Efficient Oblivious Sorting and Shuffling for Hardware Enclaves

***Abstract*—Oblivious sorting is arguably the most important building block in the design of efficient oblivious algorithms. We propose new oblivious sorting algorithms for hardware enclaves. Our algorithms achieve asymptotic optimality in terms of both computational overhead and the number of page swaps the enclave has to make to fetch data from insecure memory or disk. We also aim to minimize the concrete constants inside the big-O. One of our algorithms achieve bounds tight to the constant in terms of the number of page swaps.**

We have implemented our algorithms and made them publicly available through open source. In comparison with (an unoptimized version of) bitonic sort, which is asymptotically non-optimal but the de facto algorithm used in practice, we achieve a speedup of $2000\times$ for 12 GB inputs.

1. Introduction

Although Oblivious RAM [18], [19], [43] provides a generic approach for compiling any program to an oblivious counterpart, for computational tasks in practice, we can often design efficient oblivious algorithms that asymptotically and concretely outperform the generic ORAM simulation. Oblivious sorting [7], [10], [34], [40], [42] is arguably one of the most important building blocks in the design of such efficient oblivious algorithms. Specifically, oblivious sorting is key to common graph algorithms [8], [23], [35] including breadth-first search [8], [23], connected components [40], minimum spanning tree/forest [40], clustering [35], list ranking [40], tree computations with Euler tour [40], and tree contraction [40]. Oblivious sorting can also be used for securely initializing common ORAM algorithms [45] including Path ORAM [44], which has been deployed at a large scale in practice [1]. Moreover, any computational task that can be efficiently expressed as a streaming-Map-Reduce algorithm has an efficient oblivious implementation using oblivious sort [23], [35].

Status quo of oblivious sorting. It is long known that using sorting networks such AKS [3], oblivious sorting can be accomplished with $O(N \log N)$ cost where N denotes the size of the input array; further, $O(N \log N)$ cost is known to be optimal [16], [34] (either assuming the indivisibility model [34] or assuming the Li-Li network coding conjecture [16]). Unfortunately, AKS [3] is completely impractical due to the enormous constants involved in the expander graphs in its construction. Although more efficient $O(N \log N)$ or $\widetilde{O}(N \log N)$ oblivious sorting algorithms[1] exist (e.g., randomized shell sort [21], bucket oblivious sort [4] and improved variants [40]), they have not been the schemes of choice for practical implementation either. Instead, the famous Bitonic sort [7] is the *de facto* oblivious sorting scheme in practice. Even though Bitnoic sort has $\frac{1}{4}N \log N(\log N+1)$ number of compare-and-exchanges, which is non-optimal, its simplicity and small constant make it attractive in practice.

Oblivious sorting for hardware enclaves. In this paper, we consider efficient oblivious sorting algorithms for hardware enclaves. Our motivating application is securely outsourcing data and computation to an untrusted worker equipped with a secure processor such as Intel SGX. The untrusted worker can be a cloud server hosting an outsourced encrypted database, or a blockchain consensus node executing the smart contract logic of confirmed transactions in a privacy-preserving manner. Our goal is to safeguard the privacy of the data from both software attacks (e.g., from an untrusted operating system or co-resident applications) and physical attacks (e.g., from the administrator of the machine).

In the past, the majority of the literature on oblivious algorithms adopt the standard Random Access Machine (RAM) model (i.e., the textbook model of computation for analyzing algorithms), which focuses on minimizing the *computational overhead*. However, as earlier works have pointed out, the standard RAM model is not the best fit for studying algorithms for hardware enclaves, since *page swaps* in and outside the enclave constitute a major performance bottleneck. Specifically, in many applications, the data size is much larger than the amount of protected enclave memory. In these cases, the data is stored in encrypted format in either insecure memory or disk external to the enclave. Whenever the enclave needs to fetch some data from outside the enclave, it must swap the corresponding page into the enclave, and meanwhile swap another page out to make room. The page swap is a heavy-weight operation, since it involves encryption of the page to be swapped out and decryption of the page that is swapped in. Moreover, for Intel SGX, we also need to make an Outside Call (`OCall`) in case a disk swap is needed. Earlier studies [45] have provided detailed benchmarking results showing that the cost of

1. We use $\widetilde{O}(\cdot)$ to hide $\mathsf{poly} \log \log$ factors.

a 4KB page swap can be 66× more expensive than moving a 4KB page in memory.

Challenges. Therefore, to design concretely efficient oblivious sorting algorithms for hardware enclaves, we are faced with the following two main challenges.

1) First, we would like to have an algorithm whose computational overhead is both asymptotically optimal (i.e., $O(N \log N)$), and concretely better than the *de facto* $O(N \log^2 N)$ bitonic sort for moderate to large data sizes.
2) We would like to have algorithms that are not only optimal in terms of *computational overhead*, but also optimal in terms of *page swaps*. This model of algorithm design matches the goal of the classical external-memory model [2].
3) Although a few earlier works [10], [20], [40] have constructed asymptotically optimal oblivious sorting algorithms in the external-memory model, these algorithms are not practically efficient. Therefore, another interesting question is whether we can get an optimal external-memory oblivious sorting algorithm that is tight to the constant in the big-O.

Throughout the paper, we use the following notation which is standard in the external-memory algorithms literature: we use M to denote the size of the enclave's protected memory, B to denote the page size, and N to denote the number of input elements.

1.1. Our Results and Contributions

Asymptotically and concretely optimal external-memory oblivious sorting. We propose two new oblivious sorting algorithms for hardware enclaves, called *flex-way butterfly o-sort* and *flex-way distribution o-sort*, respectively. Both algorithms achieve *asymptotic optimality in both the computational overhead and number of page swaps*, with small constants in the big-O. Our algorithms are designed to minimize the number of page swaps which is a heavy-weight operation for hardware enclaves. Specifically, the flex-way butterfly o-sort algorithm achieves roughly $2\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}$ page swaps, and the flex-way distribution o-sort achieves roughly $\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}$ page swaps for practical choices of M, N, B, i.e., when $B = \log^c N$ for some constant c and $M \in B^{\omega(1)}$ (we call this the "strong tall cache assumption"). Due to the lower bound of [2], $\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}$ number of page swaps is tight to the constant.

Table 1 shows the performance of our algorithms in comparison with prior work. Here we use the number of compare-and-exchanges to characterize the computation overhead, since in our schemes, the cost of compare-and-exchanges dominates the computational overhead. For completeness, besides the strong tall cache assumption mentioned earlier, which is a fit for hardware enclaves, the table also gives the performance of our algorithms (including the constant in the big-O) for the standard tall cache assumption commonly adopted in the standard external-memory algorithms literature.

Results on oblivious shuffling. As a by-product, we also get efficient external-memory algorithms for oblivious shuffling. We list these results also in Table 1. Oblivious shuffling is also considered an important primitive in the design of oblivious algorithms, because various prior works [5], [38], [46] showed a paradigm for designing efficient oblivious algorithms: first obliviously shuffle the input, and then design a corresponding oblivious algorithm for a shuffled array — for various computational tasks, we can enjoy more efficient oblivious algorithms for an obliviously shuffled array than an arbitrary input array.

Technical highlight. To achieve the aforementioned performance bounds, we devise some new building blocks which may be of independent interest. Our algorithm is inspired by the bucket oblivious sort algorithm by Asharov et al. [4] as well as the multi-way generalization proposed by Ramachandran and Shi [40]. However, previously, these algorithms are studied in a theoretical context, and as Table 1 shows, their concrete performance suffers. One interesting technical contribution we make is to devise a new p-way MergeSplit algorithm which is at the core of the (multi-way) bucket oblivious sort. Our new p-way MergeSplit makes use of the following techniques. First, in one key building block called Balance, we rely on an oblivious Euler tour algorithm to balance the number of keys in the left and right halves of the array. To make the Euler tour algorithm oblivious without incurring additional overhead, our key observation is a *packing trick*: as long as the number of ways p is sufficiently small, we can pack the entire adjacency matrix of the graph into a single memory word, which allows us to access an entry of the matrix obliviously with constant overhead. We describe various additional algorithmic tricks in Section B.

Open-source implementation. We implemented our algorithms 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 (URL anonymized), and for the anonymous submission, we provide an anonymous version at https://github.com/oblsort/oblsort.

Evaluation. We evaluated the performance of our oblivious sorting algorithms against the prior work. The results show that for sorting an array of 100 million 128-byte elements, depending on whether disk swaps are incurred, our flex-way butterfly o-sort is **246**× (or **2000**×) faster than an unoptimized recursive implementation of bitonic sort, and at least **12.4**× (or **7.6**×) faster than the multi-way bucket o-sort algorithm of

TABLE 1: **Comparison with prior works.** The theoretically optimal construction [40] suffers from astronomical constants in computation due to the use of expander graphs. For algorithms in our results, we include an additional N/B number of page swaps due to rounding in the worst case.

Algorithm	Page Swaps	Exchanges
	Prior works	
Theoretically optimal [40]	$O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$	$O(N\log N)$
Bitonic (unoptimized) [7]	$(\frac{1}{2}+o(1))\frac{N}{B}\log^2 N$	$(\frac{1}{4}+o(1))N\log^2 N$
Bitonic (optimized) [7], [15]	$(\frac{1}{2}+o(1))\frac{N}{B}\log^2\frac{N}{M}$	$(\frac{1}{4}+o(1))N\log^2 N$
Randomized Shell Sort† [21]	$(24+o(1))N\log\frac{N}{M}$	$24N\log N$
Bucket o-sort [4]	$(4+o(1))\frac{N}{B}\log\frac{N}{B}$	$(1+o(1))N\log N\log^2\log N$
Multi-way bucket o-sort* [40]	$(17+o(1))\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}$	$(16+o(1))N\log N\log\log N$
OR-shuffle [42]	$(\frac{1}{2}+o(1))\frac{N}{B}\log^2\frac{N}{M}$	$(\frac{1}{4}+o(1))N\log^2 N$
	Our results	
Standard tall cache assumption: $B\geq\log^2 N$ *and* $M\geq B^2$*:*		
Flex-way butterfly o-sort	$((3+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1)\frac{N}{B}$	$(2.23+o(1))N\log N$
Flex-way butterfly o-shuffle	$((2+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1)\frac{N}{B}$	$(2+o(1))N\log N$
Strong tall cache assumption: $B=\log^c N$ *and* $M\in B^{\omega(1)}$*:*		
Flex-way butterfly o-sort	$((2+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1)\frac{N}{B}$	$(\max(c,1)+1.23+o(1))N\log N$
Flex-way distri. o-sort	$((1+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1.5)\frac{N}{B}$	$\frac{1}{2}(c+1+o(1))N\frac{\log N\log\log N}{\log\log\log N}$
Flex-way butterfly o-shuffle	$((1+o(1))\log_{\frac{M}{B}}\frac{N}{B}+1)\frac{N}{B}$	$(\max(c,1)+1+o(1))N\log N$

$*$: We replaced the SPMS algorithm in [40] with external-memory merge sort for better concrete performance.
†: Parametrized with inverse-polynomial failure probability.

Ramachandran and Shi [40]. We also implemented an optimized version of bitonic sort using the optimizations in ObliDB [15]. For the same data size as above, our flex-way butterfly o-sort is **4.1**× (or **4.2**×) faster than the optimized version of bitonic; and at a larger data size of 1 billion with 200-bytes-wide elements, our flex-way butterfly sort is **7.2**× (or **4.8**×) faster than the optimized bitonic sort.

For our flex-way distribution o-sort, although it incurs fewer page swaps than our flex-way butterfly o-sort, the computational overhead is slightly higher. Therefore, in a scenario without disk swaps, its performance is roughly comparable to our flex-way butterfly o-sort. With disk swaps, our flex-way distribution o-sort is 2.4× slower than our flex-way butterfly o-sort for sorting an input array of 100 million entries each of 128 bytes. The reason is that the flex-way distribution o-sort has a preprocessing phase that requires randomly accessing pages on disk, and for the SSD drive we use, random access is much slower than sequential access.

We also compare the performance of our oblivious shuffling algorithm with prior work. For shuffling 100 million 128-byte elements, depending on whether disk swaps are involved, our flex-way butterfly o-shuffle is 5.5× (or 7.5×) faster than OrShuffle [42] and 16× (or 10.7×) faster than the multi-way bucket shuffle [40].

2. Preliminaries

2.1. Problem Definition and Threat Model

Problem definition. We consider a general sorting abstraction where the input array contains N elements, each with a *comparison key* (or *key* for short) and a payload string. We assume that the key fits in one memory word, whereas we didn't place any restriction on the payload length. The goal is to sort the input array by the elements' keys. For shuffling, the input is the same, and we want to output a random permutation of the input array.

Threat model. We assume that the server utilizes secure hardware enclaves, such as Intel's SGX, to ensure the integrity of computations. Although recent studies have revealed vulnerabilities in off-the-shelf trusted hardware, such as Spectre [30] and Foreshadow [47], this paper does not address the design of provably secure trusted hardware. While Intel's SGX is employed as a test-bed to showcase our ideas, our algorithmic constructions are applicable to various existing hardware enclave technologies.

Due to the limited capacity of enclave's secure memory, a page swap mechanism is needed to retrieve encrypted pages from external storage while evicting some pages from the internal memory — see Section E.3 for detailed explanation. During the page swap, the operating system can observe the page-level accesses and tamper with the pages it provides. Further, we assume that the OS can also monitor fine-grained memory accesses within the enclave, e.g., through well-known cache-timing attacks [9], [41].

2.2. Computational Model

We assume that the CPU has a constant number of registers. To capture the amount of data transfer across the enclave's boundary, we adopt the external-memory model [2] where we treat the enclave's memory as the "cache", and the outside insecure memory or storage as the "external memory".

Notations. Throughout the paper, we use N to denote the number of input elements we want to sort or shuffle. When we use divide-and-conquer or recursion techniques, we often need to denote the size of a subproblem. We often use n to denote the size of a subproblem, and we explicitly distinguish n from N.

2.2.1. Word Size and Instruction Set. We assume that each memory word is at least $\log N$ bits long, i.e., a memory word can at least store an index into the array. We assume that the following operations on words take constant time: integer addition, multiplication, bitwise AND, XOR, and SHIFT. Using these operations, we can construct a series of constant-time operations on memory words, as elaborated in Section E.2. Besides, we assume that generating a random (or cryptographically secure pseudo-random) bit takes constant time.

2.2.2. SGX Enclave and External-Memory Model. As mentioned in Section 1, the external-memory model [2] is a great fit for hardware enclaves like Intel's SGX. In particular, we care about optimizing both the computation overhead and the number of page swaps.

2.3. Background on Bucket Oblivious Sort

We first review some background on bucket oblivious sort by Asharov et al. [4] and a subsequent multi-way variant by Ramachandran and Shi [40].

2.3.1. Original Bucket Oblivious Sort. Asharov et al. [4]'s idea is to first design an *oblivious random bucket assignment* algorithm, which randomly assigns each input element to one of $O(N/Z)$ output buckets, each of poly-logarithmic capacity Z; further, the obliviousness property requires that the access patterns do not leak the destination bucket of each element. Then, from the oblivious random bucket assignment, they get an *oblivious shuffling* algorithm, which randomly permutes the inputs without revealing the permutation. Finally, they can apply any non-oblivious, comparison-based sorting algorithm to the shuffled array. Specifically, their algorithm works as follows:

Oblivious random bucket assignment. Given an input array of length N, the goal is to obliviously assign each element into a random bucket of capacity $\mathsf{poly}\log N$, and there are in total $2N/Z$ buckets. In Asharov et al. [4], both Z and N must be a power of 2. Notice that each bucket will be half full in expectation.

To achieve this, Asharov et al. [4] uses a butterfly network of buckets, each also of capacity Z. Each bucket can contain *real* elements and *fillers*. The levels in the butterfly network are numbered $0, 1, \ldots, \log \frac{2N}{Z}$, and each level has $2N/Z$ buckets. Initially, a random label of $\log \frac{2N}{Z}$ bits is assigned to each input element. The label denotes which final-layer bucket the element goes to. They place exactly $Z/2$ input elements in each bucket at layer 0, and pad the buckets to a full capacity with fillers. Then, all elements are routed to their destination buckets in the final layer along the topology of the butterfly network. In the butterfly network, a pair of buckets in level ℓ route to a pair of buckets in level $\ell + 1$ — this is accomplished through a MergeSplit operation. The MergeSplit operation takes two input arrays (i.e., buckets) each of size Z, where each array contains real elements marked with either 0 or 1, as well as some fillers. MergeSplit then routes each real element to either the left output bucket or the right output bucket in layer $\ell + 1$ according to their mark, and each output bucket is again padded with fillers to a maximum capacity. Specifically, the ℓ-th bit of the random label is used as the mark at level ℓ.

Oblivious shuffling. An oblivious shuffling algorithm can be obtained by first running an oblivious random bin assignment algorithm on the input array, and then using bitonic sort to sort within each bucket. The real elements are then extracted in sorted order of their labels from each bucket — this step reveals the number of real elements in each bucket, but Asharov et al. [4] proved that this information is simulatable without knowledge of the input array.

Oblivious sort. An oblivious sort algorithm can be obtained by first applying an oblivious shuffling algorithm on the input, and then applying any non-oblivious, *comparison-based* sorting algorithm on the outcome.

Efficiency. Asharov et al. [4] suggested a couple ways to instantiate the MergeSplit primitive.

- For asymptotically optimal computation overhead, they realize each MergeSplit using $O(1)$ number of linear-time oblivious compactions [39]. In this way, the resulting oblivious shuffling and sorting algorithms would achieve $O(N \log N)$ computation overhead, but they did not propose a practically efficient instantiation of linear-time oblivious com-

paction. Known constructions are complicated and suffer from astronomical constants due to their use of expander graphs.

- Alternatively, they suggested using bitonic sort [7] or an $O(n \log n)$ deterministic 0-1 sorting algorithm [34] proposed by Lin, Shi, and Xie to instantiate the MergeSplit. These algorithms are more practical but the resulting oblivious shuffling and sorting algorithms would have $O(N \log N (\log \log N)^2)$ or $O(N \log N \log \log N)$ computation overhead assuming $Z = \mathsf{poly} \log N$.

Asharov et al. [4] did not consider how to implement their bucket oblivious sort algorithm efficiently in an external memory model.

2.3.2. Multi-way Generalization. Ramachandran and Shi [40] proposed a multi-way generalization of the bucket oblivious sort algorithm, where they insisted the number of ways be a power of 2. Take 4-way as an example. In the original bucket oblivious sort, we look at the *next bit* of the element's label at each level to decide whether to route the element left or right. In a 4-way instantiation, we look at the *next two bits* of the element's label at each level to route it in one of the four ways. Equivalently, this can be interpreted as merging each 2×2 batch of 2-way MergeSplit gates in the original bucket oblivious sort into a single 4-way MergeSplit gadget.

Specifically, Ramachandran and Shi [40] suggested using $p \in \Theta(\log N)$ ways to reduce the number of levels in the butterfly network by a $\Theta(\log \log N)$ factor. They suggested using the following approaches for instantiating each p-way MergeSplit as well as the non-oblivious, comparison-based sorting algorithm to be applied to the shuffled array:

- For optimal asymptotics, they suggested using the AKS sorting network [3] or an optimal sorting network for $\log p$-bit keys to instantiate the p-way MergeSplit. Further, they suggested using Sample Partition Merge Sort (SPMS) [12] to instantiate the non-oblivious, comparison-based sorting. In this way, the resulting sorting algorithm achieves $O(N \log N)$ computational overhead and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ page swaps, both of which are asymptotically optimal.
- For a more practical instantiation, they applied a bin-packing algorithm based on bitonic sort [7], [11] to instantiate the p-way MergeSplit, and they suggested a new method to instantiate the non-oblivious, comparison-based sort. The resulting algorithm achieves $O(N \log N \log \log N)$ computational overhead, and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ page swaps.

2.4. Why Prior Approaches are Slow

To the best of our knowledge, the closest to what we want for hardware enclaves is the (practical) oblivious sorting algorithm by Ramachandran and Shi [40], which achieves $O(N \log N \log \log N)$ computational overhead and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ page swaps (see Section 2.3).

However, Ramachandran and Shi's (practical) algorithm [40] also suffers from several inefficiencies asymptotically and concretely:

- *Multi-way* MergeSplit. They lack an efficient instantiation of the p-way MergeSplit. Specifically, they use bitonic sort, which is asymptotically non-optimal.
- *Rounding.* They pad the input array size to the next power of 2, introducing a $2\times$ factor in both computation and page swap overhead in the worst case.
- *Cache-agnostic restriction.* They propose a *cache-agnostic* algorithm, which cannot be configured with the parameters M and B [17]. Due to the specific recursion technique adopted by their cache-agnostic algorithm, they effectively require that the number of levels of the multi-way butterfly network be a power of 2 as well. This introduces another $2\times$ factor in the number of page swaps. For the hardware enclave setting, we typically know the parameters M and B upfront, so it is unnecessary to insist on having a cache-agnostic algorithm.
- *Slack parameter*. They insist that each bucket is half full, which introduces up to $2\times$ overhead to both computation and page swaps.
- *Transposition.* They use a matrix transpose algorithm to rearrange the array between two recursive calls, leading to another $2\times$ increase in the number of page swaps. It also incurs some additional computation but is dominated by other computational costs.
- *Memory marshalling overhead.* Because their algorithm is not in-place, the enclave's memory usage effectively increases by $2\times$. Consequently, their batch size is reduced by a factor of $2\times$, which affects the concrete performance.
- *Other inefficiencies.* Ramachandran and Shi [40] suggested using the SPMS algorithm [12] for instantiating the non-oblivious, comparison-based sort. Unfortunately, the SPMS algorithm holds more theoretical interest than practical application. Since the cache-agnostic property is not needed, we can instead use more efficient instantiations such as external-memory merge sort. In fact, the constants in Table 1 already assumes we have replaced the SPMS algorithm with external-memory merge sort.

3. New Building Blocks

3.1. Balance

Our crucial contribution towards constructing an efficient oblivious sort is an efficient multi-way MergeSplit algorithm, which in turn 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 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 Section D.

Parameter requirements. We require that $p \leq \min(\sqrt{\log N}, n)$.

Intuition: translate Balance **into an Euler-tour on the exchange graph.** First, we pair up elements from the left and right halves of the array A, specifically $A[i]$ and $A[n/2+i]$ for each $i \in 0, \ldots, n/2-1$. Imagine now we scan through all the pairs, and whenever we encounter a pair, we want to decide whether to exchange them or not. Our goal is that after the exchanges, the left and right halves are balanced for every key.

We can equivalently think of this as a graph problem. We create a multi-graph G (called the exchange graph), whose vertices correspond to the p distinct keys $\{0, 1, \ldots, p-1\}$. 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.

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 digraph 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 $0 \leq u < v < p$, 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.

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 $p^2 \leq \log N$, and the adjacency matrix can hence fit in a single memory word. It is also easy to see that in the pre-processed graph, all vertices still have even degree. At this moment, we find an Euler tour to orient the remaining edges (see lines 7-12 of Algorithm 1).

If the Euler tour we find causes the last pair of elements to be swapped, we reverse the Euler tour rather than swap the elements (see line 14-15 of Algorithm 1).

Example. Figure 1a depicts the network structure of the Interleave (see Algorithm 2) algorithm which calls Balance as a building block. In the first level, the input array has the keys $[0, 2, 0, 1, 1, 1, 2, 2, 1, 0, 0, 2]$ and $p = 3$. The union of the solid and dashed edges in Figure 1b depict the multi-graph G. The dashed edges will be oriented and pruned during the preprocessing. We then run an oblivious Euler-tour algorithm to orient the remaining solid edges as depicted in Figure 1c.

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 a single memory word, that is, the number of vertices $p \leq \sqrt{\log N}$. 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 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 a single word. Starting from vertex 0, we traverse G through unvisited edges until a dead end is reached. The unvisited edge can be indexed using the least significant bit (LSB) operation as described in Section 2.2.1. 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 be the worst-case upper bound over all possible inputs, which is 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 $p \leq \min(\sqrt{\log N}, n)$, 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 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.

Finally, we can always 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).

Fact 3.1 (Computation overhead of Balance)**.** *The* Balance *algorithm requires $O(n)$ numerical computation and $n/2 - 1$ exchanges.*

Proof. We can obtain the fact by summing up the cost of each component analyzed above. □

Claim 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 a single word. In this way, accessing an entry in the adjacency matrices requires $O(1)$ word-level operations. Further, recall that all `if` conditionals use fake accesses to ensure that the access patterns for both branches are the same. With this in mind, it is easy to see that lines 4-6 where we construct the pre-processed graph G have deterministic and 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 deterministic and fixed access patterns. □

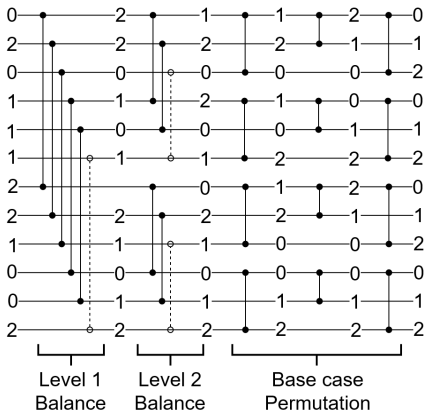
3.2. Interleave

Syntax. Interleave receives an input array containing $n = pZ$ elements marked with keys in $\{0, 1, \ldots, p-1\}$. It is promised that each distinct key appears exactly Z times. The output is a rearranged array such that the i-th element has key $i \mod p$. For $p = 3$ and $Z = 4$ as an example, the output elements should have the key sequence $0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2$.

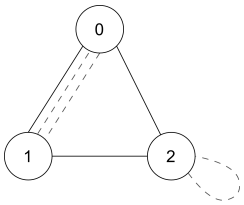
Parameter requirements. We require that Z be a power of two and $p \leq \sqrt{\log N}$.

Intuition: recursive Balance**.** 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 only once, and we can use a permutation network to arrange them in order.

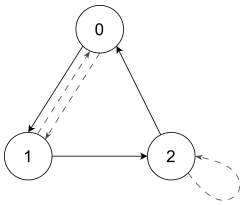
Detailed algorithm. Algorithm 2 shows the Interleave procedure. At line 3 we call Balance so that each key appears $Z/2$ times on the left and $Z/2$ times on the right. As Z is a power of two, we can call Interleave on both halves recursively.



(a) Interleave network for $p = 3$ and $Z = 4$



(b) Level 1 exchange graph.



(c) Euler tour.

Figure 1: An example of the Interleave algorithm. The algorithm recursively balances the input and runs permutation at the base case. In Figure 1a, the dotted lines mean the two elements will not be swapped. In Figure 1b and 1c, 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 $p \le \sqrt{\log N}$.

Output: A is rearranged such 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)

For the base case, there are p elements with distinct keys in $\{0, 1, ..., p-1\}$. In Section D, we show how to modify Waksman's permutation network to sort them obliviously in $p \log p$ time for $p \le \sqrt{\log N}$.

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

Proof. Deferred to Section C.3.1. □

Claim 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.*

Proof. Deferred to Section C.3.2. □

3.3. Multi-way MergeSplit

Syntax. A p-way MergeSplit takes in p input buckets 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 bucket. All output buckets are padded with fillers to a maximum capacity of Z. If any bucket 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 oblivious sort by Asharov et al. [4].

Parameter requirements. We require the number of ways $p \le \sqrt{\log N}$ and the bucket size[2] $Z \le 2^{\sqrt{\log N}}$. We require the bucket 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 4.1.

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 buckets by extracting the corresponding elements from each chunk.

Detailed algorithm. Algorithm 3 shows the MergeSplit procedure. The preprocessing can be achieved with two linear passes on the keys. The first pass counts the occurrences of each distinct key. If any key appears more than Z times, we detect a bucket overflow and simply abort. The second pass marks the fillers and ensures that each key appears Z times. We make both passes oblivious by packing the counts in a single word and updating them through the Extract and Set operations defined in Section 2.2.

2. Our flex-way 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 $p \le \sqrt{\log N}$

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

Output: p buckets denoted $A' = A'_0||A'_1||\ldots A'_{p-1}$. 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 bucket overflows.

Algorithm:

1⋆ For $k \in \{0,\ldots,p-1\}$, let $C_k \leftarrow$ Count of real elements marked with key k in A.
2 Abort if any $C_k > Z$.
3⋆ For $k \leftarrow 0$ **to** $p-1$: mark the next $Z - C_k$ fillers with the key k.
4 Interleave(A, p)
5 For $k \leftarrow 0$ **to** $p-1$: $A'_k \leftarrow [A[k], A[p+k], ..., A[p(Z-1)+k]]$

⋆: *can be accomplished obliviously with a linear scan as long as all counts fit in a single word*

Fact 3.5 (Computation cost of MergeSplit)**.** *The* MergeSplit *algorithm requires* $O(pZ\log(pZ))$ *numerical computation and no more than* $pZ(\frac{1}{2}\log Z + \log p + 1)$ *exchanges.*

Proof. Deferred to Section C.3.3. □

Claim 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.*

Proof. Deferred to Section C.3.4. □

To conclude, in Section 3, we presented a novel multi-way MergeSplit algorithm. For $p \in \Theta(\sqrt{\log N})$ and $Z \in \Omega(\log N)$, our new MergeSplit algorithm reduces computational overhead by a factor of $(2.25 - o(1))\log\log N$ compared to the oblivious Bin-Packing algorithm [11], [40] based on bitonic sort. It also outperforms an implementation using two-way MergeSplit based on OrCompact [42] with a savings factor of $(0.25 - o(1))\log\log N$. In terms of actual run time, our algorithm is about 18 times faster than Bin-Packing and 2 times faster than the implementation based on OrCompact (see Section 5 Figure 8).

4. Flex-Way Butterfly Oblivious Shuffling and Sorting Algorithms

Given our new p-way MergeSplit algorithm in Section 3.3, we use a multi-way butterfly network like Ramachandran and Shi [40] to build oblivious shuffling and sorting. In this section, we describe several optimizations at the butterfly network level which further improve the computational overhead by up to $4\times$, and the number of page swaps by up to $8\times$ under the standard tall-cache assumption that $M \ge B^2$ and $B \ge \log^2 N$. In particular, we save up to $2\times$ in both metrics with a new rounding technique, up to $2\times$ in both metrics with a tightened slack factor, and up to $2\times$ in page swaps that stems from their need to be cache-agnostic. Moreover, under a strong tall cache assumption, we save another $2\times$ factor in page swaps by eliminating the need for matrix transposition. In Section B.1, we also describe how to search for optimal concrete parameters for hardware enclaves.

4.1. Flex-Way Butterfly Network

We explain our flex-way butterfly network designed for the oblivious random bucket assignment. Recall that we can get oblivious shuffling and sorting from oblivious random bucket assignment.

The slack factor ϵ. Asharov et al. [4] and Ramachandran and Shi [40] require that each bucket is half loaded in expectation. A simple observation is that as long as the bucket size Z is super-logarithmic, it suffices to let each bucket be $1/(1+\epsilon)$ loaded in expectation to ensure negligibly small failure, where we set $\epsilon \in \Theta(\frac{1}{\log\log N})$. In practice, when we fix Z, we can choose ϵ accordingly based on the concrete security parameter desired — see Section B.1. In this way, the number of buckets per level is $(1+\epsilon)N/Z$.

A new rounding technique. The earlier work of Ramachandran and Shi [40] (which in turn builds on top of Asharov et al. [4]) requires rounding the input length to the next power of 2. This can incur a $2\times$ overhead in the worst case. To get rid of this $2\times$ factor, we no longer insist on the number of ways to be a power of 2, and moreover, we allow the number of ways to be non-uniform in the entire butterfly network. More concretely, we have the following problem. Recall that originally, the number of buckets per level is $\mathsf{NBkt} := (1+\epsilon)N/Z$. Our goal is to round NBkt up to some integer NBkt^*, and moreover, express the rounded number of buckets as $\mathsf{NBkt}^* := p_1 \times p_2 \times \ldots \times p_L$, such that the following conditions are satisfied:

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

The first condition requires that the number of ways is not too small — this is needed to ensure that the total computational overhead is $O(N\log N)$ since our MergeSplit takes $O(Z\log Z)$ computational overhead. The second condition makes sure that the rounding introduces only $(1+o(1))\times$ overhead rather than $2\times$.

In the Appendix (Lemma C.1), we show that it is always possible to find a feasible rounding-factoring solution for any NBkt.

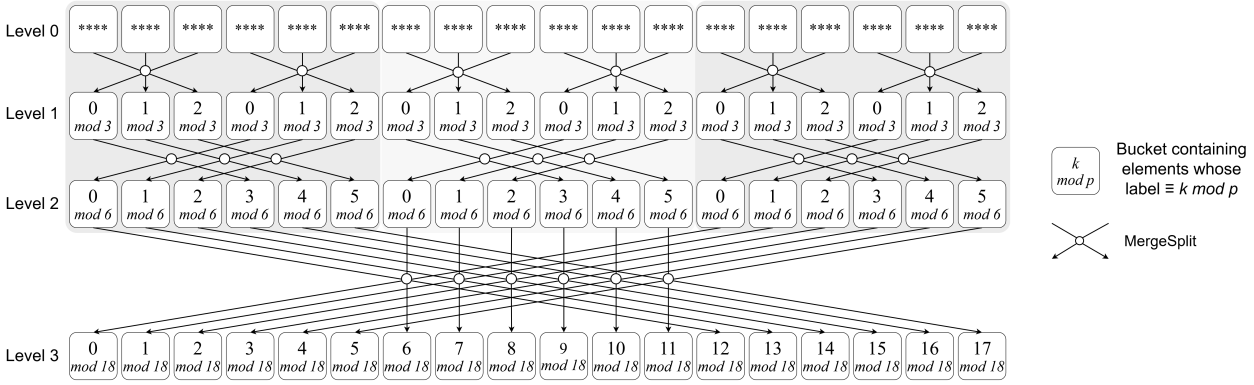


Figure 2: The $3 \times 2 \times 3$ butterfly network assigns elements to 18 buckets according to the residue of their random labels. The routing is in-place as MergeSplit simply overwrites the input buckets.

Our flex-way butterfly network. As mentioned, in our butterfly network, the numbers of ways $p_1, \ldots, p_L$ need not be a power of two, which makes the instantiation of our flex-way butterfly network more flexible. Further, we adopt an *in-place* optimization, which saves the enclave memory usage by a factor of 2.

Specifically, in earlier works [4], [40], the random labels assigned to elements are represented as bit strings. Suppose the number of ways $p = 2^k$, then at each level, the earlier works would look at the next k bits of the label, starting from the *most significant bit*, to determine how to route each element (see also Section 2.3.2).

In our case, we choose an integer label τ uniformly at random from the range $[0, \mathsf{NBkt}^*)$ where $\mathsf{NBkt}^* = p_1 \times \ldots \times p_L$. We maintain the following invariant: an element with the label τ reaches the j-th bucket at level ℓ only if $\tau \equiv j \mod \left(\prod_{h \le \ell} p_h\right)$. For the special case when $p_1 = p_2 = \ldots p_L = 2^k$, this invariant is equivalent to consuming the next k bits at each level, starting from the *least significant bit* of the label. The advantage of this approach is that the data movements now become *in place*, that is, if a set of buckets $j_1, \ldots, j_p$ are the inputs to some MergeSplit instance, then the outputs would be written to exactly the same buckets $j_1, \ldots, j_p$ at the next level. The in-place nature effectively doubles the utilization of the Enclave Page Cache (EPC) and reduces the memory marshalling overhead.

Figure 2 provides an illustrative example of our flex-way butterfly network with structure $\mathsf{NBkt}^* = 3 \times 2 \times 3$, and Figure 4 explicitly compares our approach with prior work [4], [40] for the special case where the butterfly network is two-way.

Optimizing the number of page swaps. We use a similar "batching + matrix transposition" trick as in earlier works [40] to achieve $O((N/B) \log_{M/B} N/B)$ number of page swaps rather than $O(N \log N)$ page swaps if implemented naïvely. In comparison with

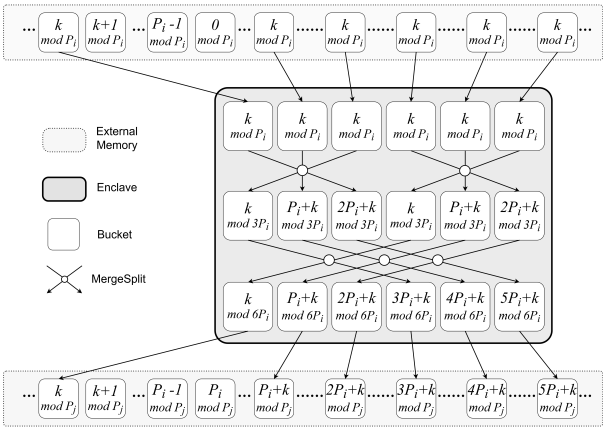


Figure 3: Route elements in batch from the i-th level to the j-th level of the butterfly network. In this example, $j = i + 2$, $p_{i+1} = 3$, $p_{i+2} = 2$, and $P_i := \prod_{h=1}^{i} p_h$.

Ramachandran and Shi [40], the main difference is that we can avoid a $2\times$ blow up due to the cache-agnostic restriction. More specifically, if the EPC can fit $p_i \times p_{i+1} \times \ldots \times p_j$ number of buckets, we can fetch this many buckets in one batch and perform multiple layers of routing within the enclave before writing back the output buckets. We illustrate this batching idea in Figure 3. In general, since the buckets are stored in pages, we also need to apply a *matrix transposition* trick as described in [40], which enhances locality by moving the relevant set of buckets adjacent to each other.

We use a "piggybacking" trick to save a constant number of scans. For example, the preprocessing operations, such as reading input, tagging random labels, and padding fillers, are piggybacked on top of reading and processing the first level of the butterfly network — this saves an extra preprocessing pass. Similarly, recall that at the end of the random bin assignment algorithm, we need to randomly shuffle within each bucket. Here, we piggyback the within-bucket shuffling on top of the

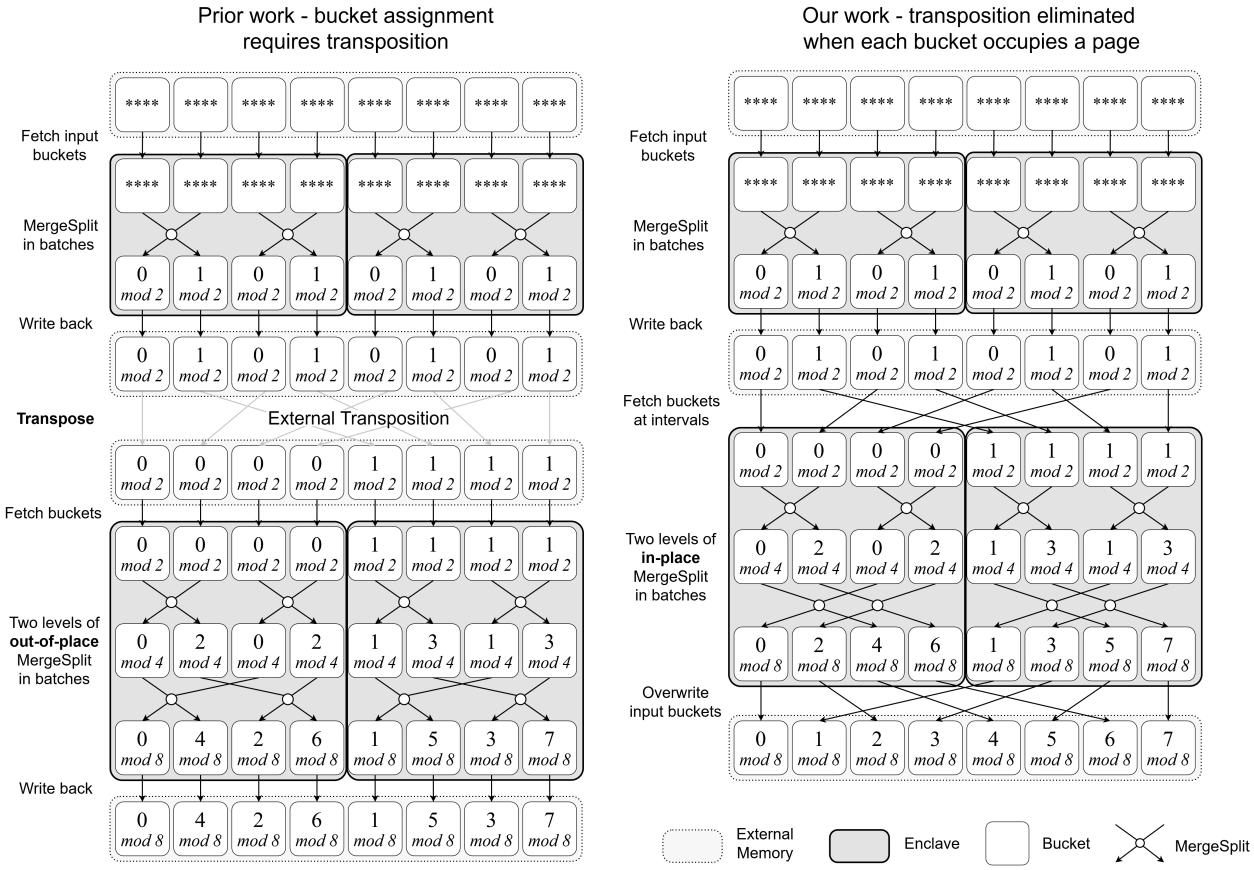


Figure 4: Our practical scheme (on the right) eliminates matrix transposition when each bucket occupies a page. It also enables in-place routing by restructuring the butterfly network.

last level of the butterfly network, which avoids an extra scan. Moreover, the first level of the external-memory merge sort is piggybacked on top of the last level of the butterfly network too (after we perform the within-bucket shuffling), which saves an extra scan.

Putting it all together. Given the oblivious random bucket assignment based on our flex-way butterfly network, we can get oblivious shuffling by randomly permuting within each bucket, and removing all filler elements. Given oblivious shuffling, we can now get oblivious sorting by running an external-memory merge sort algorithm on the shuffled array, which uses quicksort to create sorted chunks that fit in EPC and a priority queue to merge these chunks hierarchically.

To summarize, our oblivious sorting algorithm differs from Ramachandran and Shi [40] in the following aspects. First, we devise a more efficient multi-way MergeSplit algorithm. Second, we use a butterfly network with variable number of ways and constructed to facilitate in-place routing. Third, we applied the simpler and more efficient external-memory mergesort in the non-oblivious sorting phase. Finally, we introduce various other optimizations, including choosing tighter parameters, avoiding the cache-agnostic overhead, and reducing the number of page swaps via piggybacking.

Analysis. Using a similar proof scheme as Ramachandran and Shi [40], we can show that when the bucket size $Z \in \Omega(\log N(\log\log N)^3)$ and the slack factor $\epsilon \in \Theta(1/\log\log N)$, the failure probability will be negligibly small in N, and our algorithms obliviously simulate the shuffling and sorting procedures — see Lemma C.4 and Lemma C.9. The time complexity of the algorithms has been listed in Table 1, and the analysis is deferred to Section C.4 and C.5.

4.2. Further Optimizations Under Strong Tall Cache Assumption

In this section, we describe another optimization for reducing the number of page swaps. The optimization is to make the bucket size equal to the page size[3], which eliminates the need for matrix transposition, as shown

3. When the page size is very small, e.g., when $B \leq \log N$, a bucket may span across multiple pages to ensure the negligible failure probability. For our enclave model, recall that the page size is not an intrinsic invariant, and we can always tune the page size larger. This characteristic also allows us to achieve perfect alignment even though the bucket size is restricted to be a power of two.

in Figure 4. The optimization effectively reduces the number of page swaps by half in the shuffling phase.

As a trade-off, the number of exchanges during the multi-way MergeSplit increases as the bucket size grows. However, under a strong tall cache assumption suitable for hardware enclaves, i.e., assuming that the page size $B \in \log^c N$ for constant $c > 0$ and the enclave size $M \in B^{\omega(1)}$, we can still get asymptotically tight bounds with small constants for both the page swaps and computational overhead. The bounds have been listed in Table 1, and the proofs are deferred to Section C.4 and C.5.

5. Experimental Results

We evaluated the performance of our sorting and shuffling algorithms on an Intel Xeon Platinum 8352S processor, using a single core running at 2.2 GHz frequency. Throughout the evaluation, we restrict the failure probability of all our algorithms within 2^{-60}. The implementation details are described in Section B.

Figure 5 and 6 compares the performance of our algorithms with prior work. Specifically, they include two sets of data points, with or without disk swaps. For the scenario without disk swaps, we provision enough RAM space and avoid disk swaps. For the scenario with disk swaps, we restrict the RAM space to 192 MB and disk swaps are incurred during the sorting/shuffling algorithms. In our experiments, we use ThinkSystem's SATA 6Gbps SSD and allocate 128MB of enclave protected memory. Figure 5(a) and 6(a) illustrate the runtime in relation to the input size, where each element consists of an 8-byte key and a 120-byte payload. Figure 5(b) and 6(b) examine the runtime when the input size remains constant at 100 million, while the size of each element varies from 8 bytes to about 1 KB. In all evaluation, our algorithms significantly outperform all the baselines.

Comparison with prior work. For sorting an array of 100 million 128-byte elements, depending on whether disk swaps are involved, our flex-way butterfly sort is 246× (or 2000×)[4] faster than an unoptimized recursive implementation of bitonic sort, and at least 12.4× (or 7.6×) faster than the multi-way bucket o-sort algorithm of Ramachandran and Shi [40][5]. For the same setup as above, our flex-way butterfly o-sort is 4.1× (or 4.2×) faster than an optimized version of bitonic; and at a larger data size of 1 billion with 200-bytes-wide elements, our flex-way butterfly sort is 7.2× (or 4.8×) faster than the optimized bitonic sort.

The results for oblivious shuffling is similar. For shuffling 100 million elements of 128 bytes, depending on whether disk swaps are involved, our flex-way butterfly shuffle algorithm is 5.5× (or 7.5×) faster than OrShuffle [42] and 16× (or 10.7×) faster than the multi-way bucket shuffle [40].

When all data fits in RAM, our flex-way distribution o-sort has comparable performance with our flex-way butterfly o-sort (when the input is not shuffled a priori). This is because the page-wise shuffling algorithm cannot fully randomly permute the array, which requires a larger bucket size to attain the desired security parameter — see Section A. When disk swaps are involved,

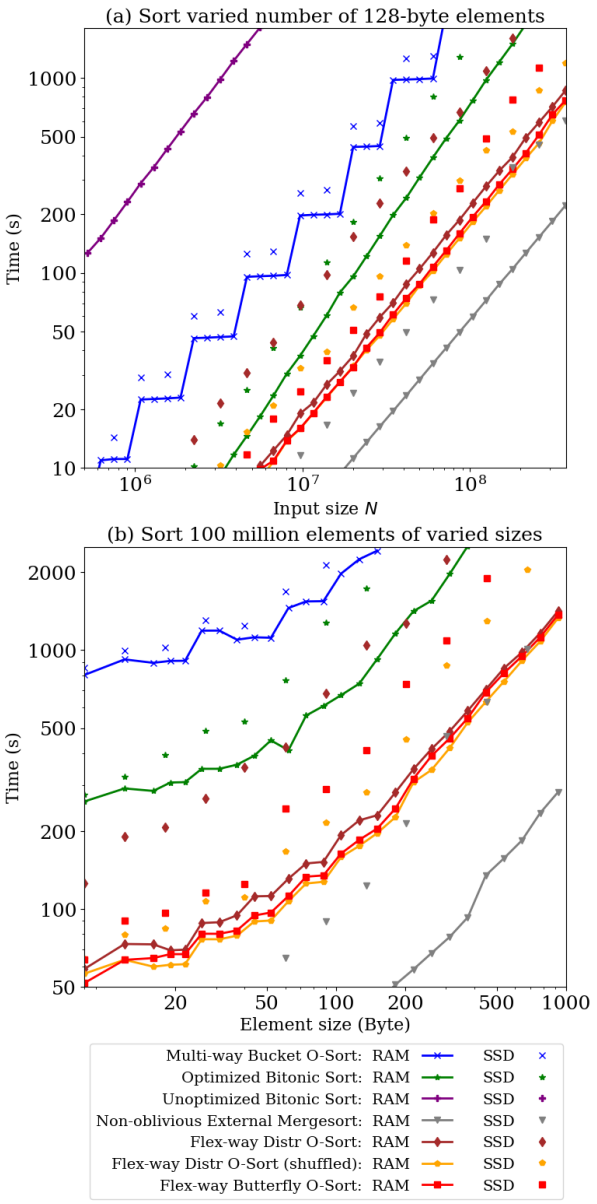


Figure 5: Comparing our sorting algorithms with prior work. The connected lines represent the case when data fits in RAM, while the scattered dots represent the case when data goes to an SSD.

4. The numbers in the parentheses are for the scenario with disk swaps.

5. Here we are comparing with a faster version of Ramachandran and Shi's algorithm [40] where the SPMS sort is replaced with a faster external-memory merge sort.

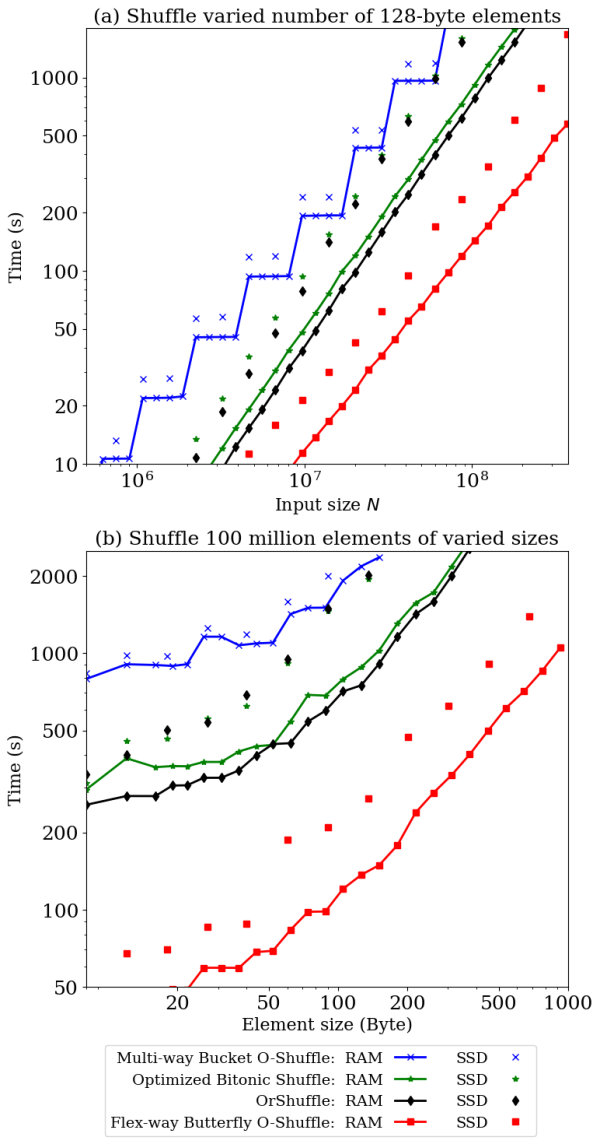


Figure 6: Comparing our shuffling algorithm with prior work.

our flex-way distribution o-sort is less performant since the initial page-wise shuffling phase requires accessing random pages on disk.[6] However, if the input array is already randomly shuffled a priori (not necessarily obliviously), our flex-way distribution o-sort algorithm slightly outperforms our flex-way butterfly o-sort — specifically, with 100 million entries each of 128-bytes (already shuffled), the performance gap is 5.5% when data fits in RAM and 25.7% when data goes to SSD.

Performance breakdown. Figure 7 provides a runtime breakdown for each algorithm, separating computation time from page swaps. In this example, we limit the available RAM space to 192 MB, so the runtime also includes cryptographic costs (the orange bar) and the overhead of `OCalls` and disk I/O (the gray bar). The runtime of the parameter solvers is insignificant and do not show up in the figure.

Figure 8 shows a micro benchmark of the multi-way MergeSplit algorithm. Our implementation based on the Interleave building block is about 18 times faster than the oblivious Bin-Packing algorithm [11] used in [40]. It is also about 2 times more efficient than an implementation using OrCompact [42].

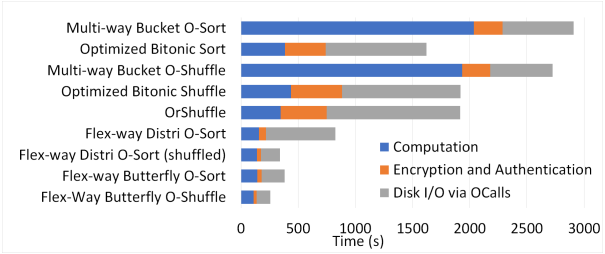


Figure 7: Breakdown of page swaps and computation time on input of 100 million 128-byte elements, using 192MB RAM and unlimited disk space.

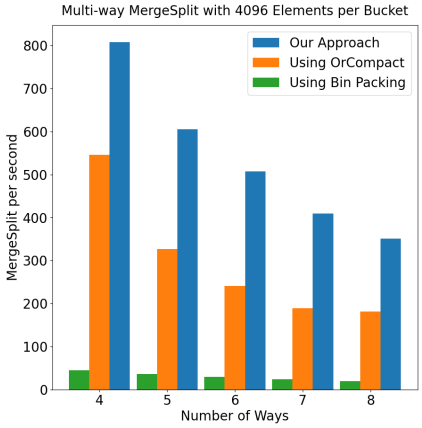


Figure 8: Throughput of our multi-way MergeSplit compared with prior methods. Each bucket contains 4096 elements of 136 bytes, including the 8-byte label.

Deferred Materials

We defer the following materials to the appendices: 1) a description of our flex-way distribution o-sort algorithm; and 2) practical optimizations including a solver for optimal concrete parameters. 3) detailed proofs for our schemes; 4) details of the Permute building block which is an improvement of Waksman's permutation network; and 5) additional background materials.

6. Despite the short seek time of SSDs, randomly reading 4 kB data blocks is still 33 times slower than sequential access in our test setup, which is mainly due to the read-ahead mechanism in the Linux file system and hardware.

References

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

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

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

[4] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *SOSA*, 2020.

[5] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi. OptORAMa: Optimal Oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2020*, 2020. To appear. See also: https://eprint.iacr.org/2018/892.

[6] R. Bardenet and O.-A. Maillard. Concentration inequalities for sampling without replacement. *Bernoulli*, 21(3), aug 2015.

[7] K. E. Batcher. Sorting networks and their applications. In *AFIPS*, 1968.

[8] M. Blanton, A. Steele, and M. Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIA CCS*, 2013.

[9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, Aug. 2017. USENIX Association.

[10] T.-H. H. Chan, Y. Guo, W.-K. Lin, and E. Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, 2018.

[11] T.-H. H. Chan and E. Shi. Circuit opram: Unifying statistically and computationally secure orams and oprams. Cryptology ePrint Archive, Paper 2016/1084, 2016. https://eprint.iacr.org/2016/1084.

[12] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Trans. Parallel Comput.*, 3(4), mar 2017.

[13] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[14] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont. Everything you should know about intel sgx performance on virtualized systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(1), mar 2019.

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

[16] A. Farhadi, M. Hajiaghayi, K. G. Larsen, and E. Shi. Lower bounds for external memory integer sorting via network coding. In *STOC*, 2019.

[17] M. Frigo, C. 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.

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

[19] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

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

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

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

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

[24] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[25] T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 61–72, New York, NY, USA, 1994. Association for Computing Machinery.

[26] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

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

[28] Intel. *Intel Software Guard Extensions (Intel® SGX) Developer Guide*, 2018.

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

[30] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *Commun. ACM*, 63(7):93–101, jun 2020.

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

[32] T. Leighton and C. Plaxton. A (fairly) simple circuit that (usually) sorts. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, volume 1, pages 264–274, 1990.

[33] C. Leiserson, H. Prokop, and K. Randall. Using de bruijn sequences to index a 1 in a computer word. 02 1970.

[34] W. Lin, E. Shi, and T. Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In *SODA*, 2019.

[35] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy*, 2015.

[36] Y. Ma. *Fault-Tolerant Sorting Networks*. PhD thesis, USA, 1995.

[37] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296, 2018.

[38] S. Patel, G. Persiano, M. Raykova, and K. Yeo. Panorama: Oblivious ram with logarithmic overhead. In *FOCS*, 2018.

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

[40] V. Ramachandran and E. 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.

[41] T. Ristenpart, E. Tromer, H. Shacham, and S. 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.

[42] S. Sasy, A. Johnson, and I. Goldberg. Fast fully oblivious compaction and shuffling. In *ACM CCS*, 2022.

[43] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[44] E. Stefanov, M. van Dijk, E. Shi, T. H. Chan, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.

[45] A. Tinoco, S. Gao, and E. Shi. Enigmap : External-memory oblivious map for secure enclaves. In *Usenix Security*, 2023.

[46] S. Tople, H. Dang, P. Saxena, and E. Chang. Permuteram: Optimizing oblivious computation for efficiency. *IACR Cryptol. ePrint Arch.*, page 885, 2017.

[47] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 991–1008, USA, 2018. USENIX Association.

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

Appendix A. External-Memory Flex-Way Distribution Oblivious Sorting Algorithm

In this section, we introduce the flex-way distribution o-sort, an alternative oblivious sorting algorithm inspired by the butterfly-random-sort algorithm presented in [40]. The flex-way distribution o-sort offers improved constants in terms of the number of page swaps. As a trade-off, it has slightly worse complexity for computation. The efficiency of the algorithm depends on the page size being reasonably small. Therefore, 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 $M = B^{\omega(1)}$.

The key distinction of flex-way distribution o-sort is that elements are assigned to buckets 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.

A.1. Algorithm Description

Load-balancing the input array using page-wise shuffling. For load-balancing purpose, our flex-way distribution o-sort algorithm requires 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 needs not 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. In our actual instantiation, instead of using a full permutation, we will actually use 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 flex-way butterfly o-shuffle algorithm on N/B indices, or using a more efficient non-oblivious variant.

One challenge is that the permutation π itself may not fit within the enclave memory, and it is too costly to read $\pi[i]$ whenever we want to access page i during the algorithm. Fortunately, observe that we actually only need to access the random permutation π in the first memory pass of the flex-way distribution o-sort algorithm, and the access to π is sequential. This means we only need to incur $|\pi|/B$ additional page swaps for reading π during the execution of the entire algorithm.

Later in Section A.2 and Lemma C.16, 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. By Lemma C.14, 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 Lemma C.14, 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, each partition should contain $R \in \Theta(\log^{3+\epsilon'} N)$ elements (see Lemma C.15). 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 flex-way butterfly network described in Section 4.1 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 bucket 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 buckets 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 4.1, 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 bucket size Z. The following invariant is maintained: an element e reaches the j-th bucket at level ℓ only if $e \in [Q_{di}, Q_{d(i+1)})$, where $i = \left\lfloor \frac{jZ}{dR} \right\rfloor$ 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 $p \in \Theta(\log \log N)$.

Figure 9 illustrates an example that assigns elements into 6 partitions through 3×2 butterfly networks.

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.

Optimizing page swaps. Same as in Section 4.1, we apply the trick of batching and "piggybacking" to optimize the number of page swaps. Since the bucket size is greater than the page size, we can always eliminate matrix transposition.

A.2. Analysis

Our flex-way distribution o-sort algorithm obliviously simulates the sorting functionality, and we state its time complexity in Theorem A.1.

Theorem A.1 (Flex-way distribution o-sort)**.** *Our flex-way distribution o-sort obliviously simulates the sorting functionality. 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.*

Proof. Deferred to Section C.6. □

Appendix B. Implementation and Practical Optimization

B.1. Solving Optimal Concrete Parameters

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

- The solver tests all feasible bucket sizes Z and selects the one that yields the minimum estimated runtime. For a fixed target failure probability, enlarging the buckets 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 buckets 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. For flex-way butterfly o-sort, we utilize the complementary cumulative distribution function of the binomial distribution to tighten the bound of failure probability. The calculation for flex-way distribution o-sort is more complicated, and we describe it in Section C.7. Correspondingly, the solver calculates the minimum number of buckets $\mathsf{NBkt} = \left\lceil \frac{N}{\lfloor Z/(1+\epsilon) \rfloor} \right\rceil$.
- 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 bucket is T_{btnc}, and the time to swap a bucket is T_{swap}. We use the following objective and constraints for flex-way

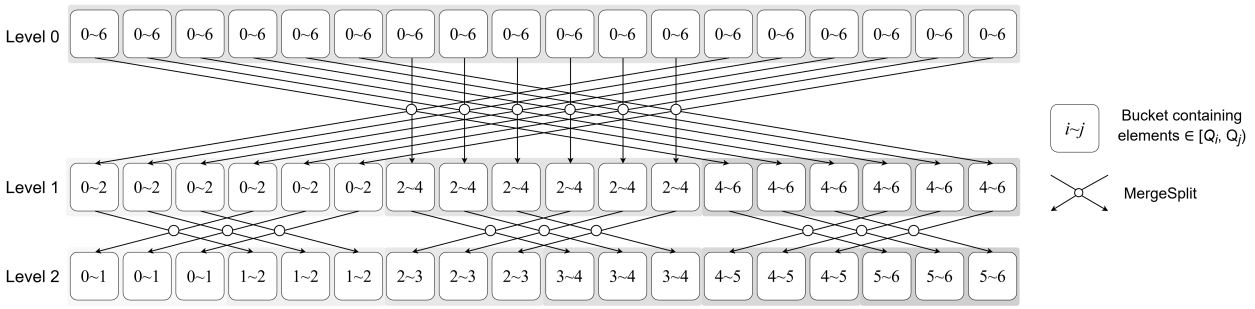


Figure 9: The butterfly network assigns elements from 18 buckets into 6 partitions based on the selected pivots. The routing can be implemented in-place as MergeSplit simply overwrites the input buckets.

butterfly o-sort and o-shuffle (the formula are similar for flex-way distribution o-sort):

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

$$\text{s.t.} \quad \mathsf{NBkt} \le \mathsf{NBkt}^* = \prod_{\ell=1}^{L} p_\ell$$

$$M/Z \ge \prod_{i=a_j+1}^{a_{j+1}} p_\ell \quad \forall 0 \le j \le n_{\mathrm{p}}$$

$$0 = a_0 < a_1 < a_2 < \ldots < a_{n_{\mathrm{p}}+1} = L$$

$$2 \le p_\ell \le \sqrt{w}, \quad p_\ell \in \mathbb{Z}, \quad 1 \le \ell \le L$$

In the formula above, L represents the number of butterfly network levels. The number of buckets after rounding up is given by $\mathsf{NBkt}^* = \prod_{\ell=1}^{L} p_\ell$, 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.

Table 2 and Table 3 lists the optimal butterfly network parameters for various input sizes under our test environment. The butterfly network is more flattened than a 2-way butterfly network, and the extra padding overhead caused by rounding is less than 2%.

B.2. Implementation Details

We offer a C++ implementation of flex-way butterfly o-sort/shuffle and distribution o-sort 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 transferring them out of the Enclave Page Cache, and to maintain the freshness of each page, a unique timestamp is always applied.

Labels and randomness. In flex-way butterfly 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 swaps. As discussed in Section E.3, each `OCall` has a startup cost due to context switching. To minimize the total cost, we combine multiple page reads or writes into a single `OCall`. For example, during the external mergesort, each sorted chunk is read in through a two-page ring buffer, where the second page is fetched lazily, allowing us to synchronize page reads from multiple sorted chunks.

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[7]. 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.

Accelerate permutation at the base case. We replace Permute with an optimal sorting network [29] at the base case of Interleave for better concrete performance.

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

TABLE 2: Example of parameters for flex-way butterfly o-sort/shuffle targeting failure probability of 2^{-60} or smaller. ϵ stands for the minimum slack factor that satisfies the failure probability requirement, and ϵ_{actual} is the actual slack factor taking rounding into account. The parameters are optimized for the scenario when each element is 128 bytes (including the sort key and payload), the EPC size is 128MB, and the RAM space is unlimited.

N	Z	ϵ	ϵ_{actual}	Butterfly Network Structure	Page Swaps
10^6	4096	0.1674	0.1796	$(8 \times 6) \times 6$	$(2+\epsilon)\frac{N}{B}$
10^7	8192	0.1158	0.1239	$(7 \times 7) \times (4 \times 7)$	
10^8	4096	0.1753	0.1796	$(5 \times 6 \times 6) \times (4 \times 5 \times 8)$	
10^9	16384	0.0839	0.1010	$(5 \times 8) \times (5 \times 8) \times (6 \times 7)$	$(3+2\epsilon)\frac{N}{B}$
10^{10}	4096	0.1852	0.208	$(2 \times 8 \times 8) \times (5 \times 6 \times 6) \times (2 \times 8 \times 8)$	

TABLE 3: Example of parameters for flex-way distribution o-sort targeting failure probability of 2^{-60} or smaller. α stands for the sampling rate, and ϵ_{actual} is the actual slack factor with rounding taken into account. In the expression of butterfly network structure, the last factor represents the number of buckets in each partition. The parameters are optimized for the setting of 128-byte wide element, 4kB page, 128MB EPC, and unlimited RAM.

N	Z	α	ϵ_{actual}	Butterfly Network Structure	Page Swaps
10^6	32768	0.02	0.278	$(3) \times 13$	$(1+\alpha)(1.5+\epsilon)\frac{N}{B}$
10^7	16384	0.01	0.5483	$(3 \times 7) \times 45$	
10^8	32768	0.01	0.4156	$(3 \times 5) \times (4 \times 4) \times 18$	$(1+\alpha)(2.5+2\epsilon)\frac{N}{B}$
10^9	32768	0.015	0.3873	$(7) \times (8) \times (6) \times (7) \times 18$	$(1+\alpha)(4.5+4\epsilon)\frac{N}{B}$
10^{10}	16384	0.015	0.6053	$(6 \times 6) \times (3 \times 6) \times (6 \times 6) \times 42$	$(1+\alpha)(3.5+3\epsilon)\frac{N}{B}$

B.3. 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 $10\% \sim 15\%$ 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 [31]. With the aforementioned cache optimization, it is essentially equivalent to the implementation of the ObliDB work [15]. We also include a non-optimized version of bitonic sort that encrypts each element separately and does not utilize a cache.
- Bitonic shuffle is implemented as described in [42]. 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 [42]. We did not implement the BORPStream algorithm introduced in [42], 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 [12] 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.

Appendix C. Deferred Proofs

C.1. Rounding Lemma

When constructing our flex-way 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

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

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

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

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

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

Moreover,

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

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

□

C.2. Additional Preliminaries

It is well-known that matrix transposition can be done with $O(N/B)$ page swaps [17]. 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 bucket.

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

Proof. We apply the cache-agnostic transposition algorithm [17], 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

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

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

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

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

The following Bernstein's inequality [6] is applied in the security proof of flex-way distribution o-sort.

Theorem C.3 (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 $\epsilon > 0$,

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

C.3. Analyzing Building Blocks

C.3.1. Fact 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 $O(n \log n)$. □

C.3.2. Claim 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 Claim 3.2 and Claim D.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. □

C.3.3. Fact 3.5: Computation cost of MergeSplit.

Proof. Preprocessing p buckets of size Z requires $O(pZ)$ numerical computation, and the final transposition makes pZ exchanges. By Fact 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. □

C.3.4. Claim 3.6: Obliviousness of Algorithm 3.

Proof. For $p \leq \sqrt{\log N}$ and $Z \leq 2^{\sqrt{\log N}}$, it only requires $p \log Z \leq \log N$ bits to store all the counts in the preprocessing step. As described in the detailed algorithm, by packing the counts into a single word, the preprocessing step can have a fixed access pattern. By Claim 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. □

C.4. Analyzing Flex-Way Butterfly O-Shuffle

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

Lemma C.4. *Our flex-way butterfly o-shuffle algorithm described in Section 4 obliviously simulates the random permutation functionality.*

Proof. The proof is similar to earlier works [4], [40]. 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 (which in term decide how we choose the other parameters including Z, ϵ and the number of ways). Therefore, it suffices to prove that the outcome of the algorithm has negligible statistical distance from a perfectly random permutation. Using the same proof as earlier works [4], [40], it suffices to prove that except with negligible (in N) probability, no bucket will receive more than Z elements. This holds as long as Z is super-logarithmic in N due to the following reasoning.

Consider a fixed bucket $A_j^{(\ell)}$ at level ℓ and index j. It can only receive real elements from P_ℓ initial buckets, 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

$$\begin{aligned} P_{\text{overflow}} &= \exp(-\Omega(\epsilon^2 Z)) \leq \exp(-\Omega(\log N \log\log N)) \\ &= N^{-\Omega(\log\log N)}. \end{aligned}$$

At each level, there are $(1+\epsilon)N/Z$ buckets, 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 buckets, we obtain the desired bound on the failure probability. □

Lemma C.5. *Given bucket size $Z = \log^c N$ where $c > 1$, flex-way butterfly o-shuffle incurs $(1 + c + o(1))N \log N$ exchanges and $O(N \log N)$ numerical computation.*

Proof. By Lemma C.1, the flex-way 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 Fact 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 $Z \in \Theta(\log^c N)$, it requires the following number of exchanges for routing:

$$\begin{aligned} &(1+o(1))N\frac{\log O(N/Z)}{\log p}(\frac{1}{2}\log Z + \log p + 1) \\ &\in (1+c+o(1))N\log N. \end{aligned}$$

Calling bitonic sort to shuffle each bucket at the last level requires $\frac{1}{4}Z\log Z(\log Z+1)$ exchanges. Since the size of each bucket is $\text{polylog}\, N$ and the label length is $\Theta(\log N)$, the probability of label collision in each bucket is $O(N^{-C})$ for some positive constant C. Therefore, we only need $1+o(1)$ trials in expectation. Since there are $\frac{(1+o(1))N}{Z}$ buckets, the expected number of exchanges to sort the buckets 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 in Section 2.2, we can determine the multiplicative inverse of all choices of the divisor $p \le \sqrt{\log N}$ 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 $O(N \log N)$. □

Corollary C.6. *Given bucket size* $Z = \Theta(\log N(\log\log N)^3)$, *flex-way butterfly 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 Lemma C.5. □

Lemma C.7. *Given that* $B \ge \log^2 N$, $M \ge B^2$ *and* $Z \in \Theta(\log N(\log\log N)^3)$, *the number of page swaps for flex-way butterfly o-shuffle is*

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

Proof. By Lemma C.1, there are $(1+o(1))N/Z$ buckets 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:

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

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

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

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

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

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

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

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 Lemma C.2. Therefore, the total number of page swaps is

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

□

Theorem C.8 (Flex-way butterfly 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, flex-way butterfly 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 Lemma C.7, 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:

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

levels. Therefore, the number of passes is:

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

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

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

The number of exchanges can be obtained by applying Lemma C.5. □

C.5. Analyzing Flex-Way Butterfly O-Sort

Lemma C.9. *Our flex-way butterfly o-sort algorithm described in Section 4 obliviously simulates the sorting functionality.*

Proof. The proof resembles earlier works [4], [40], which showed that if we apply an oblivious random permutation algorithm and then any non-oblivious,

comparison-based sort, the resulting algorithm obliviously simulates the sorting functionality. □

Lemma C.10. *Given bucket size $Z = \log^c N$ where $c > 1$, flex-way butterfly 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 [27]. 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 Lemma C.5, the expected number of exchanges required by flex-way butterfly 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 flex-way butterfly sort is still $O(N \log N)$. □

Corollary C.11. *Given bucket size $Z = \log N(\log \log N)^3$, flex-way butterfly o-sort incurs $(2.23 + o(1))N \log N$ exchanges.*

Proof. Similar to the proof of Corollary C.6. □

Lemma C.12. *Given that $M \geq B^2$ and $B \geq \log^2 N$, the number of page swaps for flex-way butterfly 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 Lemma C.7, we obtain the target expression. □

Theorem C.13 (Flex-way butterfly 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, flex-way butterfly 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.8. 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 Lemma C.10 and Lemma C.12. □

C.6. Analyzing Flex-Way Distribution O-Sort

Lemma C.14 (Number of sampled elements)**.** *For the algorithm described in Section A, 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 C.15. *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. □

Lemma C.16. *Our flex-way distribution o-sort algorithm described in Section A obliviously simulates the sorting functionality.*

Proof. Similar to Lemma C.4, 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 Lemma C.15, 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 Lemma C.15, we have that except with $\mathsf{negl}(N)$ probability, it must be that

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

Considering a bucket Bkt^* in this partition, it can only receive elements from $q/(j-i)$ buckets 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 Bkt^*, 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 Lemma C.15 happen, it holds that

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

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

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

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

By Bernstein's inequality (Theorem C.3), we have the following:

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

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

Lemma C.17 (Computational overhead of flex-way distribution o-sort)**.** *When the page size $B = \log^c N$ for $c > 0$ and the enclave size $M \in B^{\omega(1)}$, flex-way 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 [42] on each batch of size Z' incurs $O(Z' \log Z')$ exchanges. Since $Z' \in \mathrm{poly} \log N$, 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 Lemma C.1, the flex-way 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 Fact 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 $R \in \mathrm{polylog} N$, it takes $o(N \log N)$ exchanges to sort all the $\frac{(1+o(1))N}{R}$ parti-

tions. 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 upperbounded by the number of exchanges, due to the same reasoning as Lemma C.5. Therefore, the time complexity of numerical computation is $O(\frac{N\log N\log\log N}{\log\log\log N})$. □

Lemma C.18 (Number of page swaps for flex-way distribution o-sort)**.** *When the page size $B = \log^c N$ for $c > 0$ and the enclave size $M \in B^{\omega(1)}$, flex-way 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.8, 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 $o(1)$. □

C.7. Tighter Bound on Failure Probability of Flex-Way Distribution O-Sort

In this section, we present an improved method for estimating the failure probability when determining concrete parameters for our flex-way distribution o-sort. This method provides a significantly tighter upper bound compared to the Chernoff bounds used in Lemma C.15 and Lemma C.16. The key idea is to avoid setting a conservative threshold (δ) on the partition size. Instead, we measure the conditional overflow probability for each possible partition size and apply the law of total probability.

Let's consider a bucket A_{ij} in the partition $[Q_i, Q_j)$. This bucket can only receive elements from $q/(j-i)$ buckets at the initial level and contains a total of $r_{ij} \coloneqq \frac{Zq}{(1+\epsilon)(j-i)}$ real elements. We define the random variable X_{ij} to represent the size of partition $[Q_i, Q_j)$ and let $|S|$ denote the sample size. As $|S|$ is much larger than the number of partitions q, we assume that $|S|$ is a multiple of q. Since Q_i and Q_j represent the i-th and j-th quantile of the sample S, there are $s_{ij} \coloneqq \frac{(j-i)|S|}{q}$ elements sampled in $[Q_i, Q_j)$. Assume that Q_i has rank r_i, and let Y_r be a Bernoulli random variable that takes 1 when the element with rank r is sampled. Except for the first and last partitions, the probability mass function for X_{ij} is given by:

$$\begin{aligned}
&\Pr[X_{ij}=x]\\
&=\Pr[Y_{r_i+x}=1 \mid Y_{r_i}=1]\\
&\quad\cdot\Pr[\sum_{r=r_i+1}^{r_i+x-1} Y_r = s_{ij}-1 \mid Y_{r_i}=Y_{r_i+x}=1]\\
&=\frac{|S|-1}{N-1}\cdot \mathrm{pmf}_{hg}(s_{ij}-1, x-1, |S|-2, N-2)
\end{aligned}$$

where pmf_{hg} is the probability mass function of the hypergeometric distribution. The probability for the first and the last partition can be analyzed similarly.

Given $X_{ij}=x$, the probability that bucket A_{ij} overflows is given by:

$$\Pr[A_{ij}\text{ overflows} \mid X_{ij}=x] = \mathrm{sf}_{hg}(\frac{Z}{B}, \frac{r_{ij}}{B}, \frac{x}{B}, \frac{N}{B})$$

where sf_{hg} is the survival function of the hypergeometric distribution. In the formula above, we have considered the worst-case scenario where every page contains elements of consecutive ranks.

Therefore, the total probability of bucket A_{ij} overflowing is:

$$\begin{aligned}
&\Pr[A_{ij}\text{ overflows}]\\
&\quad=\sum_{x=1}^{N}(\Pr[A_{ij}\text{ overflows} \mid X_{ij}=x]\cdot\Pr[X_{ij}=x]).
\end{aligned}$$

Since calculating the exact formula above is computationally expensive, in practice, we can find an upper bound on the failure probability by considering the two ends separately and using a step size τ. Specifically, we have:

$$\begin{aligned}
&\Pr[A_{ij}\text{ overflows}] \le\\
&\sum_{\gamma=0}^{x_{\max}/\tau}(\Pr[A_{ij}\text{ overflows} \mid X_{ij}=x_{\min}+(\gamma+1)\cdot\tau]\\
&\qquad\cdot\Pr[X_{ij}=x_{\min}+\gamma\cdot\tau])\\
&+\Pr[A_{ij}\text{ overflows} \mid X_{ij}=x_{\min}]+\Pr[X_{ij}>x_{\max}]
\end{aligned}$$

where $(j-i)N/q < x_{\min} < x_{\max} < N$.

Finally, we obtain the overall failure probability by applying a union bound across all buckets on all layers.

Appendix D. Permute Using Balance

At the base case of Interleave, there are p elements with distinct keys in $\{0, 1, ..., p-1\}$, and we can apply Waksman's permutation network [48] to reorder them obliviously using no more than $p \log p$ exchanges. Specifically, for p being power of two, Waksman [48] gave an algorithm to calculate the routing plan by accessing a permutation matrix of size $p \times p$. Since $p \le \sqrt{\log N}$ and each entry of the matrix can be represented with one bit, we can pack the permutation matrix into one $\log N$-bit word and make the algorithm oblivious. Moreover, it turns out that we can also recursively apply our Balance algorithm to emulate Waksman's network for arbitrary p no more than $\sqrt{\log N}$.

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.

Parameter requirements. We require $n \le \sqrt{\log N}$ to achieve obliviousness.

Intuition: when n is a power of 2. Our permutation network structure is the same as Waksman, but we propose an algorithm that can 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 necessarily 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 $k \ge \lceil n/2 \rceil$ 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 $k \ge \lceil n/2 \rceil$ does not take up more bits than storing the original k, we can integrate the stack into the same word that stores the key.

Algorithm 4 Permute(A)

Input: The input array A contains n elements, where $n \le \sqrt{\log N}$. 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: $\quad b \leftarrow \lfloor A[i].key \,/\, m \rfloor$. Push b to $A[i].stack$.
6: $\quad A[i].key \leftarrow A[i].key \mod m$
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: $\quad$ Pop b from $A[i].stack$.
13: $\quad A[i].key \leftarrow A[i].key + b \cdot m$
14: **for** $i \leftarrow 0$ **to** $\lfloor n/2 \rfloor - 1$ **do**
15: $\quad$ Obliviously exchange $A[i]$ and $A[i + m]$. Put the one with smaller key to the front.

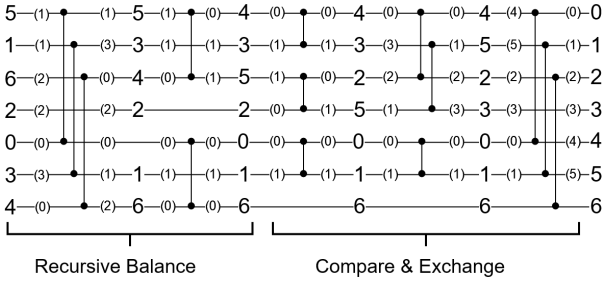


Figure 10: Permute network for $n = 7$. The solid numbers are the original keys. The numbers in the parentheses on the left of the solid numbers are the modified keys used in the previous Balance operation, and the numbers in the parentheses on the right of the solid numbers are the modified keys used in the next Balance operation.

Example. For example, Figure 10 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 [29].

Fact D.1 (Computation overhead of Permute)**.** *The* Permute *algorithm requires $O(n \log n)$ 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 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. □

Claim D.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 Claim 3.2, the access patterns of the subprocedure 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. □

Appendix E. Additional Background

E.1. Oblivious Algorithms

Access patterns. In the external-memory model, we assume that the adversary can observe both memory access patterns within the enclave and the page swap patterns. In particular, a malicious operating system can observe the page swap patterns for a hardware enclave setting. A cache-timing adversary can learn the access patterns within the enclave [24], [26].

Oblivious simulation. We define a general notion of oblivious simulation like in [4], which works for randomized functionalities too. Note that because of our definition of access patterns, our notion of oblivious simulation essentially achieves strong obliviousness [34] or double obliviousness [37], that is, both the accesses within the enclave and the page swap patterns do not leak any sensitive information.

Intuitively, a program F obliviously simulates a possibly randomized functionality f if: (1) F exhibits the same input/output behavior as f ; (2) Second, there exists a simulator $\mathsf{Sim}(|x|)$ that, without knowledge of the input x, generates memory access and page swap patterns that are statistically indistinguishable from those produced by F. In case the patterns and the functionality are randomized, we have to consider the joint distribution of the simulator and the output of the functionality.

For a program F and input x, let $\mathsf{AccPtrn}(F, x)$ denote the distribution of memory addresses and page swaps F produces on an input x.

Definition E.1 (Oblivious simulation [4])**.** A program F obliviously implements the functionality f iff there exists a simulator Sim and a negligible function $\nu(\cdot)$ such that the following holds:

$$\{\mathsf{Sim}(1^\lambda), f(x)\}_{x \in \{0,1\}^\lambda} \overset{\nu(\lambda)}{\equiv} \{\mathsf{AccPtrn}(F, x), F(x)\}_{x \in \{0,1\}^\lambda}$$

where $\overset{\nu(\lambda)}{\equiv}$ means that the two sides have statistical difference at most $\nu(\lambda)$.

E.2. Constant-time Operations on Memory Words

Given the basic operations described in Section 2.2.1, we can also build the following operations on memory words in constant time:

- $\mathsf{Negate}(W)$. Negate all bits in W, which is equivalent to XOR a bit array filled with 1s.
- $\mathsf{Extract}(W, a, b)$. Extract from W the bits between offset a and b, which can be achieved with a left shift followed by a logical right shift.
- $\mathsf{Set}(W, a, b, V)$. Set bits of W between offset a and b to be the lowest $b - a + 1$ bits of V. The operation can be done by Extract bits from W and V and concatenate them using left shifts and XORs.
- $\mathsf{LSB}(W)$. Find the offset of the least significant 1 bit in W. Whereas LSB is equivalent to a single "count-trailing-zeros" instruction on many modern CPUs, we can also derive it from the previous primitives if the length of W is no more than $\sqrt{\log N}$. First, we can get the mask for the least significant 1 bit using the formula W AND $(\mathsf{Negate}(W)+1)$. Then, as shown in [33], the bit can be indexed with a multiplication and a look-up in a small table. Although the table look-up is not oblivious in general, due to the limited length of W, the table only contains $\lfloor \sqrt{\log N} \rfloor$ entries and the length of each entry is at most $\lceil \frac{1}{2} \log \log N \rceil$ bits. Therefore, the table can be packed into a single word and looked up obliviously via Extract.

We also construct some constant time operations on unsigned integers using the primitives.

- $\mathsf{Subtract}(X, Y)$. Subtract Y from X, which is equivalent to $X + \mathsf{Negate}(Y) + 1$.
- $\mathsf{IsLess}(X, Y)$. Return if X is less than Y, equivalent to Extract the highest bit of $\mathsf{Subtract}(X, Y)$.
- $\mathsf{DivConst}(X, C)$. Divide X by C, where C is an a-priori known constant. As shown in [25], this stems from multiplication combined with a right shift.
- $\mathsf{ModConst}(X, C)$. Return X modulo C, where C is an a-priori known constant. This can be expressed as $\mathsf{Subtract}(X, C * \mathsf{DivConst}(X, C))$.

E.3. SGX Enclave and External-Memory Model

Transfer of data across the boundary of SGX enclave. Secure processors, like those incorporating Intel Software Guard Extensions (SGX), offer a hardware-based secure environment known as the enclave. The enclave ensures sensitive data to be decrypted and processed only within its protected boundaries. To facilitate this, SGX introduces a dedicated secure memory region called the Enclave Page Cache (EPC). In SGX v1, the EPC memory size is limited to 128MB, while in SGX v2, it typically ranges from 8GB to 512GB.

In practice, it is common for the size of the data to exceed the available EPC memory or even the physical RAM capacity. Consequently, it becomes necessary to store the data in an encrypted form in insecure physical memory or external storage. Given our threat model assumes a potentially malicious operating system, it is imperative to also ensure the authentication of the data.

There are typically three approaches to transfer data in and out of the enclave.

- Use built-in swapping instructions (e.g., EWB) that provide cryptographic data protection.
  Previous research has demonstrated that these built-in instructions involve a complex page eviction process, resulting in significant performance overheads [14], [45]. Therefore, this approach is not employed in our implementations.
- Use third-party cryptographic packages and copy data to/from untrusted memory directly.
  Alternatively, we can efficiently encrypt and authenticate data within the enclave using a third-party cryptographic package, leveraging hardware acceleration like Intel's Advanced Encryption Standard Instructions (AES-NI). The encrypted data can be directly transferred between the EPC and non-EPC memory through a unified virtual address space of the enclave's host process [13].
- Use third-party cryptographic packages and run applications outside the enclave via `OCall`.
  When data exceeds the physical RAM capacity, external storage such as a disk file system needs to be accessed. Such operation typically involves system calls, which are prohibited inside the enclave. To address this, we adopt the `OCall` mechanism to execute untrusted applications outside the enclave [28]. Again, the data must be encrypted and authenticated before transferred to the untrusted applications.

Motivation of external-memory model. Due to the startup cost of `OCall`, disk swap, and cryptographic operations, data should be transferred in a granularity of at least 4KB [45], which resembles the paging mechanism in operating systems. Nevertheless, the benchmark study in [45] demonstrates that transferring 4KB of data across the enclave boundary results in up to $66\times$ higher overhead compared to moving it within the enclave. Consequently, an algorithm designed for enclave should exploit data locality, ensuring maximum utilization of a fetched page before swapping it out.

Therefore, the external-memory model [2] aligns perfectly with hardware enclaves, as emphasized by [45]. The external-memory model can be seen as an enhancement of the traditional RAM model. While the conventional RAM model primarily measures the number of instructions, the external-memory model additionally considers factors of cache misses and page swaps, which are typically more resource-intensive.

Notations. We use the following notations globally:

- The *page size* B is the maximum number of elements that fit in a page.
- The *enclave size* M is the maximum number of elements that fit in the enclave's protected memory.

Note that we define B and M in terms of the number of elements rather than the number of bytes — the definition disassociates our asymptotic bounds with the length of the payload string.

By convention, the cost of a page swap includes both reading a page into the enclave and writing a page back to the external memory. In practice, however, page reads and page writes do not necessarily occur together. To account for this, we consider a single page read or page write as equivalent to 0.5 page swaps, as they have approximately the same overhead in our experiments.

E.4. Additional Overview of Previous Works

Deterministic sorting networks. Although AKS and zig-zag sorting networks [3], [22] are well-known for achieving a size of $O(N \log N)$, their construction of expander graphs results in an astronomical constant. In practice, Batcher's odd-even mergesort and bitonic sort [7] offer faster alternatives, but suffer from suboptimal asymptotic complexity.

Probabilistic sorting networks. Leighton and Plaxton [32] proved that there is a network of depth $7.44 \log N$ that succeeds on random input except with negligible probability. However, they did not provide

an explicit network construction, and for practical input sizes, the constant must substantially increase to ensure a reasonable success probability [32]. Although subsequent work [36] provided an empirical construction that outperforms Batcher's sorting network, no proof was presented, and it is challenging to construct the circuit for input sizes exceeding 2^{14}.

Randomized Shell sort. Randomized Shell sort [21] implements oblivious sorting using $24N \log N$ compare-and-exchanges. However, its access pattern exhibits poor locality, leading to high overhead from page swapping. Additionally, the algorithm has a non-negligible failure probability.

Goodrich's oblivious external-memory sorting. The sorting algorithm in [20] attains optimal complexity for page swaps but incurs significant computational overhead when strong obliviousness is required.