times$ worst-case overhead. While the original butterfly network construction of Ramachandran and Shi [43] requires out-of-place data movements, we adopt an in-place variant of the butterfly network instead. Furthermore, we avoid the matrix transposition operations needed in the original multi-way bucket o-sort [43] through a careful choice of parameters. We describe various additional optimizations in subsequent technical sections as well as the appendices.

2 Preliminaries

2.1 Threat Model

We assume that the server uses secure hardware enclaves, such as Intel SGX, to ensure computational integrity. While we use Intel SGX as a test platform, our algorithmic constructions are compatible with various hardware enclave technologies. In this paper, we assume the hardware enclave itself is secure, as explored in complementary research on designing provably secure trusted hardware [50, 54, 20].

Due to the limited capacity of an enclave's secure memory, a page swap mechanism is required to manage encrypted pages between external storage and internal memory. Even with sufficient secure memory, a malicious operating system could still revoke access rights to an enclave page, causing a page fault [9]. During page swaps, the OS could observe page-level access patterns and tamper with the pages. Additionally, the OS might monitor fine-grained memory accesses within the enclave through cache-timing attacks [44, 8]. Therefore, our algorithm must be fully oblivious [45] (also referred to as doubly oblivious [37]), ensuring that no secret information is leaked through page-level or EPC memory access patterns.

2.2 Background on Bucket Oblivious Sort

Our flexway o-sort algorithm builds on the bucket o-sort framework proposed by Asharov et al. [4], and a multi-way variant introduced by Ramachandran and Shi [43].

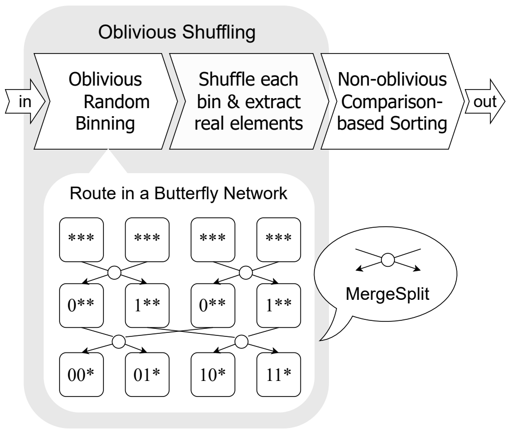

Oblivious Shuffling
in
Oblivious Random Binning
Shuffle each bin & extract real elements
Non-oblivious Comparison-based Sorting
out
Route in a Butterfly Network
***
***
***
***
0**
1**
0**
1**
00*
01*
10*
11*
MergeSplit

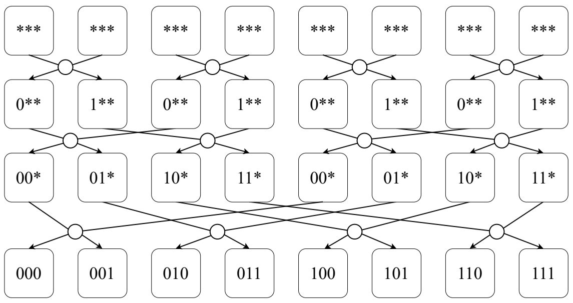
Figure 1: Procedure of Bucket Oblivious Sort.

The "oblivious shuffle + non-oblivious sort" framework. We depict the bucket oblivious sort algorithm [4] in Figure 1. The algorithm first *oblivious shuffle* (or *o-shuffle* for short) the input array, i.e., randomly permuting the array without leaking the permutation, and then employ a comparison-based, non-oblivious sorting algorithm on the shuffled array. Asharov et al. [4] proved that this paradigm is oblivious.

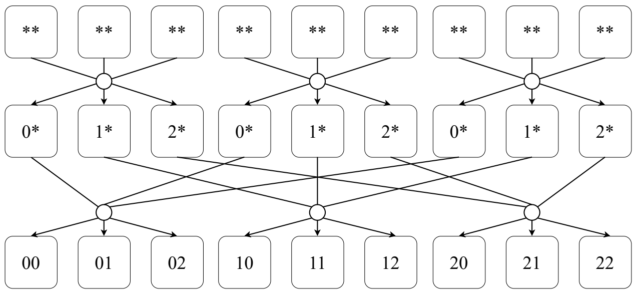
Oblivious random binning: core primitive in o-shuffle. A core primitive for realizing an o-shuffle is called an *oblivious random binning*, which addresses the following problem: given an input array of size N, we want to obliviously place each element randomly into one of $(1+\epsilon)N/Z$ bins, where $Z = \mathsf{poly}\log N$ is the size of each bin and $\epsilon > 0$ is a constant called the slack factor. For simplicity, prior works [4, 43] set $\epsilon = 1$. Asharov et al. [4] showed that we can realize an o-shuffle in the following way.

<u>O-shuffle</u>:

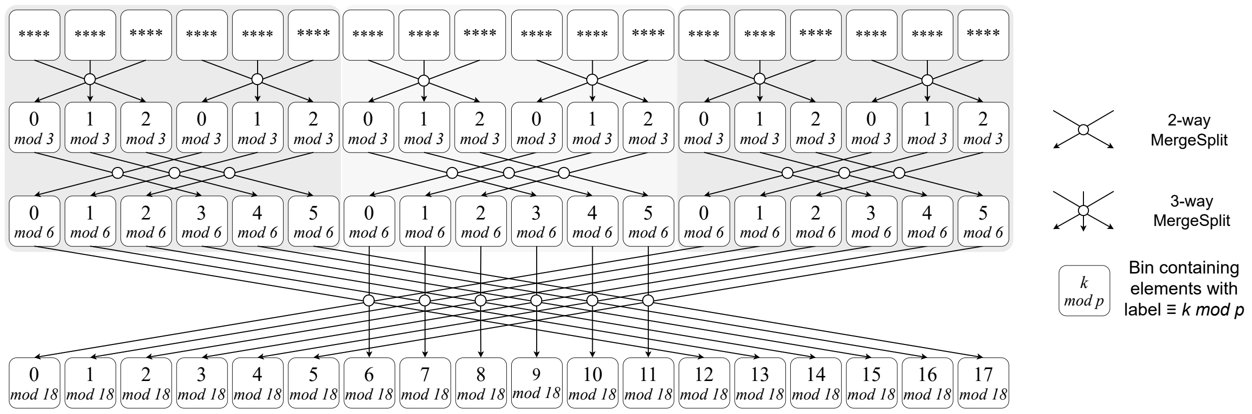
1. Run random binning on the input array.
2. Obliviously shuffle the elements within each bin.
3. Extract all real elements and drop the fillers.



(a) 2-way butterfly network for NBin = 8. Uses binary labels.



(b) 3-way butterfly network for NBin = 9. Uses ternary labels.



(c) A $3 \times 2 \times 3$ flexway butterfly network for NBin = 18. The MergeSplit function routes elements based on the modulo of their labels.

Figure 2: Comparison of (a) the two-way butterfly network from [4], (b) the multi-way variant from [43], and (c) the flex-way variant proposed in this work. The flex-way butterfly network introduces fewer restrictions on the number of bins and is restructured to enable in-place MergeSplit (see section 4.2 for details).

Note that Step 3 leaks how many real elements fall within each bin. Asharov et al. [4] showed that this leakage is safe as the number of real elements in each bin can be simulated.

Realizing oblivious random binning. We now describe the oblivious random binning algorithm by Asharov et al. [4]. Their construction uses a butterfly network with $(\log \mathsf{NBin}) + 1$ layers, where each layer has $\mathsf{NBin} = (1 + \epsilon)N/Z$ bins of capacity Z. The algorithm works as follows:

Oblivious random binning:

- Initially, a random label of $\log \mathsf{NBin}$ bits is assigned to each input element. The label denotes which final-layer bin the element goes to.
- Place exactly $Z/(1 + \epsilon)$ input elements in each bin at layer 0, and pad the bins to a full capacity with fillers.
- All elements are routed to their destination bins in the final layer along the butterfly network, through a sequence of MergeSplit operations as defined below. When routing from layer ℓ to layer $\ell + 1$, each element uses the ℓ-th bit of its label to decide the direction to go for the MergeSplit.

In expectation, the number of real elements that land in each bin is exactly $Z/(1 + \epsilon)$. Earlier work [4] proved that if $Z \in \mathsf{poly}\log N$, the probability that any bin overflows is negligible, so a simulator assuming no overflow produces access patterns statistically indistinguishable from real-world execution.

Core subroutine MergeSplit. In the butterfly network, a pair of bins at level ℓ route to a pair of bins at level $\ell + 1$ — this is accomplished through a MergeSplit operation.

The MergeSplit operation takes two input arrays (i.e., bins) each of size Z, where each array contains real elements tagged with a key 0 or 1, as well as some fillers. MergeSplit then routes each real element to either the left or the right output bin based on its key, and each output bin is padded with fillers to its capacity Z. For obliviousness, the algorithm's access patterns should not leak where each element is going.

A multi-way variant. Ramachandran and Shi [43] introduced a p-way variant of the bucket o-sort algorithm, reducing the depth of the butterfly network from $O(\log(N/Z))$ to $O(\log_p(N/Z))$. Each bin is connected to p bins in the next layer, and the MergeSplit operation is extended to handle p input and p output bins. Specifically, the MergeSplit routes each element at level ℓ to one of the p output bins at level $\ell + 1$ based on the ℓ-th digit of its label in base-p. By setting $p = \log N$, Ramachandran and Shi [43] gave a theoretical construction of oblivious sorting with optimal work. Figure 2b shows an example of p-way butterfly network where $p = 3$.

Key challenge: an efficient oblivious MergeSplit? The reason why the bucket o-sort [4] and the subsequent multiway variant [43] are inefficient is because they lack an efficient MergeSplit algorithm. Specifically, bucket o-sort uses bitonic sort to realize the MergeSplit. As bitonic sort requires $O(n \log^2 n)$ work on an input size n, the resulting oblivious sorting suffer from $O(N \log N (\log \log N)^2)$ work. While we can replace bitonic sort with a more efficient compaction algorithm with $O(n \log n)$ work [25, 45], the resulting oblivious sorting still requires $O(N \log N \log \log N)$ work.

By contrast, the multiway bucket o-sort algorithm [43] uses a $\log N$-way butterfly network, and applies AKS sorting [2] to realize the $\log N$-way MergeSplit with $O(n \log n)$ work. This approach achieves asymptotic optimality for oblivious sorting (i.e. $O(N \log N)$ work), but suffers from astronomical constants due to the expander graphs in AKS.

It is also possible to use linear-work oblivious compaction [41] to realize the MergeSplit in the original (2-way) bucket o-sort [4]. This also achieves optimality in work but again known linear-work oblivious compaction [41] relies on expander graphs and suffers from enormous constants.

3 A New Multi-Way MergeSplit

3.1 Overview

Overview of our flexway o-sort. Our new flexway o-sort algorithm follows the same overall framework as multi-way bucket o-sort [4, 43]. However, unlike earlier works [43] that use either a 2-way or $\log N$-way butterfly network, we set the number of ways to be $p \in \Theta(\sqrt{\log N})$. Both choices $p = \log N$ and $p \in \Theta(\sqrt{\log N})$ results in a butterfly network of $\Theta(\log_p N) = \Theta(\log N / \log \log N)$ layers.

The reason why we choose $p \in \Theta(\sqrt{\log N})$ (as opposed to $\Theta(\log N)$ like in earlier work [43]) is because it allows us to pack the adjacency matrix of a graph with p vertices into $O(1)$ memory words. This *packing trick* is later used to implement a more efficient p-way MergeSplit algorithm.

Overview of our p-way MergeSplit. As mentioned, Ramachandran and Shi's multi-way bucket o-sort [43] is inefficient because they lacked an efficient multi-way MergeSplit. One of our main contributions is to devise a practical oblivious p-way MergeSplit algorithm that achieves $O(n \log n)$ work with input size $n = pZ$, where $p \in \Theta(\sqrt{\log N})$ and $Z \in 2^{O(\sqrt{\log N})}$. As shown in Figure 3, our novel p-way MergeSplit algorithm works as follows. Suppose we have an input array consisting of real elements and fillers, and each real element is marked with a key from $\{0, 1, \ldots, p-1\}$, denoting which way it wants to go. Now, perform the following steps:

1. *Preprocess*: We first preprocess the input array and tag each filler with a key from $\{0, 1, \ldots, p-1\}$, such that at the end, there is an equal number of elements with each key $k \in \{0, 1, \ldots, p-1\}$.
2. *Interleave*: We run a recursive algorithm called Interleave to rearrange the input array such that in the output array, the elements have interleaving keys $0, 1, 2, \ldots, p-1, 0, 1, 2, \ldots, p-1, \ldots$, and so on.

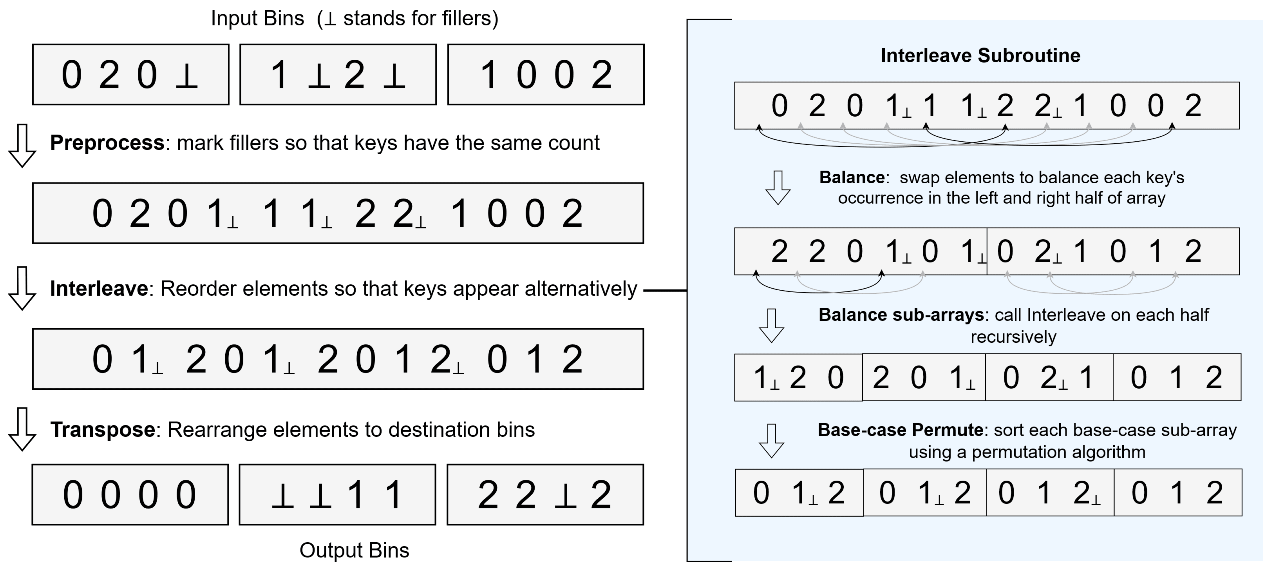

Input Bins (⊥ stands for fillers)
0 2 0 ⊥
1 ⊥ 2 ⊥
1 0 0 2
Preprocess: mark fillers so that keys have the same count
0 2 0 1⊥ 1 1⊥ 2 2⊥ 1 0 0 2
Interleave: Reorder elements so that keys appear alternatively
0 1⊥ 2 0 1⊥ 2 0 1 2⊥ 0 1 2
Transpose: Rearrange elements to destination bins
0 0 0 0
⊥ ⊥ 1 1
2 2 ⊥ 2
Output Bins
Interleave Subroutine
0 2 0 1⊥ 1 1⊥ 2 2⊥ 1 0 0 2
Balance: swap elements to balance each key's occurrence in the left and right half of array
2 2 0 1⊥ 0 1⊥ 0 2⊥ 1 0 1 2
Balance sub-arrays: call Interleave on each half recursively
1⊥ 2 0
2 0 1⊥
0 2⊥ 1
0 1 2
Base-case Permute: sort each base-case sub-array using a permutation algorithm
0 1⊥ 2
0 1⊥ 2
0 1 2⊥
0 1 2

Figure 3: **Left:** An illustration of our MergeSplit construction. Input elements are marked with keys 0, 1, 2, and we use $\perp$ to denote filler elements. **Right:** The Interleave subroutine, which in turn relies on the Balance subroutine as a building block.

3. *Transpose*: Finally, rearrange elements into output bins by transposing the result of the previous step , so that the keys have the form $\underbrace{0, 0, \ldots, 0}_{Z}, \underbrace{1, 1, \ldots, 1}_{Z}, \ldots, \underbrace{p, \ldots, p}_{Z}$.

In the remainder of the section, we describe how each step is performed. Specifically, the core of our new multi-way MergeSplit is the Interleave subroutine, a recursive algorithm that calls another building block called Balance. The Balance subroutine is also the technical highlight of our algorithm, since it expresses the problem of rearranging an input array in a specific way as an Euler tour problem for a small graph with $p \in O(\sqrt{\log N})$ vertices. By leveraging the fact that the adjacency matrix of this graph can be packed in $O(1)$ memory words, we can make the algorithm highly efficient.

3.2 Balance

As mentioned, our p-way MergeSplit algorithm relies on a central building block called Balance.

Syntax. The Balance algorithm takes as input an array A containing n elements, each marked with a key from $\{0, 1, \ldots, p-1\}$. It is promised that each distinct key appears an even number of times. The goal of Balance is to rearrange the array A such that

- each key appears the same number of times in the left and right half of the array; and
- the last element of A is the same as the input array.

The requirement of preserving the last element of the array is used in later building blocks when the array length is not a power of 2 — see the Permute algorithm in Section A.

Constraints. $p \in O(\sqrt{\log N})$.

Intuition: translate Balance into an Euler-tour on the exchange graph. First, we pair up elements in the left and right halves of the array A, specifically $A[i]$ and $A[n/2 + i]$ for each $i \in \{0, \ldots, n/2 - 1\}$. Our goal is to exchange some of these pairs to make the array balanced.

We can equivalently think of this as a graph problem. We create a multi-graph G (called the exchange graph) with p vertices, one for each key in $\{0, 1, \ldots, p-1\}$. If an element with the key u is paired with an element with the key v, we draw an edge between the vertices u and v. Observe that the resulting multi-graph may have parallel edges and self-loops. Further, since each key appears an even number of times in the input array, every vertex in G has an even number of incident edges (each self-loop counts twice).

Now, our goal is to orient the edges (i.e., assign a direction to each edge), such that every vertex has the same in-degree and out-degree. In particular, if there is a directed edge (u, v), it means that during one encounter with the pair u and v, we should rearrange them such that u appears on the left and v appears on the right. Clearly, if every vertex has the same in-degree and out-degree, it means that the corresponding key appears the same number of times in the left and right halves.

To achieve this, we find an Euler tour in the exchange graph G and then orient the edges accordingly. To make the algorithm oblivious and efficient, we preprocess G to compress it into a simple graph. In particular, the preprocessing removes parallel edges and self-loops as they occur (see lines 4-6 of Algorithm 1). More specifically, if there are r edges between two vertices u and v, we do the following preprocessing:

- *Case 1: r is even.* In this case, we can assign $r/2$ of these edges one direction and the remaining $r/2$ the opposite direction. We can prune all these r edges, i.e. there is no edge left between u and v.
- *Case 2: r is odd.* In this case, we can assign $(r-1)/2$ of these edges one direction, and $(r-1)/2$ of them the opposite direction. We can prune these $r-1$ edges, such that there is only one edge left between u and v whose direction remains to be assigned.

With the above pre-processing, the pruned graph G always has 0 or 1 edge remaining between every pair of vertices. Therefore, the total number of edges is at most $p^2 \in O(\log N)$, and the adjacency matrix hence fits in $O(1)$ memory words. Moreover, every time we prune a pair of parallel edges or a self-loop, we reduce the degree of the endpoint(s) by 2, so all vertices still have even degree after the preprocessing. At this moment, we find an Euler tour to orient the remaining edges (see lines 7-12 of Algorithm 1).

Oblivious Euler-tour algorithm for small graphs. Without the obliviousness requirement, there is a standard Euler-tour algorithm with $O(p^2)$ overhead where p^2 is the maximum number of edges for a graph with p vertices. Unfortunately, the standard Euler-tour algorithm is not oblivious. Our key insight is that we can have an oblivious version of the standard Euler-tour algorithm, as long as the adjacency matrix of the graph can fit in $O(1)$ memory words, that is, the number of vertices $p \in O(\sqrt{\log N})$. In this way, we can obliviously access each entry of the adjacency matrix by invoking $O(1)$ word-level operations supported by the RAM. Further, to ensure obliviousness, we also need to make sure that the Euler-tour algorithm does not abort prematurely which may leak the number of edges in the graph. In our oblivious implementation, we make sure that the loop always iterates for a fixed number of iterations (see line 8 of Algorithm 1).

Detailed algorithm description. We give a full description of our $\mathsf{Balance}$ algorithm in Algorithm 1. Basically,

- *Construct pre-processed graph.* Lines 4-6 construct the simple graph G where parallel edges and self-loops have been pruned. This part requires $O(n)$ numerical computation and no exchange.
- *Euler tour.* Lines 7-12 finds an Euler tour using depth-first search (DFS), and the orientation of the edges are stored in another directed simple graph D whose adjacency matrix can also be packed into $O(1)$ words. Starting from vertex 0, we traverse G through unvisited edges until reaching a dead end. The unvisited edge can be indexed using the least significant bit (LSB) operation. As each vertex has an even degree, we are guaranteed to return to the starting vertex. We then move to the next vertex and repeat the process until all the edges are visited. To make the search oblivious, we always pad the number of iterations to the worst-case, which equals the number of vertices plus the number of edges. In other words, we perform fake operations after all vertices have been visited. This part requires $O(p + \min(p^2, n)) = O(\min(\log N, n))$ numerical computation as long as $p \in O(\sqrt{\log N})$, and no exchange.
- *Conditional exchange of elements.* Lines 13-18 exchange the element pairs based on the orientation of their corresponding edge in D. Some pairs may not have any corresponding edge in D because they appear even number of times and all the edges got pruned. Therefore, we add an edge in D between any pair that is not directly connected (the direction can be arbitrary, see line 13). Every time a conditional exchange occurs between two elements with keys u and v, we reverse the edge in D between vertices u and v in D (see line

Algorithm 1 Balance(A, p)

Input: Input array A contains n elements each marked with a key in $\{0, 1, ..., p-1\}$. Let k_i denote the key of the i-th element. Each key occurs even times.
Output: A is rearranged such that each key appears the same number of times in the left and the right half. Also, the last element of A should not change.
// All `if` conditionals use fake accesses to ensure that the access patterns are identical for both branches.

```
1: $n \leftarrow |A|$, $m \leftarrow n/2$
2: if $n = 2$ then return
3: Construct a simple graph $G$ and directed graph $D$, both with $p$ vertices numbered from 0 to $p-1$ and
   no edge initially. Represent each graph with an adjacency matrix and pack it in a word.
4: for $i \leftarrow 0$ to $m-1$ do
5:     if $(k_i, k_{i+m}) \in G$ then prune $(k_i, k_{i+m})$ of $G$
6:     else if $k_i \neq k_{i+m}$ then add $(k_i, k_{i+m})$ to $G$.
7: Start the walk at vertex $t \leftarrow 0$.
8: for $p + \min(m, \frac{1}{2}p(p-1))$ iterations do
9:     if $\exists\, v$ such that $(t, v) \in G$ then
10:        Add $(t, v)$ to $D$ and delete it from $G$.
11:        $t \leftarrow v$
12:    else $t \leftarrow \min(t+1, p-1)$
13: For all $0 \le u < v < p$, add directed edge $(u, v)$ to $D$ if there is no edge between $u$ and $v$ in $D$.
14: if $(k_{m-1}, k_{n-1}) \notin D$ then reverse all edges in $D$.
15: Reverse the edge between $k_{m-1}$ and $k_{n-1}$ in $D$.
16: for $i \leftarrow 0$ to $m-2$ do
17:    Exchange $A[i]$ and $A[i+m]$ if $(k_i, k_{i+m}) \notin D$.
18:    Reverse the edge between $k_i$ and $k_{i+m}$ in $D$.
```

18), so that next time u and v will be arranged in the opposite order. This effectively guarantees that the pruned edges between u and v are assigned to either direction the same number of times. This part takes $O(n)$ numerical computation and $n/2 - 1$ exchanges.

As an optimization, we can avoid exchanging the last element pair by conditionally negate the adjacency matrix and use a reversed Euler tour (see line 14-15 of Algorithm 1).

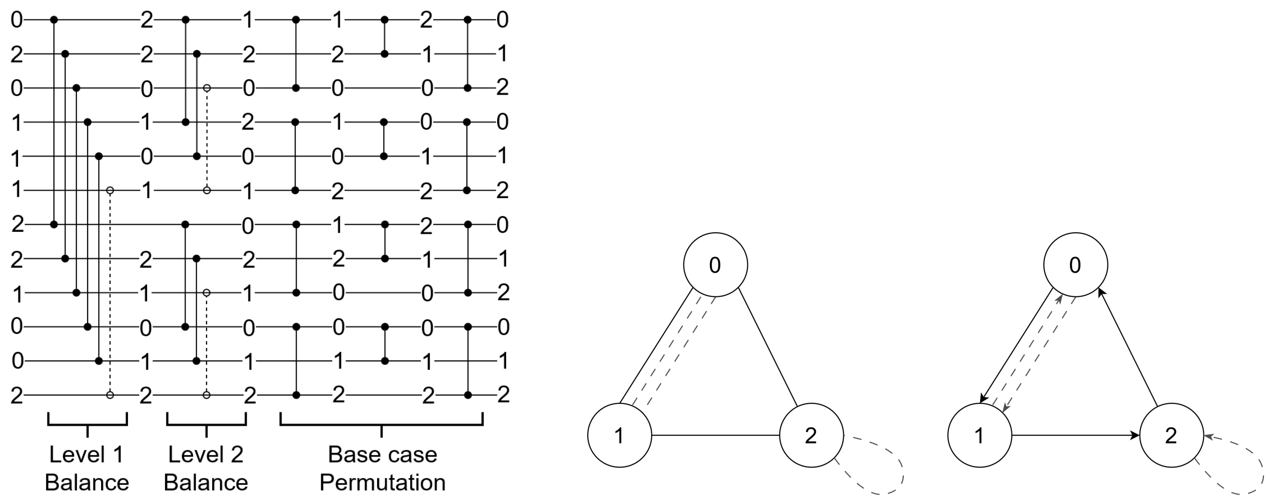
Lemma 3.1 (Computational overhead of Balance)**.** *Algorithm 1 incurs $O(n)$ numerical computation and $n/2 - 1$ exchanges.*

Lemma 3.2 (Obliviousness of Algorithm 1)**.** *The memory access patterns of Algorithm 1 are deterministic and depend only on the length of the input array and the parameter p but not the contents of the array.*

Proof. As mentioned, the adjacency matrices of G and D can each be packed into $O(1)$ words. In this way, accessing an entry in the adjacency matrices requires $O(1)$ word-level operations. Further, all `if` conditionals use fake accesses to make the access patterns for both branches the same. Hence, lines 4-6 where we construct the pre-processed graph G have fixed access patterns. Lines 7-12 where we find the Euler tour iterate for a fixed number of times and the access patterns within each iteration of the loop are fixed. Similarly, Lines 13-18 where we perform the actual conditional exchanges also enjoy fixed access patterns. $\square$

3.3 Interleave

Syntax. Interleave receives an input array containing $n = pZ$ elements marked with keys in $\{0, 1, \ldots, p-1\}$, each key appearing exactly Z times. The output is a rearranged array such that the i-th element has



(a) Interleave network for $p = 3$ and $Z = 4$ (b) Level 1 exchange graph. (c) Euler tour.

Figure 4: An example of the Interleave algorithm. The algorithm recursively balances the input and runs permutation at the base case. In 4a, the dotted lines mean the two elements will not be swapped. In 4b and 4c, solid and dashed edges jointly denote the exchange graph G before pre-processing, and the solid edges denote the graph G after pre-processing.

Algorithm 2 Interleave(A, p)

Input: The input array A contains $n = pZ$ elements each marked with a key from $\{0, 1, \ldots, p-1\}$. Each key appears exactly Z times and Z is a power of two. We require $p \in O(\sqrt{\log N})$.
Output: A is rearranged so that the i-th element has key $i \mod p$.

```
1: $n \leftarrow |A|$
2: **if** $n = p$ **then**: PERMUTE($A$); **return**
3: BALANCE($A$, $p$)
4: INTERLEAVE($A[0 : \frac{n}{2} - 1]$, $p$)
5: INTERLEAVE($A[\frac{n}{2} : n - 1]$, $p$)
```

key $i \mod p$. For $p = 3$ and $Z = 4$ as an example, the output elements should have the key sequence $0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2$.

Constraints. Z is a power of two, $p \in O(\sqrt{\log N})$.

Intuition. The basic idea behind Interleave is to recursively balance the number of each key on the left and right half of the array. At the base case, each key appears excatly once, and we use a permutation network to arrange them in order.

Detailed algorithm. Algorithm 2 shows the Interleave procedure. At line 3 we call Balance so that each key appears $Z/2$ times on the left and $Z/2$ times on the right. As Z is a power of two, we can call Interleave on both halves recursively.

For the base case, there are p elements with distinct keys in $\{0, 1, ..., p-1\}$. In Section A, we define the Permute algorithm to sort these elements obliviously in $O(p \log p)$ time for $p \in O(\sqrt{\log N})$.

Lemma 3.3 (Computation cost of Interleave)**.** *Algorithm 2 incurs $O(n \log n)$ numerical computation and no more than $\frac{1}{2}n(\log n + \log p)$ exchanges.*

Lemma 3.4 (Obliviousness of Algorithm 2)**.** *The memory access patterns of Algorithm 2 are deterministic and depend only on the length of the input array and the parameter p but not the contents of the array.*

We defer the proofs of Theorem 3.3 and Theorem 3.4 to Section C.2.

3.4 Multi-way MergeSplit

Syntax. A p-way MergeSplit takes in p input bins each containing Z elements. Each element is either a real or a filler element. Every real element has a key in $0, 1, ..., p-1$, and a payload string. The goal of the MergeSplit function is to redistribute the real elements so that all elements with key j appear in the j-th output bin. All output bins are padded with fillers to a maximum capacity of Z. If any bin overflows (i.e., if any key appears more than Z times in the input), the algorithm simply aborts. For the special case $p = 2$, this is exactly the MergeSplit primitive used in the original bucket o-sort by Asharov et al. [4].

Constraints. $p \in O(\sqrt{\log N})$ and $Z \in 2^{O(\sqrt{\log N})}$.[1] We require the bin size Z to be a power of 2, but do *not* require p to be a power of 2. This weakened precondition provides flexibility for constructing the butterfly network in Section 4.

Obliviousness requirement. For obliviousness, we require that for any input where each key does not appear more than Z times, the memory access patterns of the algorithm are deterministic and the same.

Intuition. Our new MergeSplit algorithm has a preprocessing step to obliviously mark every filler element also with a key so that each distinct key appears exactly Z times. Next, we call Interleave to rearrange elements into chunks of size p where each chunk contains elements with keys ordered from 0 to $p-1$. Finally, we create the bins by extracting the corresponding elements from each chunk.

Detailed algorithm. Algorithm 3 shows the MergeSplit procedure. The preprocessing involves two linear passes over the keys. The first pass counts the occurrences of each distinct key. If any key appears more than Z times, we detect a bin overflow and abort. The second pass marks the fillers and ensures that each key appears Z times. We make both passes oblivious and achieve linear runtime using a packing trick similar to that in algorithm 1. Note that representing all p counters requires $p \cdot \log Z \in O(\log N)$ bits. Using bitwise CPU instructions such as shifting and bitwise AND, we can pack all the counters into $O(1)$ memory words and update a counter obliviously in constant time.

Lemma 3.5 (Computation cost of MergeSplit)**.** *Algorithm 3 incurs $O(pZ \log(pZ))$ numerical computation and no more than $pZ(\frac{1}{2}\log Z + \log p + 1)$ exchanges.*

Lemma 3.6 (Obliviousness of Algorithm 3)**.** *As long as the input promises that each key appears no more than Z times, then the memory access patterns of Algorithm 3 are deterministic and depend only on the length of the input array and the parameter p but not the contents of the array.*

We defer the proofs of Theorem 3.5 and Theorem 3.6 to Section C.2.

4 Flexway O-Sort

4.1 Basic Algorithm and External-Memory Efficiency

As mentioned, our basic algorithm is the following: we use the same framework of multi-way bucket o-sort [43] with the following changes: 1) we set the number of ways $p \in \Theta(\sqrt{\log N})$, and 2) we adopt a new p-way MergeSplit algorithm described in Section 3.4.

This gives us an oblivious random binning algorithm, from which we can easily construct an oblivious random permutation. To get oblivious sorting, we need to run a non-oblivious comparison-based sort after

[1]Our flexway butterfly sorting network later chooses $Z = \log^c N$ for a constant $c > 1$ to get both negligible in N failure probability and optimal computational overhead.

Algorithm 3 p-way MergeSplit for $p \in O(\sqrt{\log N})$

Input: $A := A_0||A_1||\dots||A_{p-1}$ where $p \in O(\sqrt{\log N})$. For $j \in \{0, \dots, p-1\}$, each bin A_j has size Z where Z is a power of 2 and $Z \in 2^{O(\sqrt{\log N})}$. Each A_j contains real and filler elements; and each real element has a key from $\{0, \dots, p-1\}$.

Output: p bins denoted $A' = A'_0||A'_1||\dots A'_{p-1}$. We want to route all real elements in the input with key k to A'_k, padded with fillers to a size of Z. Output Abort if any bin overflows.

```
1: for $k \leftarrow 0$ to $p-1$ do    // Preprocess
2:     $C_k \leftarrow$ Count of real elements marked with key $k$.
3:     Abort if any $C_k > Z$.
4:     Mark the next $Z - C_k$ fillers in $A$ with key $k$.
5: Interleave($A$, $p$)
6: for $k \leftarrow 0$ to $p-1$ do    // Transpose
7:     $A'_k \leftarrow [A[k], A[p+k], ..., A[p(Z-1)+k]]$
```

* *The preprocessing can be implemented obliviously with $O(1)$ linear scans by packing the counts in $O(1)$ words.*

the oblivious random permutation. The work of Ramachandran and Shi [43] adopts the SPMS algorithm which is theoretically optimal in terms of both work and number of page swaps, but concretely inefficient. In our work, we use the external merge-sort algorithm [33] to realize the non-oblivious sort.

External-memory efficiency. The above basic algorithm achieves $O(N \log N)$ work. At this moment, we explain why this algorithm can achieve an optimal number of page swaps as observed by Ramachandran and Shi [43]. The reason is that the butterfly network structure enjoys good *locality* — see Figure 5 (left). Instead of working on the MergeSplit operations layer by layer, we can work on them k layers at a time: each time we fetch a batch of bins into the enclave and perform k layers of MergeSplit operations among those bins. Then, we perform a *matrix transposition* operation to bring together bins that will be grouped in the next k layer's MergeSplit operations. In Figure 5 (left), we show a 2-way butterfly network (rather than multi-way) for clarity, and we use both $k = 1$ and $k = 2$ due to imperfect rounding. Standard analysis shows that if the batch size is $\Theta(M)$, the above approach achieves $O(N/B \cdot \log_{M/B}(N/B))$ number of page swaps [43], which is optimal even for non-oblivious sort [1].

4.2 Other Optimizations

So far, we have significantly improved the constants relative to Ramachandran and Shi [43] while maintaining the asymptotical optimality. However, to achieve the constants claimed in Table 1, we need several additional tricks.

New rounding technique and flexible ways. Prior works [43, 4] insist both the bin size Z and the number of ways each MergeSplit performs to be powers of two. Consequently, they have to round the number of elements in the butterfly network to the next power of two, leading to a potential $2\times$ overhead. We instead do not impose the power-of-two constraint on the number of ways, and moreover, allow a non-uniform number of ways at different levels of the butterfly network. More concretely, we have the following problem. Recall that initially, the number of bins per level is NBin $:= (1+\epsilon)N/Z$. Our goal is to round NBin up to some integer that can be expressed as as NBin$^* := p_1 \times p_2 \times \dots \times p_L$, and moreover:

1. For every $\ell \in [L]$, $\lfloor\sqrt{\log N}\rfloor/2 \le p_\ell \le \lfloor\sqrt{\log N}\rfloor$;
2. NBin$^* = (1 + o(1))$NBin.

The first condition ensures that the number of ways is not too small and hence the total work is $O(N \log N)$. The second condition makes sure that the rounding introduces only $(1 + o(1))\times$ overhead. An example of such a flexway butterfly network is shown in Figure 2c. In Section C.1, we show how to find a solution for any NBin.

Our flexway butterfly also requires a corresponding modification in our external-memory implementation. Specifically, the number of layers k we fetch in a batch will also be non-uniform at different depths in the butterfly. At any point of time, we always pick largest k such that the batch size can maximally utilize the enclave's EPC memory (of size B).

In-place butterfly and saving matrix transposition. As shown in Figure 5 (left), the original multi-way bucket o-sort [43] suffers from several concrete inefficiencies. First, MergeSplit operations are performed *not in-place*, i.e., the outputs cannot be written to the input bins. This requires extra working buffer in the enclave's EPC memory, and thus reduces the effective M parameter. Second, in between every k layers of MergeSplit operations, the original multi-way bucket o-sort relies on a *matrix transposition* algorithm to rearrange bins, such that bins that are grouped in the next k layers of MergeSplit become close together. This matrix transposition increases the concrete overhead.

To address the these overheads, our new algorithm shown in Figure 5 (right) introduces the following optimizations:

1. *Avoid matrix transposition.* We set the bin size to be at least a page size, that is, 4 KB. In this way, we can skip the matrix transpose and directly fetch the relevant bins from discontiguous memory locations into the enclave.

2. *In-place operations.* By indexing the buckets in a reverse lexicographical order, our butterfly network supports in-place reads and writes both within the enclave and for accesses to external memory.

   First, within the enclave, whenever a MergeSplit operates on a set of bins, we write the outputs to the same set of bins. Second, when the enclave fetches a set of bins from the external memory and performs a batch of MergeSplit operations, the output bins are written back to the same locations in external memory.

   The in-place optimization improves the memory utilization by $2\times$ and hence reduces the page swap overhead.

Solver for optimal concrete parameters. Given the enclave EPC memory size M, the page size B, the input size N, and the desired security parameter, we need to choose several internal parameters, including the bucket size, number of buckets per layer, batch size, and the number of ways at each level of the butterfly network. We implemented a solver to search for the parameters that minimizes the overhead of rounding, and maximally balances the computation and the page swap overhead. We defer the details to Section D.1.

Table 2 lists the optimal butterfly network parameters determined by our solver for various input sizes. It also lists the concrete constants for element-wise exchanges and page swaps in our o-sort algorithm. Compared with the theoretically calculated constants of Table 1, here, the constants for element-wise exchanges are larger, but the constants for page swaps are smaller — this is by intention since the solver attempts to balance these two sources of overheads and minimize the total running time under our testing setup.

5 Experimental Results

5.1 Experimental Setup

We evaluated the sorting algorithms on an Intel Xeon Platinum 8352S processor with 2.2 GHz base frequency and DDR4 RAM. We defer implementation details to Section D.

We evaluated both the [**EPC** $>$ **data**] and the [**EPC** $<$ **data**] settings. In the latter setting, we limit the EPC size to 128 MB which matches SGX v1. In both settings, we varied the number of elements and the size of the payload strings.

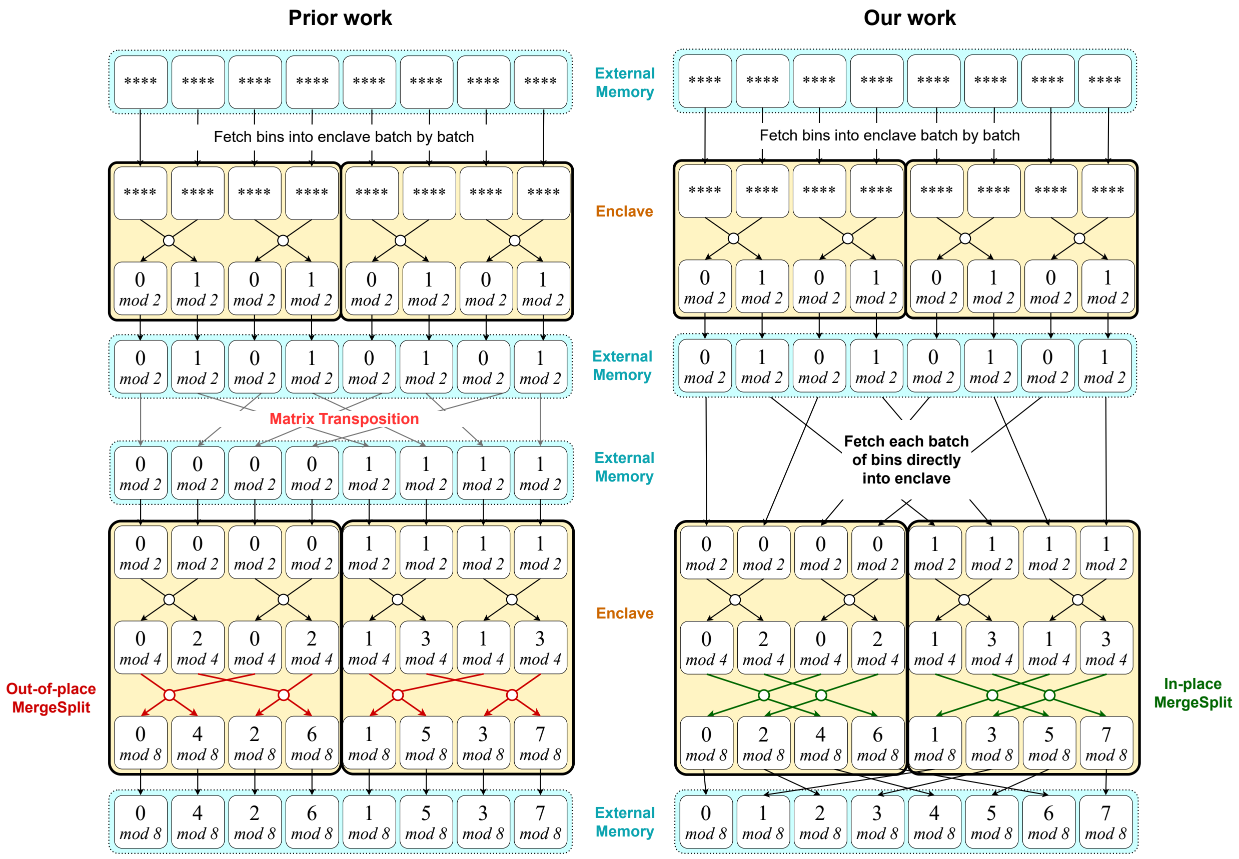


Figure 5: We use an in-place variant of the butterfly, and we get rid of the matrix transposition.

Table 2: Optimal concrete parameters generated by our solver for a security failure probability of 2^{-60} or smaller. The parameters are optimized for 128-byte elements, 128 MB EPC, and 4 KB pages. To speed up the Euler tour search in the Balance algorithm, the solver sets the number of ways $p \leq 8$ so that each adjacency matrix fit within a single 64-bit word. For larger elements, the number of ways may increase as the element-wise exchanges dominate the computation.

N	Z	Butterfly Network Structure	Element-wise Exchanges	Page Swaps
10^6	4096	$(8 \times 6) \times 6$	$3.95 N \log N$	$2.19 \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$
10^7	8192	$(7 \times 7) \times (4 \times 7)$	$4.05 N \log N$	$1.75 \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$
10^8	4096	$(5 \times 6 \times 6) \times (4 \times 5 \times 8)$	$4.01 N \log N$	$1.52 \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$
10^9	16384	$(5 \times 8) \times (5 \times 8) \times (6 \times 7)$	$4.14 N \log N$	$1.93 \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$
10^{10}	4096	$(2 \times 8 \times 8) \times (5 \times 6 \times 6) \times (2 \times 8 \times 8)$	$4.13 N \log N$	$1.82 \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$

Baselines. We compare with the following baselines:

- *Prior oblivious sort algorithms.* We compare with bitonic sort [6], randomized shell short [26], and multi-way bucket o-sort [43]. For the multi-way bucket o-sort, we did not implement the version in the original paper [43] due to its astronomical constants. Instead, we use the "oblivious bin placement" algorithm described in earlier work [11] for multi-way MergeSplit, and external-memory mergeSort for the comparison-based sort.
- *Non-oblivious baseline.* For the [EPC > data] setting, we use `std::sort` as the non-oblivious baseline, and for the [EPC < data] setting, we use external-memory MergeSort as the non-oblivious baseline.

5.2 Results on Oblivious Sorting

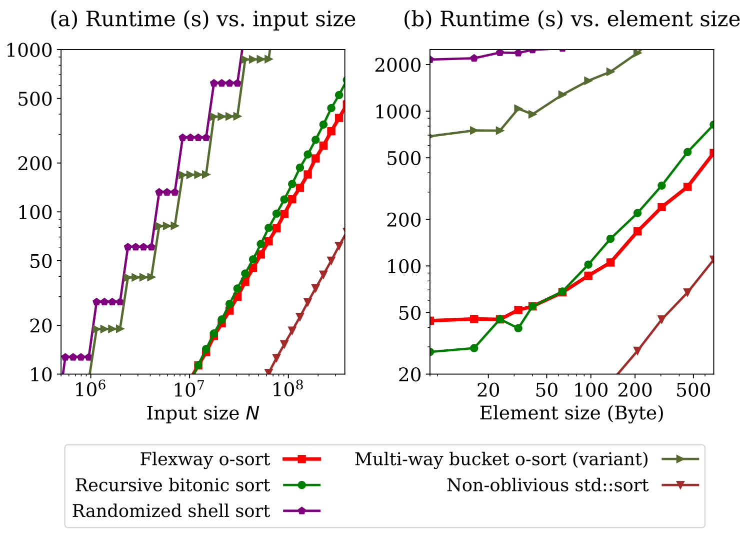


Figure 6: Comparing our sorting algorithm with prior works when EPC $\geq$ data. Fig (a) shows the runtime in relation to the input size, where each element consists of an 8-byte key and a 120-byte payload. Fig (b) fixs an input size of 100 million and varies the size of each element.

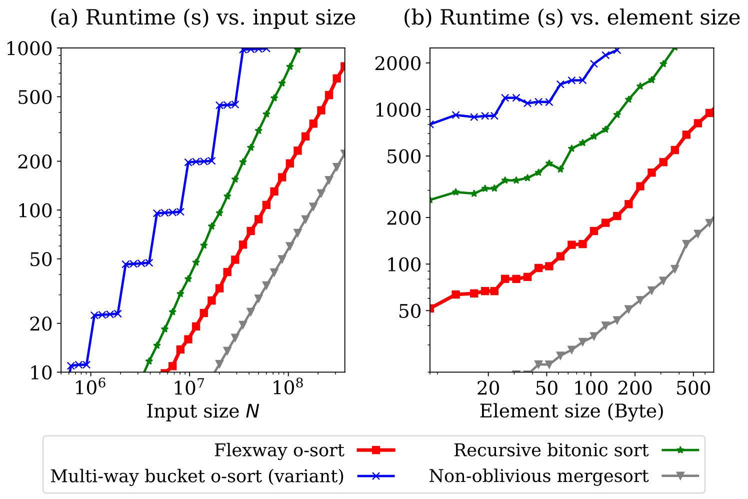


Figure 7: Comparing our sorting algorithm with prior works (128 MB EPC). Other parameters same as Figure 6.

The "EPC $>$ data" setting. For an input of 100 million 128-byte elements, our flexway o-sort is 28.8× faster than randomized shell sort [26], 14.3× faster than multi-way bucket o-sort [43], and 32% faster than the recursive bitonic sort. As the input size expands, our speedup over bitonic sort increases due to our asymptotic enhancement. For example, with 1 billion elements each 200-bytes-wide, our flexway o-sort becomes 88% faster than the recursive bitonic sort.

The "EPC $<$ data" setting. When sorting 100 million 128-byte elements using a 128 MB EPC, our flexway o-sort is at least 12.4× faster than the multi-way bucket o-sort algorithm [43], 38× faster than a non-recursive implementation of bitonic sort, and 4.1× faster than a recursive version of bitonic. At a larger data size of 1 billion with 200-bytes-wide elements, our flexway o-sort becomes 7.2× faster than the recursive bitonic sort.

Performance of MergeSplit. Figure 8 shows a micro benchmark of the multi-way MergeSplit algorithm. Our new algorithm is approximately 18× faster than the naïve approach using two bitonic sorts [6, 11], and 2× faster than a multi-way instantiation from OrCompact [45, 39].

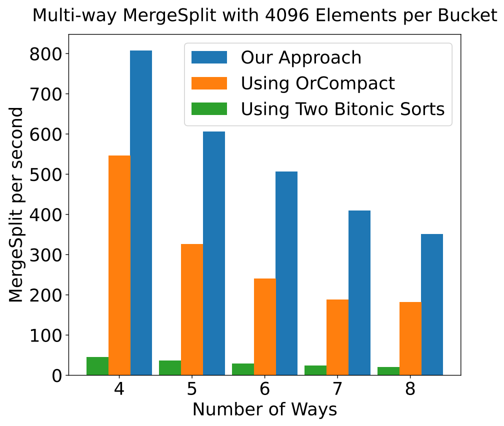


Figure 8: Throughput of our multi-way MergeSplit compared with prior methods. Each bin contains 4096 elements of 136 bytes, including the 8-byte label.

5.3 Applications

Table 3 compares the runtime of three applications implemented with our algorithm and the recursive bitonic sort:

- **Histogram**: obliviously count frequency of 256-byte URLs.
- **ORAM initialization**: obliviously initializing an ORAM tree. We implement the initialization algorithm described in EnigMap [51], assuming each data entry is 128 bytes.
- **DB join**: left join two tables based on an 8-byte ID. Apart from the ID, each row is 256 bytes long for both tables.

In the histogram application, with an input size of 2^{26} elements, our algorithm is 5.5× faster than the baseline given a 128 MB EPC, and 2.3× faster when the EPC is unlimited. Notably, the speedup surpasses the results in Figure 6 and Figure 7. This is because histogram requires costly oblivious comparisons between long URLs, and our algorithm incurs significantly fewer comparisons than bitonic sort.

For ORAM initialization, our algorithm achieves a 5.0× speedup over the baseline with a limited 128 MB EPC, and a 44% acceleration when the EPC is unlimited.

For database Join, when both tables have 2^{26} rows, our algorithm is 4.9× faster than the baseline with a 128 MB EPC, and 65% faster when the EPC is unlimited.

Table 3: Benchmark Results for Different Applications.

Application	Input size	Runtime (s) (128 MB EPC)		Runtime (s) (EPC $\geq$ data)	
		Ours	Bitonic	Ours	Bitonic
Histogram	2^{23}	**45.66**	187.2	**35.14**	71.80
	2^{26}	**447.5**	2448	**328.6**	760.6
ORAM init	2^{23}	**153.0**	529.1	**123.7**	139.8
	2^{26}	**1483**	7400	**1086**	1568
DB join	2^{23}	**247.2**	715.1	**132.1**	172.7
	2^{26}	**2038**	9954	**1206**	1985

5.4 Results on Oblivious Shuffling

Since our flexway o-sort first obliviously shuffles the array and then applies a non-oblivious comparison-based sort, we also obtain an oblivious shuffle as a by-product.

For shuffling 100 million elements of 128 bytes with no EPC limit, our flexway o-shuffle is 72.5× faster than the Waks-on/Waks-off o-shuffle [46] and 2.3× faster than OrShuffle [45]. When running in SGX with a 128 MB EPC, our flexway o-shuffle algorithm is 5.5× faster than OrShuffle [45] and 16× faster than the multi-way bucket o-shuffle [43].

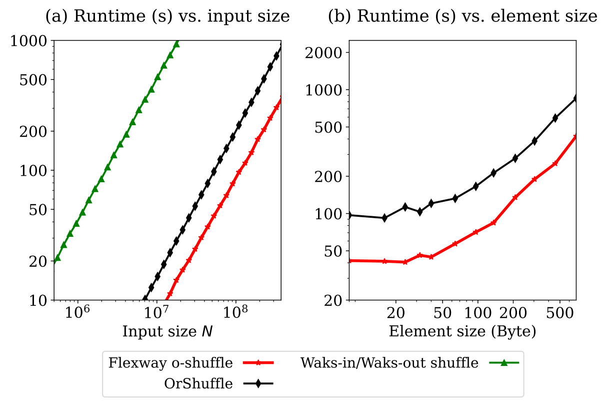


Figure 9: Comparing our shuffling algorithm with prior works when EPC size $\geq$ data size. Parameters same as Fig. 6.

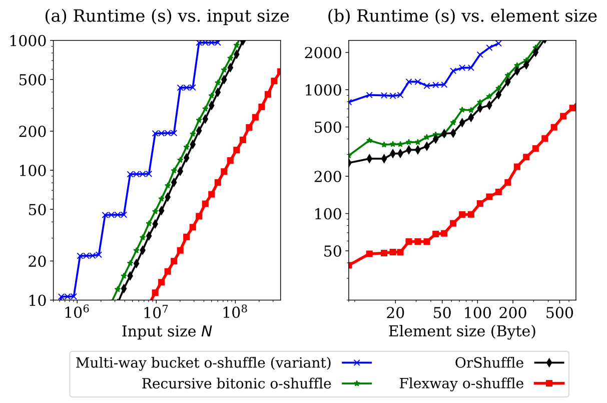


Figure 10: Comparing our shuffling algorithm with prior works (128 MB EPC). Parameters same as Fig. 7.

6 Concurrent Work

Concurrent work [39] implements oblivious sorting and shuffling in a distributed model using the same algorithmic framework as ours. Earlier work [10] observed that butterfly networks are well-suited not only for external-memory algorithms but also for distributed settings.

In comparison, our new MergeSplit building block is both asymptotically and concretely faster than theirs and can serve as a drop-in replacement in their framework. As shown in Figure 8, our MergeSplit algorithm is approximately 2× faster than OrCompact [45], the algorithm employed in [39]. Since MergeSplit dominates the computation for large inputs, this improvement is expected to reduce the computational overhead in [39] by nearly 50%.

7 Conclusion

In this paper, we introduced flexway o-sort, a new oblivious sorting algorithm that is asymptotically optimal, concretely efficient, and well-suited for hardware enclaves. Future work will focus on parallelizing the algorithm to further improve performance and adapting the algorithm to circuits for protocols such as multi-party computation and fully-homomorphic encryption.

8 Acknowledgments

This work is in part supported by NSF awards 2128519 and 2044679, an ONR grant, and a DARPA SIEVE grant under a subcontract from SRI. For Gu, Tinoco, and Shi, the majority of the work was conducted while the authors were at CMU.

References

[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, sep 1988.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *STOC*, 1983.

[3] A. K. M. Mubashwir Alam, Sagar Sharma, and Keke Chen. Sgx-mr: Regulating dataflows for protecting access patterns of data-intensive sgx applications. *Proceedings on Privacy Enhancing Technologies*, 2021:5 – 20, 2020.

[4] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *SOSA*, 2020.

[5] Ré mi Bardenet and Odalric-Ambrym Maillard. Concentration inequalities for sampling without replacement. *Bernoulli*, 21(3), aug 2015.

[6] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS*, 1968.

[7] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIA CCS*, 2013.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[9] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page Table-Based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[10] T.-H. Hubert Chan, Kai-Min Chung, Wei-Kai Lin, and Elaine Shi. MPC for MPC: secure computation on a massively parallel computing architecture. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 75:1–75:52. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[11] T-H. Hubert Chan and Elaine Shi. Circuit opram: Unifying statistically and computationally secure orams and oprams. 2017.

[12] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *ACM Comput. Surv.*, 56(9), apr 2024.

[13] Shumo Chu, Danyang Zhuo, Elaine Shi, and T.-H. Hubert Chan. Differentially oblivious database joins: Overcoming the worst-case curse of fully oblivious algorithms. In *2nd Conference on Information-Theoretic Cryptography, ITC 2021, July 23-26, 2021, Virtual Conference*, volume 199 of *LIPIcs*, pages 19:1–19:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[14] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. *ACM Trans. Parallel Comput.*, 3(4), mar 2017.

[15] Duke CPS196.1. Duke cps196.1 bitonic sort implementation.

[16] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 655–671, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. Benchmarking the second generation of intel sgx hardware. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, oct 2019.

[19] Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. In *STOC*, 2019.

[20] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 555–568. ACM, 2017.

[21] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, page 1–7, New York, NY, USA, 1990. Association for Computing Machinery.

[22] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 285–297, 1999.

[23] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[24] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[25] Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. *CoRR*, abs/1103.5102, 2011.

[26] Michael T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6), dec 2011.

[27] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in O(N Log N) time. In *STOC*, 2014.

[28] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[29] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[30] William Holland, Olga Ohrimenko, and Anthony Wirth. Efficient oblivious permutation via the waksman network. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 771–783, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal oblivious priority queues and offline oblivious RAM. In *SODA*, 2021.

[32] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *Asiacrypt*, 2014.

[33] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[34] Hans Werner Lang. Bitonic sorting network for n not a power of 2. `https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm`, 2018.

[35] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In *SODA*, 2019.

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

[37] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296, 2018.

[38] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 377–394. IEEE Computer Society, 2015.

[39] N. Ngai, I. Demertzis, J. Ghareh Chamani, and D. Papadopoulos. Distributed & scalable oblivious sorting and shuffling. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 156–156, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[40] M. S. Paterson. Improved sorting networks with $o(\log n)$ depth. In *Algorithmica*, 1990.

[41] Enoch Peserico. Deterministic oblivious distribution (and tight compaction) in linear time. *CoRR*, abs/1807.06719, 2018.

[42] Thomas Pornin. Bearssl: A small ssl/tls library.

[43] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 373–384. ACM, 2021.

[44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, New York, NY, USA, 2009. ACM.

[45] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *ACM CCS*, 2022.

[46] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 3328–3342, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 842–858. IEEE, 2020.

[48] Signal. Technology deep dive: Building a faster oram layer for enclaves. `https://signal.org/blog/building-faster-oram/`, 2022.

[49] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310. ACM, 2013.

[50] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2435–2450. ACM, 2017.

[51] Afonso Tinoco, Sixiang Gao, and Elaine Shi. Enigmap : External-memory oblivious map for secure enclaves. In *Usenix Security*, 2023.

[52] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, jan 1968.

[53] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 850–861, New York, NY, USA, 2015. Association for Computing Machinery.

[54] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 503–516. ACM, 2015.

[55] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

A Additional Algorithmic Details

At the base case of Interleave, there are p elements with distinct keys in $\{0, 1, ..., p-1\}$, and we need to sort them in ascending order. While we can apply a concretely-efficient sorting network in practice given that p is small, it does not satisfy our requirement for an asymptotically $O(n \log n)$ MergeSplit algorithm. Instead, our construction applies a Waksman's permutation network [52] to reorder the elements obliviously using no more than $p \log p$ exchanges. For p being power of two, Waksman [52] gave an algorithm to calculate the routing plan using a permutation matrix of size $p \times p$. Since $p \in O(\sqrt{\log N})$ and each entry of the matrix can be represented with one bit, we can make the algorithm oblivious by packing the permutation matrix into $O(1)$ words. However, as we want our butterfly network to have flexible ways, we need to emulate the permutation network for arbitrary p no more than $\sqrt{\log N}$. In this section, we describe how to achieve this goal by recursively applying our Balance algorithm.

Syntax. Permute takes in an array A with n elements whose keys are a permutation of $\{0, 1, \ldots, n-1\}$. The goal of Permute is to rearrange the array A such that the i-th element has key i.

Constraints. We require $n \in O(\sqrt{\log N})$.

Intuition: when n is a power of 2. Our permutation network structure is the same as Waksman, but we propose an algorithm that can obliviously calculate the routing plan along the way using Balance as a primitive. For simplicity, we first assume n to be a power of two. As a preprocessing step, we modify each element's key to be its original key modulo $n/2$. The set of keys hence becomes $\{0, 1, ..., n/2-1\}$ and each distinct key appears exactly twice. Also, the number of distinct keys is no more than $\sqrt{\log N}$. With these preconditions, we can call Balance on the array, so that both the left and right half contain a suit of these new keys. We then call Permute on each half recursively. As a result, for $i \in \{0, 1, ..., n/2-1\}$, both $A[i]$ and $A[i+n/2]$ have new key i, which means their original keys are i and $i+n/2$. To complete the permutation, we just need to conditionally exchange $A[i]$ and $A[i+n/2]$, putting the one with the original key i at the front.

Detailed algorithm: when n is not a power of 2. Algorithm 4 gives a full description of Permute and generalizes it to handle lengths that are non-powers of 2.

- *Handling lengths that are non-powers of* 2. If the current n is not even, we pad a filler element with key n at the end of the array (line 3). Since Balance does not change the last element, the filler is still at the end, and we may exclude it after performing the Balance operation. In an actual implementation, the filler can be imaginary and need not occupy any extra space.
- *Save and restore the original keys without extra space.* When we modify an element's key k to the new key $k \mod \lceil n/2 \rceil$, we need to save the original key k. This can be achieved without extra space. Conceptually, imagine that each element owns a stack, and when we modify k to the new key $k \mod \lceil n/2 \rceil$, we push the bit whether $k \geq \lceil n/2 \rceil$ to its stack (lines 5-6). This way, we can recover the original key k after the recursion (lines 12-13). Since storing $k \mod \lceil n/2 \rceil$ and whether $k \geq \lceil n/2 \rceil$ does not take up more bits than storing the original k, we can integrate the stack into the same word that stores the key.

Example. For example, Figure 11 depicts the network structure of the Permute algorithm for $n = 7$ (see Algorithm 4). The first two levels of the network are recursive calls of Balance. Then, in the remaining levels, elements are exchanged by comparing the restored keys. This network uses 14 exchanges, which is better than an optimal comparison-based sorting network for $n = 7$ using 16 compare-and-exchanges [33].

Lemma A.1 (Computational overhead of Permute)**.** *Algorithm 4 incurs $O(n \log n)$ numerical computation and no more than $n \log n$ exchanges.*

Proof. Let $f(n)$ denote the number of exchanges to Permute an input of size n. When n is even, Balance involves $n/2-1$ exchanges, and $n/2$ exchanges are performed at the end. We need to solve two subproblems of size $n/2$. When n is odd, Balance involves $(n+1)/2-1$ exchanges, and $(n-1)/2$ exchanges are performed

Algorithm 4 Permute(A)

Input: The input array A contains n elements, where $n \in O(\sqrt{\log N})$. The i-th element has a key $\pi(i)$ along with a payload, where π is a permutation of set $\{0, 1, ..., n-1\}$. Also, we let each element own a stack, which can be integrated into the same word storing the key.

Output: A is rearranged such that the i-th element has key i.

```
1: $n \leftarrow |A|$, $m \leftarrow \lceil n/2 \rceil$
2: **if** $n = 1$ **then return**
3: If $n$ is odd, pad a filler with key $n$ at the end of $A$.
4: **for** $i \leftarrow 0$ **to** $2m - 1$ **do**
5:     $b \leftarrow \lfloor A[i].key \,/\, m \rfloor$. Push $b$ to $A[i].stack$.
6:     $A[i].key \leftarrow A[i].key \mod m$
7: Balance($A$, $m$)
8: If $n$ is odd, remove the last element of $A$.
9: Permute($A[0 : m-1]$)
10: Permute($A[m : n-1]$)
11: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
12:     Pop $b$ from $A[i].stack$.
13:     $A[i].key \leftarrow A[i].key + b \cdot m$
14: **for** $i \leftarrow 0$ **to** $\lfloor n/2 \rfloor - 1$ **do**
15:     Obliviously exchange $A[i]$ and $A[i+m]$. Put the one with smaller key to the front.
```

at the end. We need to solve two subproblems of size $(n + 1)/2$ and $(n - 1)/2$. To conclude, $f(n) = f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + n - 1$. For the base cases, $f(1) = 0$ and $f(2) = 1$. We can inductively prove that $f(n) \leq n \log(n-1)$ for $n \geq 3$.

Finally, since Balance requires $O(n)$ numerical computation, it follows that Permute requires $O(n \log n)$ numerical computation. □

Lemma A.2 (Obliviousness of Algorithm 4). *The memory access patterns of Algorithm 4 are deterministic and depend only on the length of the input array but not the contents of the array.*

Proof. Clearly, all the `if` conditions depend only on the input size n, so does the number of loop cycles. As shown in Theorem 3.2, the access patterns of the sub-procedure Balance depend only on the length of its input A and the parameter m. Further, given the length of the original array, the input lengths and m to all recursions are fixed. Finally, the conditional exchanges at line 15 also enjoy deterministic and fixed access patterns. □

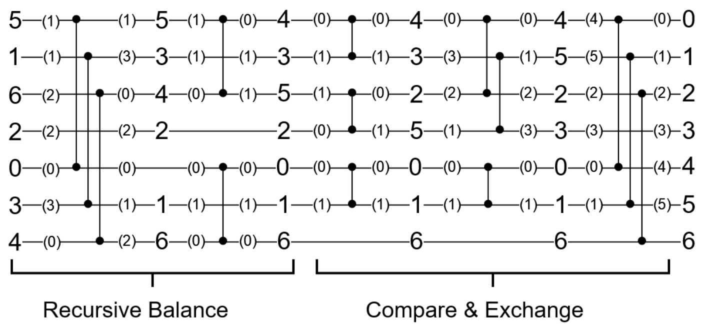


Figure 11: Permute network for $n = 7$. The solid numbers are the original keys. The numbers in the parentheses on the left of the solid numbers are the modified keys used in the previous Balance operation, and the numbers in the parentheses on the right of the solid numbers are the modified keys used in the next Balance operation.

B The Distribution O-Sort Algorithm

In this section, we suggest *distribution o-sort*, a variant of our flexway o-sort algorithm that achieves tight constant for the number of page swaps at the price of higher computation overhead. Under our evaluation setup, the trade-off causes distribution o-sort to be slightly slower than flexway o-sort. However, distribution o-sort may be favored in setups where data are swapped to slow HDD or remote storage. Moreover, distribution o-sort is theoretically interesting as it achieves tight number of page swaps up to the constant.

We present the algorithm under the *strong* tall cache assumption which is suitable for hardware enclaves, that is, assuming that the page size $B = \log^c N$ and the enclave memory size $M = B^{\omega(1)}$.

In distribution o-sort, elements are assigned to bins directly based on their ranks rather than randomly-assigned labels. Specifically, we route elements through the butterfly network by comparing them with $\Theta(N/\log^{3+\epsilon'} N)$ pivots that are sampled randomly from the input array.

This modification eliminates the need for the external-memory merge sort, thus saving an additive $\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}$ term in page swap cost.

B.1 Algorithm Description

Load-balancing the input array using page-wise shuffling. For load-balancing purpose, we require the input elements to be distinct and randomly shuffled. To ensure elements are distinct, we can assign each element its index in the input array as a tie-breaker.

The initial shuffling is more challenging to achieve. Although it doesn't need to be oblivious, i.e., we do not need to hide the permutation that is applied, we cannot directly apply a linear-time shuffling algorithm such as the Fisher-Yates since it suffers from $\Theta(N)$ page swaps. Therefore, instead of doing a full permutation, we perform a weaker version that shuffles the input array only on a page granularity.

To accomplish this, we will simply initialize a random permutation π on $[1 \ldots N/B]$, and to read the i-th page of the permuted array, we simply read the $\pi[i]$-th page of the original array. Initializing the random permutation π can be accomplished by invoking our earlier flexway o-shuffle algorithm on N/B indices, or using a more efficient non-oblivious variant.

While the permutation π itself may not fit into the enclave memory, the access to π is sequential and hence incurs $|\pi|/B$ additional page swaps during the execution of the entire algorithm. This translates into an $o(1)$ multiplicative overhead.

Later in Section B.2, we show that even such a weak page-wise permutation achieves sufficient load-balancing, as long as we set the final partition size to be a poly-logarithmic factor larger than the page size — because of this choice of partition size, we will need the strong tall cache assumption to get our desired asymptotic bounds.

Obliviously finding approximate pivots. Before instantiating the butterfly network, we want to find $q - 1$ pivots that are approximately the q-quantiles. The high-level idea is to obliviously down-sample the original array by a $1/\log N$ factor, and then obliviously sort the down-sampled array. More concretely, we can run the following algorithm where we use a batch size $Z' \geq \log^{3+\epsilon'} N$ for some constant $\epsilon' \in (0, 1)$, and assume N is a multiple of Z'.

- For each iteration $i \in [N/Z']$, do the following:
  - Fetch the i-th batch of Z' elements from the input array $\mathbf{I}$ into the enclave. For each element in the batch, flip a coin that comes up heads with probability $1/\log N$. If the coin is heads, write down the element itself into an output Y_i (which was initialized to be empty), otherwise write down a filler into Y_i.
  - Use oblivious compaction to move all elements marked with 1 in Y_i to the front, and truncate the resulting array at length $2Z'/\log N$, let the outcome be X_i. It can be shown that all elements marked with 1 are in X_i except with negligible probability in N.
- Obliviously sort the array $X_1||X_2||\ldots||X_{N/Z'}$, moving all real elements to the front and sorted by their respective keys, and let the outcome be D. By Chernoff bound, the number of real elements in D is in the range $[(1 - N^{-1/3})N/\log N, (1 + N^{-1/3})N/\log N]$ except with negligible probability.

- Number the elements in D from 0 to $|D|-1$. Choose the pivots to be the elements indexed $0, \lfloor a \rfloor, \lfloor 2a \rfloor, \ldots, \lfloor (q-1) \cdot a \rfloor$ in D where $a = \frac{(1+\delta)N}{q \cdot \log N}$.

Partitioning based on pivots. The next step is to divide elements into q partitions by comparing with the pivots. To achieve negligible overflow probability, we let each partition contain $R \in \Theta(\log^{3+\epsilon'} N)$ elements. The partitions are defined using $q-1$ pivots denoted as $Q_1, Q_2, \ldots, Q_{q-1}$, which approximate the q-quantiles of the input. For convenience of handling border cases, we define $Q_0 = -\infty$ and $Q_q = +\infty$.

To assign elements to partitions, we adapt the flexway butterfly network described in Section 4 with the following modifications:

- We change the number of ways at each level from $\Theta(\sqrt{\log N})$ to $\Theta(\log \log N)$, and we set the bin size $Z \in \Theta(B \log N (\log \log N)^3)$, where B is the page size.
- We calculate the keys for multi-way MergeSplit (Algorithm 3) by comparing the elements with pivots, rather than using the residue of the random labels.
- We rearrange the bins in the butterfly network so that each partition becomes consecutive.

Intuitively, we want to create partitions with finer granularity as the level of the butterfly network increases.

Reusing the notations from Section 4, we set the number of ways at each level as $p_1, \ldots, p_L$, where $p_l \in \Theta(\log \log N)$ and $p_1 \times p_2 \times \ldots \times p_L = q$, and we require that the final partition size R to be a multiple of the bin size Z. The following invariant is maintained: an element e reaches the j-th bin at level ℓ only if $e \in [Q_{di}, Q_{d(i+1)})$, where $i = \lfloor \frac{jZ}{dR} \rfloor$ and $d = \prod_{h > \ell} p_h$. Considering an element in partition $[Q_{di}, Q_{d(i+1)})$, to decide which partition it should go to in the next p-way MergeSplit operation, the element is compared with $p-1$ pivots: $Q_{d(i+1/p)}, Q_{d(i+2/p)}, \ldots, Q_{d(i+(p-1)/p)}$. Note that we cannot perform a binary search here due to the obliviousness requirement, which is the reason why we require $p \in \Theta(\log \log N)$.

Sorting each partition. Once the partitioning is complete, we can apply bitonic sort within each partition and remove all the fillers to obtain the final output. This step is piggybacked to the final level of partitioning, which avoids an extra scan and a total of N/B page swaps can be saved.

B.2 Analysis

The distribution o-sort algorithm achieves complexity stated in the following theorem. Its proofs can be found in Section B.3.

Theorem B.1 (The distribution o-sort algorithm). *Distribution o-sort described in this section achieves the following performance bounds. When the page size $B = \log^c N$ for $c > 0$ and the enclave size $M \in B^{\omega(1)}$, it incurs $(\frac{1}{2} + c + o(1)) N \frac{\log N \log \log N}{\log \log \log N}$ exchanges and $O(N \frac{\log N \log \log N}{\log \log \log N})$ numerical computation and $(1 + o(1)) \frac{N}{B} (\log_{\frac{M}{B}} \frac{N}{B} + 1.5)$ page swaps.*

B.3 Analysis of Distribution O-Sort

Lemma B.2 (Number of sampled elements). *For the algorithm described in Section B, the number of elements sampled in each batch of size $Z' \geq \log^{3+\epsilon'} N$ is no more than $2Z'/\log N$, and the total number of sampled elements is in the range $[(1 - N^{-1/3}) N/\log N, (1 + N^{-1/3}) N/\log N]$ except with $\mathsf{negl}(N)$ probability.*

Proof. By Chernoff bound on the sum of independent variables, the probability that more than $2Z'/\log N$ elements are sampled in a batch of size Z' is upperbounded by

$$\exp(-\Omega(\log^{1+\epsilon'} N)) \subset \mathsf{negl}(N).$$

Similarly, the probability that the total number of sampled elements deviates from $N/\log N$ by $N^{2/3}/\log N$ is upperbounded by

$$\exp(-\Omega((N^{-1/3})^2 N/\log^2 N)) \subset \mathsf{negl}(N).$$

□

Lemma B.3. *Suppose that $N/q \in \Omega(\log^{3+\epsilon'} N)$, and moreover, assume that the original array contains distinct elements. Except with $\mathsf{negl}(N)$ probability, the following holds: for any two consecutive pivots chosen denoted Q_i and Q_{i+1}, the number of elements in the original array in the range $[Q_i, Q_{i+1})$ is in the range $[(1-\delta)N/q, (1+\delta)N/q]$, where $\delta \in \Theta(\frac{1}{\log\log N})$.*

Proof. Consider two adjacent pivots Q_i and Q_{i+1}. The probability that more than $(1+\delta)N/q$ elements in the original array are sandwiched between $[Q_i, Q_{i+1})$ is the same as the probability that $\mathsf{rank}(Q_{i+1}) - \mathsf{rank}(Q_i) > (1+\delta)N/q$ where $\mathsf{rank}(\cdot)$ denotes the rank of an element in the original array. The above probability is upperbounded by the probability that among the elements whose ranks are between $[\mathsf{rank}(Q_i), \mathsf{rank}(Q_i + \lfloor (1+\delta)N/q \rfloor))$, at most $\lceil (1+N^{-1/3})N/(q\log N) \rceil$ of them are selected. Let $X_1, \ldots, X_{\lfloor (1+\delta)N/q-1 \rfloor}$ denote whether each of these elements are selected — these random variables are negatively correlated. By the Chernoff bound for negatively correlated random variables, as long as $N/q = \Omega(\log^{3+\epsilon'} N)$, we have that except with $\mathsf{negl}(N)$ probability, the number of elements in the original array in the range $[Q_i, Q_{i+1})$ is at most $(1+\delta)N/q$. The other direction can be proven in a similar fashion. ☐

Our proof will rely on Bernstein's inequality[5] as described below:

Theorem B.4 (Bernstein's inequality)**.** *Let $\mathcal{X} := (x_1, \ldots, x_m)$ be a population of m points where $x_i \in [0, B]$ for all $i \in [m]$. Let $(X_1, \ldots, X_n)$ be a sample drawn from $\mathcal{X}$ without replacement. Let $\mu := \frac{1}{m}\sum_{i=1}^m x_i$ be the mean of $\mathcal{X}$, and let*

$$\sigma^2 := \frac{1}{m}\sum_{i=1}^m (x_i - \mu)^2$$

be the variance of $\mathcal{X}$. Then, for all $\epsilon > 0$,

$$\Pr[\frac{1}{n}\sum_{i=1}^n X_i - \mu \geq \epsilon] \leq \exp\left(-\frac{n\epsilon^2}{2\sigma^2 + (2/3)\cdot B \cdot \epsilon}\right).$$

Lemma B.5. *Distribution o-sort described in Section B satisfies obliviousness.*

Proof. Similar to Theorem C.2, the algorithm is oblivious since the access patterns within the enclave as well as the page swap patterns are deterministic and depend only on input length N, M, B, the size of each element, and the desired failure probability. Therefore, it suffices to prove that the algorithm achieves sorting except with negligible (in N) probability. By Theorem B.3, the number of elements in partition $[Q_i, Q_j)$ is no more than $(1+\delta)(j-i)N/q$.

Suppose that among all input pages, the number of elements sandwiched between partition $[Q_i, Q_j)$ is $x_1, \ldots, x_{N/B}$. By Theorem B.3, we have that except with $\mathsf{negl}(N)$ probability, it must be that

$$\sum_{j \in [N/B]} x_j \leq (1+\delta)(j-i)N/q.$$

Considering a bin Bin^* in this partition, it can only receive elements from $q/(j-i)$ bins at the initial level, containing at most $\frac{Zq}{(1+\epsilon)(j-i)}$ real elements in total, where $\epsilon \in \delta + \Theta(\frac{1}{\log\log N})$ is the slack factor of padding.

Since the input is randomly shuffled on a page granularity, among the at most $n = \frac{Zq}{(1+\epsilon)(j-i)B}$ pages that can reach Bin^*, let $X_1, \ldots, X_n$ be the number of elements in each of these pages sandwiched between Q_i and Q_j. Let $X = \sum_{j\in[n]} X_j$.

Ignoring the negligible probability that the bad event of Theorem B.3 happen, it holds that

$$\mathbb{E}[X] \leq \frac{(1+\delta)(j-i)N}{qN} \cdot \frac{Zq}{(1+\epsilon)(j-i)} = \frac{(1+\delta)Z}{1+\epsilon}.$$

Similarly, we have that $\mathbb{E}[X] \geq \frac{(1-\delta)Z}{1+\epsilon}$.

Let $\mu = \frac{1}{N/B}\sum_{j\in[N/B]} x_j$, and let

$$\sigma^2 = \frac{1}{N/B}\sum_{j\in[N/B]}(x_j-\mu)^2 = \frac{1}{N/B}(\sum_{j\in[N/B]} x_j^2 - \frac{N}{B}\cdot\mu^2)$$
$$\le \frac{1}{N/B}\sum_{j\in[N/B]} x_j\cdot B - \mu^2 \le B\cdot\mu \le (1+\delta)(j-i)B^2/q.$$

By Bernstein's inequality (Theorem B.4), we have the following:

$$\Pr[X>Z] \le \exp\left(-\frac{n\cdot\left(\frac{(\epsilon-\delta)(1-\delta)Z}{(1+\delta)(1+\epsilon)n}\right)^2}{2\cdot\sigma^2}\right) \le \exp\left(-\Omega\left(\frac{Z^2}{(\log\log N)^2 nB^2/q}\right)\right)$$
$$\le \exp\left(-\Omega\left(\frac{Z}{B(\log\log N)^2}\right)\right).$$

Substituting $Z\in\Theta(B\log N(\log\log N)^3)$ and applying a union bound over all levels and all bins, we obtain the desired bound on the failure probability. □

Lemma B.6 (Computational overhead of distribution o-sort)**.** *When the page size $B=\log^c N$ for $c>0$ and the enclave size $M\in B^{\omega(1)}$, distribution o-sort incurs $\frac{1}{2}(1+c+o(1))N\frac{\log N\log\log N}{\log\log\log N}$ exchanges and $O(N\frac{\log N\log\log N}{\log\log\log N})$ numerical computation.*

Proof. In the sampling process, running OrCompact[45] on each batch of size Z' incurs $O(Z'\log Z')$ exchanges. Since $Z'\in\mathsf{poly}\log N$, it takes $O(N\log\log N)$ exchanges to obtain all the samples. As the sampling rate is $O(\frac{1}{\log N})$, the multiplicative overhead to sort the samples is $o(1)$.

By Theorem C.1, the flexway butterfly network contains no more than $(\log O(N/Z))/\log p$ levels, and MergeSplit is called $(1+o(1))N/(pZ)$ times at each level. By Theorem 3.5, each MergeSplit operation incurs no more than $pZ(\frac{1}{2}\log Z+\log p+1)$ exchanges. Substituting $p\in\Theta(\log\log N)$ and $Z\in\Theta(\log^{1+c}N(\log\log N)^3)$, it requires

$$\frac{1}{2}(1+c+o(1))N\frac{\log N\log\log N}{\log\log\log N}$$

exchanges for routing.

Calling bitonic sort within each partition at the last level requires $\frac{1}{4}R\log R(\log R+1)$ exchanges. Since the size of each partition is $R\in\mathsf{poly}\log N$, it takes $o(N\log N)$ exchanges to sort all the $\frac{(1+o(1))N}{R}$ partitions. Finally, removing fillers takes $O(N)$ exchanges. Hence, the total number of exchanges is

$$\frac{1}{2}(1+c+o(1))N\frac{\log N\log\log N}{\log\log\log N}.$$

For numerical computation, at each level of the butterfly network, every element is compared with $p-1$ pivots. Since there are at most $\Theta(\frac{\log N}{\log p_{\min}})$ levels. The total number of comparisons is $O(\frac{N\log N\log\log N}{\log\log\log N})$, and each comparison takes $O(1)$ numerical computation. Pre-shuffling the input requires at most $O(\frac{N}{B}\log\frac{N}{B})$ numerical computation. The time complexity of all other numerical computation is asymptotically upper-bounded by the number of exchanges, due to the same reasoning as Theorem C.3. Therefore, the time complexity of numerical computation is $O(\frac{N\log N\log\log N}{\log\log\log N})$. □

Lemma B.7 (Number of page swaps for the distribution o-sort algorithm)**.** *When the page size $B=\log^c N$ for $c>0$ and the enclave size $M\in B^{\omega(1)}$, distribution o-sort incurs $((1+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1.5)\frac{N}{B}$ page swaps.*

Proof. The assumptions made about the values of M and B ensure that the final partition can fit within the enclave. By applying similar reasoning as in Theorem C.6, we can determine that it takes $\frac{N}{B}((1+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1)$ page swaps to emulate the butterfly network and sort the final partitions.

The sampling process requires a reading pass over all the elements, but does not require writing them back. By the assumption that page reads and writes have the same overhead, the number of page swaps can be estimated as $0.5N/B$. Since the sampling rate is $O(\frac{1}{\log N})$, the multiplicative overhead to sort the samples is $o(1)$. Similarly, the multiplicative overhead to generate and read the permutation $\pi(\lceil\frac{N}{B}\rceil)$ is also $o(1)$. $\square$

C Deferred Proofs

C.1 Rounding Lemma

When constructing our flexway butterfly network, we rely on the following lemma to show that the multiplicative overhead due to rounding is always $o(1)$.

Lemma C.1 (Rounding lemma)**.** *For integer $N' \geq 2$ and $2 \leq p_{\max} < 2^{\sqrt{\log N'}}$, we can find $p_1, p_2, \ldots, p_L \in \mathbb{N}$ such that for every $\ell \in [L]$,*

$$\lfloor p_{\max}/2 \rfloor \leq p_\ell \leq p_{\max};$$

and moreover,

$$1 \leq \frac{\prod_{\ell\in[L]} p_\ell}{N'} \leq 1 + \frac{1}{\lfloor p_{\max}/2 \rfloor}.$$

Proof. Set $L = \lceil \log_{p_{\max}} N' \rceil$. Define

$$\beta = p_{\max}^L/N', \ \gamma = \frac{p_{\max}}{\lfloor p_{\max}/2 \rfloor}, \text{ and } \kappa = \lceil \log_\gamma \beta \rceil.$$

Immediately, we have $\beta \in [1, p_{\max})$ and $\gamma \geq 2$. This further implies

$$\kappa \leq \lceil \log \beta \rceil \leq \lceil \sqrt{\log N'} \rceil \leq \lceil \frac{\log N'}{\log p_{\max}} \rceil = L.$$

Set

$$p_1 = p_2 = \ldots = p_{\kappa-1} = \frac{p_{\max}}{\gamma},$$
$$p_{\kappa+1} = p_{\kappa+2} = \ldots = p_L = p_{\max},$$

and

$$p_\kappa = \lfloor \frac{\gamma^{\kappa-1}}{\beta} p_{\max} \rfloor + 1.$$

Since, $\log_\gamma \beta \leq \kappa < \log_\gamma \beta + 1$, we have

$$\frac{p_{\max}}{\gamma} + 1 \leq p_\kappa \leq p_{\max}.$$

Moreover,

$$\prod_{\ell\in[L]} p_\ell \geq \frac{p_{\max}^L}{\beta} = N',$$

$$\prod_{\ell\in[L]} p_\ell \leq \frac{p_\kappa}{p_\kappa - 1} \cdot \frac{p_{\max}^L}{\beta} = (1 + \frac{1}{p_{\max}/\gamma})N'.$$

$\square$

C.2 Analyzing Building Blocks

C.2.1 Theorem 3.3: Computation cost of Interleave

Proof. As each distinct key appears n/p times and n/p is a power of two, Interleave involves $\log(n/p)$ levels of recursion to reach the base case. On each level, Balance costs fewer than $n/2$ exchanges and $O(n)$ numerical computation. Permute is called n/p times at the base case, and each takes up to $p \log p$ exchanges and $O(p \log p)$ numerical computation. Hence, Interleave requires no more than $n(\frac{1}{2}\log(n/p) + \log p) = \frac{1}{2}n(\log n + \log p)$ exchanges. Since $n \geq p$, the complexity of numerical computation is $O(n \log n)$. $\square$

C.2.2 Theorem 3.4: Obliviousness of Algorithm 2

Proof. The `if` condition at line 2 only depends on the input size and parameter p. As shown in Theorem 3.2 and Theorem A.2, the Balance and Permute sub-procedures have access patterns that depend only on their input lengths and p, and given the input length of the original array, the input lengths to all recursive calls are fixed. $\square$

C.2.3 Theorem 3.5: Computation cost of MergeSplit

Proof. Preprocessing p bins of size Z requires $O(pZ)$ numerical computation, and the final transposition makes pZ exchanges. By Theorem 3.3, the Interleave procedure incurs $O(pZ\log(pZ))$ numerical computation and no more than $pZ(\frac{1}{2}\log Z + \log p)$ exchanges. Therefore, MergeSplit algorithm requires $O(pZ\log(pZ))$ numerical computation and no more than $pZ(\frac{1}{2}\log Z + \log p + 1)$ exchanges in total. $\square$

C.2.4 Theorem 3.6: Obliviousness of Algorithm 3

Proof. For $p \in O(\sqrt{\log N})$ and $Z \in 2^{O(\sqrt{\log N})}$, it only requires $p\log Z \in O(\log N)$ bits to store all the counts in the preprocessing step. As described in the detailed algorithm, by packing the counts into $O(1)$ words, the preprocessing step have a fixed access pattern. By Theorem 3.4, the call to Interleave is also deterministic and depends only on the input size and p. The transposition at the end clearly enjoys fixed access patterns as well. $\square$

C.3 Analyzing Flexway O-Shuffle

In the analysis below, we set the bin size $Z \in \Omega(\log N(\log\log N)^3)$, and the slack factor $\epsilon \in \Theta(\frac{1}{\log\log N})$.

Lemma C.2. *Our flexway o-shuffle algorithm described in Section 4 is oblivious.*

Proof. The proof is similar to earlier works [4, 43]. If there is no overflow, the input requirements of algorithms 1, 2, 3 will be satisfied, and the access pattern of the oblivious random binning process is deterministic and depend only on input length N, M, B, the size of each element, and the desired failure probability (which in turn decide how we choose the other parameters including Z, ϵ and the number of ways). After the oblivious random binning, we sort each bucket and reveal the number of real elements per bucket. Asharov et al. [4] shows that such leakage is safe because the number of real elements in each last-level bucket is simulatable without knowledge of the input array - it's just the bin loads of a balls-into-bins process. Therefore, it suffices to prove that no bin will receive more than Z elements except with negligible (in N) probability, meaning a simulator that assumes no overflow can produce access patterns statistically indistinguishable from real-world execution. This holds as long as Z is super-logarithmic in N due to the following reasoning.

Consider a fixed bin $A_j^{(\ell)}$ at level ℓ and index j. It can only receive real elements from P_ℓ initial bins, each filled with $Z/(1+\epsilon)$ real elements, where $P_\ell := \prod_{h=1}^{\ell} p_h$. An element with label τ reaches $A_j^{(i)}$ only when $\tau \equiv j \pmod{P_i}$, which occurs with a probability $1/P_i$. Since the labels are chosen independently, we can apply a Chernoff bound to show that $A_j^{(i)}$ overflows with a probability

$$P_{\text{overflow}} = \exp(-\Omega(\epsilon^2 Z)) \le \exp(-\Omega(\log N\log\log N)) = N^{-\Omega(\log\log N)}.$$

At each level, there are $(1+\epsilon)N/Z$ bins, and the butterfly network has a maximum of $\log((1+\epsilon)N/Z)$ levels (corresponding to the case where every level is two-way). By applying a union bound over all levels and all bins, we obtain the desired bound on the failure probability. $\square$

Lemma C.3. *Given bin size $Z = \log^c N$ where $c > 1$, flexway o-shuffle incurs $(1+c+o(1))N\log N$ exchanges and $O(N\log N)$ numerical computation.*

Proof. By Theorem C.1, the flexway butterfly network contains no more than $(\log O(N/Z))/\log p$ levels, and MergeSplit is called $(1+o(1))N/(pZ)$ times at each level. By Theorem 3.5, each MergeSplit operation incurs no more than $pZ(\frac{1}{2}\log Z + \log p + 1)$ exchanges.

Substituting $p \in \Theta(\sqrt{\log N})$ and $Z = \log^c N$, it requires the following number of exchanges for routing:

$$(1+o(1))N\frac{\log O(N/Z)}{\log p}\left(\frac{1}{2}\log Z + \log p + 1\right) \in (1+c+o(1))N\log N.$$

Calling bitonic sort to shuffle each bin at the last level requires $\frac{1}{4}Z\log Z(\log Z+1)$ exchanges. Since the size of each bin is $\mathsf{polylog}\, N$ and the label length is $\Theta(\log N)$, the probability of label collision in each bin is $O(N^{-C})$ for some positive constant C. Therefore, we only need $1+o(1)$ trials in expectation. Since there are $\frac{(1+o(1))N}{Z}$ bins, the expected number of exchanges to sort the bins at the last level is

$$(1+o(1))N\frac{1}{4}\log Z(\log Z+1) \in o(N\log N).$$

Finally, removing fillers takes $O(N)$ exchanges. Hence, the total number of exchanges is

$$(1+c+o(1))N\log N.$$

For numerical computation, labeling all the elements requires $O(N\log N)$ computation, as we assumed that each label has $O(\log N)$ bits and generating each random bit requires constant time. To extract the keys for MergeSplit, only one division and one modulus operation are required per element. Under the word-RAM model, we can determine the multiplicative inverse of all choices of the divisor $p \in O(\sqrt{\log N})$ at compile time, allowing each division and modulo operation to be done in constant time. Since there are $\Theta(N)$ elements and $\Theta(\frac{\log N}{\log\log N})$ levels, key extraction requires $o(N\log N)$ numerical computation in total. For both MergeSplit and bitonic sort, the amount of numerical computation is asymptotically upperbounded by the number of exchanges. Therefore, the time complexity of numerical computation is $O(N\log N)$. $\square$

Corollary C.4. *Given bin size $Z \in \Theta(\log N(\log\log N)^3)$, flexway o-shuffle incurs $(2+o(1))N\log N$ exchanges.*

Proof. We can obtain the result by substituting $c = 1+(3+o(1))\frac{\log\log\log N}{\log\log N} \in 1+o(1)$ in Theorem C.3. $\square$

Lemma C.5. *Given that $B \geq \log^2 N$, $M \geq B^2$ and $Z \in \Theta(\log N(\log\log N)^3)$, the number of page swaps for flexway o-shuffle is*

$$((2+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1)\frac{N}{B}.$$

Proof. By Theorem C.1, there are $(1+o(1))N/Z$ bins in total. Each MergeSplit performs at least $p_{\min} = \frac{1}{2}\sqrt{\log N}$ ways of partitioning and at most $p_{\max} = \sqrt{\log N}$ ways of partitioning.
Hence, the total number of levels in the butterfly network can be upper-bounded as:

$$L_{\text{total}} \in \lceil \log_{p_{\min}}((1+o(1))N/Z)\rceil \subset (1+o(1))\frac{\log N - \log Z}{1/2\cdot\log\log N}$$

A single pass of batch execution can route all elements through at least:

$$L_{\text{batch}} = \lfloor \log_{p_{\max}}\frac{M}{Z}\rfloor \geq \frac{\log M - \log Z}{1/2\cdot\log\log N} - 1$$

levels. Therefore, the number of passes is at most:

$$n_{\text{pass}} = \lceil\frac{L_{\text{total}}}{L_{\text{batch}}}\rceil \in (1+o(1))\frac{\log N - \log Z}{\log M - \log Z - 1/2\cdot\log\log N} + 1.$$

Substituting $Z \in \Theta(\log N(\log\log N)^3)$, we obtain:

$$n_{\text{pass}} \in (1+o(1))\frac{\log N}{\log M - 3/2\log\log N} + 1$$

Given $B \geq \log^2 N$, we can further derive:

$$n_{\text{pass}} \leq (1 + o(1)) \log_{\frac{M}{B}} \frac{N}{B} + 1.$$

Each pass requires $(1+o(1))\frac{N}{B}$ page swaps, except that in the first pass we only need to read $(1+o(1))\frac{N}{B}$ pages and similarly in the last pass we only need to write $(1 + o(1))\frac{N}{B}$ pages. Between every neighboring passes, matrix transposition results in another $(1 + o(1))\frac{N}{B}$ page swaps according to Theorem C.12. Therefore, the total number of page swaps is

$$n_{\text{swap}} \in 2(n_{\text{pass}} - 1)(1 + o(1))\frac{N}{B} + (1 + o(1))\frac{N}{B} = ((2 + o(1)) \log_{\frac{M}{B}} \frac{N}{B} + 1)\frac{N}{B}.$$

□

Theorem C.6 (Flexway o-shuffle for strong tall cache)**.** *When* $B = \log^c N$ *for* $c > 0$ *and* $M \in B^{\omega(1)}$*, by setting* $Z = \max(B, \log N(\log\log N)^3)$ *and eliminating matrix transposition, flexway o-shuffle incurs* $(1 + o(1))\frac{N}{B}(\log_{\frac{M}{B}} \frac{N}{B} + 1)$ *page swaps and* $(1 + o(1))(\max(c, 1) + 1)N \log N$ *exchanges.*

Proof. Similar to Theorem C.5, the total number of levels in the butterfly network can be upper-bounded as:

$$L_{\text{total}} \in (1 + o(1))\frac{\log(N/B)}{1/2 \cdot \log\log N}$$

A single pass of batch execution can route all elements through at least:

$$L_{\text{batch}} = \lfloor \log_{p_{\max}} \frac{M}{Z} \rfloor \geq \frac{\log \frac{M}{B}}{1/2 \cdot \log\log N} - 1$$

levels. Therefore, the number of passes is:

$$n_{\text{pass}} = \lceil \frac{L_{\text{total}}}{L_{\text{batch}}} \rceil \in (1 + o(1)) \log_{\frac{M}{B}} \frac{N}{B} + 1.$$

Each pass requires $(1 + o(1))\frac{N}{B}$ page swaps, except that in the first pass we only need to read $(1 + o(1))\frac{N}{B}$ pages and similarly in the last pass we only need to write $(1 + o(1))\frac{N}{B}$ pages. Therefore, the total number of page swaps is

$$n_{\text{swap}} \in (n_{\text{pass}} - 1)(1 + o(1))\frac{N}{B} + (1 + o(1))\frac{N}{B} = ((1 + o(1)) \log_{\frac{M}{B}} \frac{N}{B} + 1)\frac{N}{B}.$$

The number of exchanges can be obtained by applying Theorem C.3. □

C.4 Analyzing Flexway O-Sort

Lemma C.7. *Our flexway o-sort algorithm described in Section 4 is oblivious.*

Proof. The proof resembles earlier works [4, 43], which showed that if we apply an oblivious shuffling algorithm and then any non-oblivious, comparison-based sort, the resulting algorithm is oblivious. □

Lemma C.8. *Given bin size* $Z = \log^c N$ *where* $c > 1$*, flexway o-sort incurs* $(1.23+c+o(1))N \log N$ *exchanges and* $O(N \log N)$ *numerical computation.*

Proof. When merging multiple sorted chunks in the external memory mergesort, it is sufficient to store pointers in the heap. This approach ensures that each element is copied only once during each pass. Consequently, the complexity of merging in the exchange model can be expressed as $O(N \log_{M/B} \frac{N}{B})$, which is $o(N \log N)$.

At the base case, quick-sort incurs $\frac{\ln 2}{3} n \log n \approx 0.23n \log n$ exchanges in expectation [29]. Suppose that the batch size is M', where $M' \leq N$, the number of exchanges incurred across all base case instances amounts to an expected value of $0.23N \log M' \leq 0.23N \log N$. By Theorem C.3, the expected number of exchanges required by flexway o-sort is

$$(1.23 + c + o(1))N \log N.$$

Since external memory mergesort uses $O(N \log N)$ numerical computation, the overall numerical computation of flexway o-sort is still $O(N \log N)$. □

Corollary C.9. *Given bin size $Z = \log N(\log\log N)^3$, flexway o-sort incurs $(2.23+o(1))N\log N$ exchanges.*

Proof. Similar to the proof of Theorem C.4. □

Lemma C.10. *Given that $M \geq B^2$ and $B \geq \log^2 N$, the number of page swaps for flexway o-sort is*

$$((3+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1)\frac{N}{B}.$$

Proof. First, we show that during the last pass of shuffling, elements can be rearranged into $O(N/M)$ sorted chunks. Let M_{batch} represent the batch size at the last level.
Case 1: If $M_{\text{batch}} \geq M/2$, then there are at most $2(1+o(1))N/M$ batches, with each batch producing a sorted chunk.
Case 2: If $M_{\text{batch}} < M/2$, then we can allocate a buffer of size $M/2$ in the enclave and always copy the shuffled elements first to this buffer. When the buffer is full, we sort all the $M/2$ elements and write them out as a chunk. Consequently, there can be at most $\lceil 2N/M \rceil$ sorted chunks.

Assigning a one-page buffer to each sorted chunk, in the worst-case, the required number of page swaps to merge all the chunks hierarchically is:

$$n'_{\text{swap}} \in (1+o(1))\frac{N}{B}\cdot(1+\log_{\frac{M}{B}} O(\frac{N}{M})).$$

Considering $M \geq B^2$ and $B \geq \log^c N$, we have:

$$n'_{\text{swap}} \in (1+o(1))\frac{N}{B}\cdot\log_{\frac{M}{B}}\frac{N}{B}.$$

Combining this with Theorem C.5, we obtain the target expression. □

Theorem C.11 (Flexway o-sort for strong tall cache)**.** *When $B = \log^c N$ for $c > 0$ and $M \in B^{\omega(1)}$, by setting $Z = \max(B, \log N(\log\log N)^3)$ and eliminating matrix transposition, flexway o-sort incurs $\frac{N}{B}((2+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1)$ page swaps and $(1+o(1))(\max(c,1)+1.23)N\log N$ exchanges.*

Proof. The proof is similar to that of Theorem C.6. The only difference is that we need additional $(1+o(1))(\log_{\frac{M}{B}}\frac{N}{B}+1)$ page swaps and $0.23N\log N$ exchanges to instantiate the external-memory merge sort, as shown in Theorem C.8 and Theorem C.10. □

C.5 Analysis of Matrix Transposition

It is well-known that matrix transposition can be done with $O(N/B)$ page swaps [22]. The following lemma quantifies the constants in the big-O notation. In particular, our oblivious sorting and shuffling algorithms invoke matrix transposition where each atomic element is an entire bin.

Lemma C.12. *Suppose that each bin contains $Z \in \omega(1)$ elements, and that the enclave size $M \geq B^2$, $M < N$. Transposing a matrix containing N/Z bins incurs $(1+o(1))N/B$ page swaps.*

Proof. We apply the cache-agnostic transposition algorithm [22], which operates recursively by dividing the matrix into smaller submatrices until the submatrix can be transposed entirely within the cache.

We demonstrate that the dimensions of the submatrices are sufficiently large at the base case so that only a $o(1)$ percentage of the pages lie on the edge.

Let's consider a row-major matrix with m rows and n columns, where $mn = N/Z$. We use $Q(m,n)$ to denote the number of page swaps required to transpose this matrix. Our goal is to prove that $Q(m,n) \in (1+o(1))mnZ/B$.
Case 1: $\min(m,n) \leq \sqrt{\frac{M}{2Z}}$. Suppose that $m \leq n$. This indicates $n > \sqrt{\frac{2M}{Z}}$, since $M < N$ and $N = mnZ$. The algorithm divides the greater dimension n by 2 and conquer each half until at some point the number of columns n' satisfies $M/2 \leq 2mn'Z \leq M$. At this base case, the input submatrix contains m rows and the

number of column $n' \geq \sqrt{\frac{M}{8Z}} \in \omega(\frac{B}{Z})$. Therefore, it requires no more than $2m+mn'Z/B \in (1+o(1))mn'Z/B$ page reads to fetch the input submatrix. Since the output submatrix is contiguous, it also requires no more than $(1+o(1))mn'Z/B$ page writes to output the result. The recurrence relation is hence

$$Q(m,n) = \begin{cases} (1+o(1))mnZ/B, & 2mnZ \leq M \\ Q(m, \lfloor \frac{n}{2} \rfloor) + Q(m, \lceil \frac{n}{2} \rceil), & 2mnZ > M \end{cases}$$

whose solution is $Q(m,n) \in (1+o(1))mnZ/B$.

The case $n < m$ is analogous.

Case 2: $\min(m,n) > \sqrt{\frac{M}{2Z}}$. The algorithm performs divide-and-conquer until the size of submatrix, $m' \times n'$, satisfies $M/2 \leq 2m'n'Z \leq M$. Since the algorithm always divides the greater dimensions by 2, we have $\frac{1}{2} \leq \frac{m'}{n'} \leq 2$. This further implies $\min(m',n') \geq \sqrt{\frac{M}{8Z}} \in \omega(\frac{B}{Z})$. Therefore, the number of page swaps is no more than $2m' + 2n' + m'n'Z/B \in (1+o(1))m'n'Z/B$.

The recurrence relation is hence

$$Q(m,n) = \begin{cases} (1+o(1))mnZ/B, & \text{if } 2mnZ \leq M, \\ Q\left(\left\lfloor \frac{m}{2} \right\rfloor, n\right) + Q\left(\left\lceil \frac{m}{2} \right\rceil, n\right), & \text{if } 2mnZ < M \text{ and } m \geq n, \\ Q(m, \lfloor \frac{n}{2} \rfloor) + Q(m, \lceil \frac{n}{2} \rceil), & \text{otherwise.} \end{cases}$$

whose solution is $Q(m,n) \in (1+o(1))mnZ/B$. □

D Implementation Details

D.1 Solver for Concrete Parameters

To search for optimal concrete parameters, we developed an automatic solver that works as follows:

- The solver tests all feasible bin sizes Z and selects the one that yields the minimum estimated runtime. For a fixed target failure probability, enlarging the bins reduces the slack factor. However, it also increases the depth of recursion in MergeSplit and may potentially augment the number of page swaps since fewer bins can fit in the enclave. In practice, we limit Z within the range of $[256, 16384]$.
- Once Z is fixed, the solver employs binary search to find the minimum slack factor ϵ that satisfies the desired failure probability which we set to be 2^{-60}. For this step we assume the butterfly network to be two-way only since when it is multi-way, the failure probability can only be better. We utilize the complementary cumulative distribution function of the binomial distribution to tighten the bound of failure probability.
- The solver runs a depth-first search to determine the optimal structure of the butterfly network, considering both the costs of computation and page swaps. The time to perform a p-way MergeSplit is measured as $T_{\text{ms}}(p)$, the time to run bitonic sort within a bin is T_{btnc}, and the time to swap a bin is T_{swap}. We use the following objective and constraints for flexway o-sort and o-shuffle:

$$\min_{L, n_{\text{p}}, p_\ell, a_j} (n_{\text{p}} \cdot T_{\text{swap}} + T_{\text{btnc}} + \sum_{\ell=1}^{L} \frac{T_{\text{ms}}(p_\ell)}{p_\ell}) \cdot \mathsf{NBin}^*$$

$$\begin{aligned} \text{s.t.} \quad & \mathsf{NBin} \leq \mathsf{NBin}^* = \prod_{\ell=1}^{L} p_\ell \\ & M/Z \geq \prod_{i=a_j+1}^{a_{j+1}} p_\ell \quad \forall 0 \leq j \leq n_{\text{p}} \\ & 0 = a_0 < a_1 < a_2 < \ldots < a_{n_{\text{p}}+1} = L \\ & 2 \leq p_\ell \leq \sqrt{w}, \quad p_\ell \in \mathbb{Z}, \quad 1 \leq \ell \leq L \end{aligned}$$

In the formula above, L represents the number of butterfly network levels. The number of bins after rounding up is given by $\mathsf{NBin}^* = \prod_{\ell=1}^{L} p_\ell$, where p_ℓ is the way at level ℓ. The solver schedules n_p passes of page swaps at levels $a_1, a_2, ..., a_{n_\mathrm{p}}$, and the batch size should not exceed the enclave size. The number of ways is upper-bounded by $\sqrt{w}$, where w is the width of a memory word.

D.2 Additional Details

We have implemented our flexway o-sort/shuffle in C++ to showcase their practicality. Our implementation ensures data privacy and authenticity even in the presence of malicious operating systems. Additionally, we have incorporated multiple optimization techniques to enhance efficiency.

Security guarantees. To achieve obliviousness, we eliminate any dependency of branch or memory access on secret data. We employ AES-256-GCM mode to encrypt and authenticate data before swapping them out of the Enclave Page Cache. To maintain the freshness of each page, a unique timestamp is always applied.

Labels and randomness. In flexway o-sort and o-shuffle, each element is wrapped with an 8-byte word consisting of a 63-bit random label and a 1-bit mark for fillers. In distribution o-sort, we also wrap each element with a memory word, consisting of a tie-breaker and a bit to mark fillers. To resist malicious operating systems, we generate the random seed within the enclave using the *sgx_read_rand* function. Subsequently, we employed AES-CTR mode to produce pseudo-random numbers for labels and tie-breakers.

Efficient page swap. While dynamic paging has been supported in SGX v2 [17], we implemented our own page swap mechanism using BearSSL [42] to obtain better performance. On a Microsoft Azure instance running SGX v2 with 378 MB EPC, our custom page swap is 15.9× faster than the default page swaps for sequential accessing. The efficient page swap is applied to both our algorithms and the baselines.

Advanced CPU instructions. We apply Intel's Advanced Vector Extensions 512 (AVX-512) to accelerate the oblivious compare-and-exchange operation. Specifically, we utilize two Blend instructions (VPBLENDMQ) to conditionally swap up to 32 bytes at a time[2]. We employ AVX2 instructions in the preprocessing step of MergeSplit to count the occurrences of all keys. Further, when searching the Euler tour, we apply the CPU's built-in instruction for counting trailing zeros (TZCNT) to skip vertices with degree 0. Lastly, we leverage Intel's Advanced Encryption Standard Instructions (AES-NI) to accelerate encryption, authentication, and pseudo-random number generation.

Implementation of baselines. In our performance evaluation, we compare our algorithms with several baselines. Below, we explain how these baselines are implemented.

- To optimize page swaps in bitonic sort/shuffle and OrShuffle, we utilize a direct map cache, which exhibits 10% ∼ 15% fewer cache misses than a fully-associative LRU cache in our experiment.
- For bitonic sort, we have adopted a recursive implementation that handles arbitrary input sizes [34]. With the aforementioned cache optimization, it is essentially equivalent to the implementation of the ObliDB work [18]. We also include a non-recursive version of bitonic sort that is cited on Wikipedia [15].
- Bitonic shuffle is implemented as described in [45]. Each element is assigned a 64-bit random key for comparison, and the array is sorted accordingly.
- For OrShuffle, we incorporate the optimization using prefix sums, as suggested in [45]. We did not implement the BORPStream algorithm introduced in [45], because by their evaluation results, it is concretely slower than OrShuffle in a non-streaming setup.
- We used the practical variant of the multi-way bucket o-sort and replaced the cache-agnostic SPMS sort[14] with external mergesort to enhance concrete performance.
- The baseline algorithms also utilize AVX-512 instructions to accelerate element exchanges and AES-NI to speed up encryption and authentication.

[2]Although AVX-512 supports blending 512-bit words, it is slower than blending 256-bit words twice on our SkyLake CPU.