Waks-On/Waks-Off: Fast Oblivious Offline/Online Shuffling and Sorting with Waksman Networks[*]

Sajin Sasy
University of Waterloo
Waterloo, ON, Canada
ssasy@uwaterloo.ca

Aaron Johnson
U.S. Naval Research Laboratory
Washington, D.C., U.S.A.
aaron.m.johnson@nrl.navy.mil

Ian Goldberg
University of Waterloo
Waterloo, ON, Canada
iang@uwaterloo.ca

ABSTRACT

As more privacy-preserving solutions leverage trusted execution environments (TEEs) like Intel SGX, it becomes pertinent that these solutions can by design thwart TEE side-channel attacks that research has brought to light. In particular, such solutions need to be fully oblivious to circumvent leaking private information through memory or timing side channels.

In this work, we present fast fully oblivious algorithms for shuffling and sorting data. Oblivious shuffling and sorting are two fundamental primitives that are frequently used for permuting data in privacy-preserving solutions. We present novel oblivious shuffling and sorting algorithms in the offline/online model such that the bulk of the computation can be done in an offline phase that is *independent of the data to be permuted.* The resulting online phase provides performance improvements over state-of-the-art oblivious shuffling and sorting algorithms both asymptotically ($O(\beta n \log n)$ vs. $O(\beta n \log^2 n)$) and concretely ($> 5\times$ and $> 3\times$ speedups), when permuting n items each of size β.

Our work revisits Waksman networks, and it uses the key observation that setting the control bits of a Waksman network for a uniformly random shuffle is independent of the data to be shuffled. However, setting the control bits of a Waksman network efficiently and fully obliviously poses a challenge, and we provide a novel algorithm to this end. The total costs (inclusive of offline computation) of our WaksShuffle shuffling algorithm and our WaksSort sorting algorithm are lower than all other fully oblivious shuffling and sorting algorithms when the items are at least moderately sized (i.e., $\beta > 1400\,\mathrm{B}$), and the performance gap only widens as the item sizes increase. Furthermore, WaksShuffle improves the online cost of oblivious shuffling by $> 5\times$ for shuffling 2^{20} items of any size; similarly WaksShuffle+QS, our other sorting algorithm, provides $> 2.7\times$ speedups in the online cost of oblivious sorting.

1 INTRODUCTION

Over the last few years we have witnessed myriad privacy-preserving solutions propose Trusted Execution Environments (TEEs) to realize hitherto impractical systems. These proposals span several application domains: healthcare and genomic data analytics [9], contact discovery [30, 46], telemetry [8], collaborative machine learning [34], privacy-preserving statistics [44], file systems [14], database queries [23], and video analytics [36], to highlight a few. The advantage of TEEs is that they enable collaborative solutions between mutually untrusting parties, without having to i) rely on trusted third parties, ii) rely on non-collusion of servers, or iii) incur the prohibitive overheads of fully homomorphic encryption.

In theory, TEEs enable parties to execute computations over their private data on an untrusted remote server within an *enclave* (secure container). All parties can verify the integrity of this computation via an attestation process [2], as well as send their encrypted private data directly to this enclave, such that each party's private data is only ever decrypted and used for the agreed-upon computation. TEEs guarantee to each participant that the computation is correctly and privately (i.e., without leaking any information about the private inputs) executed within the enclave.

Unfortunately, research has shown that TEEs are susceptible to several side-channel attacks limiting privacy in practice. Some side channels violate the SGX security model, including those based on speculative execution [10, 47] and voltage variations [29, 31], and Intel has issued patches to mitigate them [20–22]. In contrast, memory and control-flow side-channel attacks [24, 27, 49] are out of scope of the SGX security model.

However, some defenses have been developed against such side channels [39, 40]. Sasy et al. [41] suggest using *fully oblivious* algorithms to circumvent all currently known software (memory and code-branching based) side-channel attacks. (We provide a more detailed background on full obliviousness in Section 2.) They provide fully oblivious algorithms for compaction and shuffling. The shuffling algorithms they present are ORShuffle and BORPStream, where the former consistently outperforms BitonicShuffle[1] (the prior state-of-the-art method for oblivious shuffling), and the latter reduces the online cost of shuffling items by partially shuffling them as they arrive in a "streaming" setting.

In the previously mentioned applications of TEEs, shuffling or sorting of data is a recurring primitive. Such data permutation steps must be fully oblivious lest they leak information about the private inputs or permutations. Furthermore, in these systems, the permutation operations reside on the critical path of execution, and so their execution speed is critical for system performance.

All the TEE systems we have cited may obtain a performance improvement from algorithms designed for the *offline/online* model, in which an offline computation phase is performed before the data is available, followed by an online phase that uses the input data. In general, the offline/online model can be useful to reduce query latency for a system that answers queries on existing data [14, 44]. It can also be used to reduce analysis time for systems that otherwise idly wait while collecting (a predetermined amount of) data from users [8]. For systems with repeated batch processing, it can also enable parallelism, as an offline phase can be performed concurrently with the current online phase.

[*] This is an extended version of our CCS 2023 paper. [43]

[1] BitonicShuffle shuffles items by attaching random labels to the items and sorting them with the Bitonic sorting network [4].

Therefore, in this work we present novel fully oblivious shuffling and sorting algorithms in the offline/online setting. Full obliviousness ensures that their inputs and outputs are kept private when they are executed entirely within a TEE enclave. Our algorithms move the bulk of the work for oblivious shuffling and sorting into an offline precomputation that is *independent of the data to be permuted*, resulting in better asymptotic and concrete online costs than prior work. For instance, shuffling 2^{20} items, each of 256 bytes, takes 0.50 s of online computation, a ≈5× speedup over ORShuffle and ≈2.7× speedup over BORPStream's online cost. Similarly, sorting the same set takes 1.13 s of online computation, a 3.1× speedup over Bitonic sort and 2.1× speedup over the online cost of a BORPStream-based sort. We note that these comparisons are generous towards BORPStream, in that they assume its streaming model in which each data item can be partially processed in sequence before the last item arrives. The performance of BORPStream is otherwise worse than ORShuffle, and our shuffling algorithm provides a >11× speedup over it.

Furthermore, even if one also takes into consideration the offline costs of our algorithms, our *total cost* for shuffling and sorting still outperforms all the aforementioned algorithms when the items are at least moderately sized (i.e., larger than about 1400 bytes). Applications permuting items of this size include ML training [34], where the items to shuffle may be images, and database queries [23], where the items to sort may be entire rows of database tables. Therefore, even without the offline/online split, our algorithms are useful for several applications, and the total costs are modest even for compute-limited servers.

We achieve these performance improvements using Waksman networks [48]. A Waksman network can apply any permutation to a set of items by sending them through a fixed-topology network of switches. Each switch has two inputs, two outputs, and a control bit; the outputs are the inputs in swapped order if the control bit is one and in the same order otherwise. For a set of n input items, there are approximately $n \log_2(n) - n + 1$ switches (and exactly this number when n is a power of two). Obliviously setting the control bits to implement a given permutation poses a challenge, but this process is agnostic to the size of items to permute.

The key observation we make is that to uniformly randomly shuffle items, one can generate the control bits for the Waksman networks *independently of the items to shuffle*. We hence reduce the online cost of shuffling to the $O(\beta n \log n)$ cost of permuting n items each of size β with a Waksman network. Further, the parallel step complexity is only $O(\log n)$. In contrast, existing oblivious shuffling algorithms like BitonicShuffle, ORShuffle, and BORPStream incur a $O(\beta n \log^2 n)$ online complexity, with a parallel step complexity of $O(\log^2 n)$. We provide detailed background on Waksman networks in Section 2. Finally, while we focus on shuffling and sorting, by definition Waksman networks can be used to apply any permutation, and in scenarios where several sets of data need to be permuted in the same order the cost of setting control bits can be amortized over these sets.

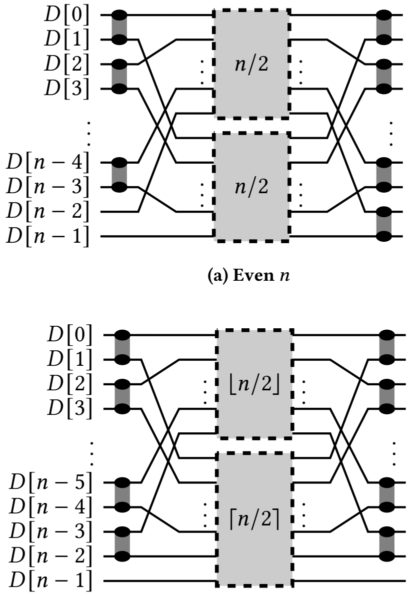
For sorting data, the control bits for a Waksman network would depend on the items to be sorted; i.e., one needs to generate the permutation that results in a sorted output, and then set the control bits accordingly. However, by using the "Scramble-then-Compute" paradigm [13, 18], one can reduce the online cost of sorting data as well. A Waksman network can be used to perform an oblivious shuffle, and then this shuffled data can be sorted using a fast non-oblivious sort such as quicksort. This algorithm has an online cost of $O(\beta n \log n)$ and outperforms bitonic sorting networks, which have a $O(\beta n \log^2 n)$ online cost, and it similarly outperforms ORShuffle or BORPStream (both having a $O(\beta n \log^2 n)$ complexity) followed by a non-oblivious sort. We summarize our contributions as follows:

(1) We provide a novel algorithm to set the control bits of a Waksman network to implement a given permutation, designed to be fully oblivious for TEEs. Our algorithm is the first fully oblivious control bit setting algorithm for Waksman networks generalized to work for any n; i.e., even when n is not a power of two.
(2) We provide a Waksman network topology that improves performance by exploiting memory locality. Our experimental results show up to 5.3× speedups over the standard Waksman layout [48].
(3) The total costs (inclusive of offline computation) of our algorithms are lower than the state of the art when the items are at least moderately sized (i.e., item size β > 1400 B). Furthermore, the performance improvements of our algorithms continue to increase as the size of each item increases. For instance, our algorithms yield >2.2× improvements in total cost when shuffling or sorting 4 KiB sized items.
(4) Finally, in the offline/online model, we significantly reduce the online costs of shuffling and sorting below those of the state of the art (achieving speedups of >5× and >2.7×, respectively). These reductions are obtained for *any* item size, improving the performance of a wide variety of TEE applications.

2 BACKGROUND

Waksman networks. A Waksman network [48] is an arrangement of binary *switches*. Each switch has two inputs and two outputs, and its behavior is governed by a control bit. If the control bit is set (i.e., equal to one), then the switch swaps its inputs. If the bit is unset (i.e., equal to zero), then the switch outputs its inputs in their original order. A Waksman network can apply any permutation to its inputs by setting its control bits appropriately. Setting a Waksman network to a uniformly random permutation is not as simple as setting its control bits independently and uniformly at random because the number of such settings is a power of two and thus is not evenly divided by the number of permutations when the number of inputs is more than two.

The Waksman network construction [5] is illustrated recursively in Figure 1. The construction is slightly different for even n (Figure 1a) and odd n (Figure 1b). In both cases, a layer of *input switches* is first applied, the results are used as inputs to *top* and *bottom* Waksman *subnetworks*, and then a layer of *output switches* are applied to the outputs of those subnetworks. The switches in a given layer take disjoint inputs, and so they can be applied in parallel. When $n = 2$, the network is a single switch, and when $n = 1$ there are no switches. Waksman networks are designed to minimize the number of switches while still being able to apply any permutation. The number of switches is exactly $n \log_2(n) - n + 1$ for n a power



(a) Even n

(b) Odd n

Figure 1: Existing recursive construction for Waksman network of n inputs [5]. Crossbars denote switches. Switches are applied left to right on input array D. Dashed rectangles are Waksman subnetworks with the given numbers of inputs. Note that inputs to subnetworks are not contiguous in D.

of two and at most $n \log_2(n) - 0.91n + 1$ for any n [5]. Observe that the topology of the Waksman network is fixed; that is, which switches are applied to which values in what order depends only on n. This property makes Waksman networks useful for oblivious computation.

Waksman [48] describes a $O(n \log n)$ algorithm to set the control bits to implement a given permutation. In this "looping" algorithm, the input and output switches are alternately set, as setting one determines the required setting for a following one, and then the algorithm is recursively called on the subnetworks. Nassimi and Sahni [32] give a more parallelizable algorithm to set control bits. It has runtime $O(n \log^2(n))$ but parallel step complexity of $O(\log^2(n))$, improving on the $O(n)$ parallel step complexity of the Waksman algorithm. Bernstein [6] describes how to perform the Nassimi-Sahni algorithm obliviously. Using an oblivious sort with runtime $O(n \log^2(n))$, which is the fastest in practice [15], the Bernstein algorithm has runtime $O(n \log^4(n))$.

Lee [25, 26] presents a different approach to setting the Waksman control bits. In the Lee algorithm, the input switches are set first, then the subnetwork switches, and finally the output switches. The algorithm is not oblivious and has the same $O(n \log n)$ runtime as the looping algorithm. However, an advantage of the Lee algorithm in the context of oblivious computation is that setting the output switches is trivially and efficiently oblivious. We therefore use the Lee algorithm as our starting point in designing a fully oblivious control bit setting algorithm.

Full obliviousness. Oblivious algorithms are needed for secure computation on existing TEEs, which have side channels that can leak properties of an execution. Shared memory resources such as caches can reveal which memory locations have been accessed, and execution time can reveal information about which execution paths were followed. A goal of TEEs is to provide confidentiality for inputs and outputs, but obliviousness is needed to protect against an adversary exploiting these side channels.

Among the variety of notions of obliviousness [23, 41, 46], we adopt full obliviousness [41] as our security goal. Full obliviousness requires both the memory accesses (i.e., the location and read/write operation of each access) and the executed sequence of instructions to be independent of the secret inputs. Consequently, our algorithms will not require any "private" memory, where accesses are hidden from the adversary. Weaker obliviousness notions omit the instructions or only require memory obliviousness at a coarser granularity, such as page-level obliviousness. However, those notions are vulnerable to side channels of the instructions sequence or operating at a finer granularity, such as cache-line-level attacks. Such side channels do already exist [24], and, given that they are tolerated by the hardware designers, future hardware designs may further create new ones.

By using full obliviousness, we prevent entire classes of side-channel vulnerabilities. We note, however, this security notion does not entirely rule out timing side channels because some processor instructions may be variable time. In our implementation, we do choose instructions that we expect to be constant time, but this property is not always explicitly promised by the hardware provider. Moreover, existing compilers do not guarantee to preserve obliviousness, although for our implementation, we check the compiled binary for obliviousness violations using the FOAV tool (see Section 6).

We follow the definition of full obliviousness given by Sasy et al. [41]. Let an algorithm A be a sequence of instructions with direct or indirect memory arguments. Let $\mathcal{E}(A, x_1, \ldots, x_k)$ denote the execution of A on inputs $(x_1, \ldots, x_k)$, which consists of the sequence of executed instructions and the memory contents at every step. A is assumed to have access to a sequence of uniformly random bytes, and therefore $\mathcal{E}$ is a random function. Let $E = \mathcal{E}(A, x_1, \ldots, x_k)$ be a random execution. We denote its output as $\mathcal{O}(E)$. We denote its instruction trace as $\mathcal{I}(E)$, which consists of the sequence of indices into A indicating the order instructions were executed (some instructions may change the execution flow and make it non-sequential). We denote the memory trace of E as $\mathcal{M}(E)$, which consists of a sequence of pairs (b, ℓ), one for each executed instruction, with $b \in \{0, 1\}$ indicating a read or write and ℓ indicating the memory location. We say that an algorithm is *efficient* if it runs in probabilistic polynomial time, and we say that two random variables are *indistinguishable* if no efficient algorithm can distinguish them with non-negligible probability.

Full obliviousness is defined as follows [41], with $[k] = \{1, \ldots, k\}$:

Definition 1. Let $E = \mathcal{E}(A, x_1, \ldots, x_k)$. For any $I \subseteq [k]$, A is *fully oblivious* with respect to $\{x_i\}_{i \in I}$ if there exists an efficient simulator $\mathcal{S}$ such that, for all inputs $(x_1, \ldots, x_k)$, $(\mathcal{O}(E), (\mathcal{M}(E), \mathcal{I}(E)))$ is indistinguishable from $\left(\mathcal{O}(E), \mathcal{S}\left((x_i)_{i \in [k] \setminus I}, (|x_i|)_{i \in I}\right)\right)$.

The inputs indicated by I in Definition 1 are *secret*, and the others are *non-secret*. Observe that the definition does not require the size of any input to be hidden. Also note that the definition requires indistinguishability of the joint distribution over outputs and traces, even though the simulator is not given the output. Therefore, no information is revealed about either the secret inputs or the output beyond what is implied by the non-secret inputs.

3 OBLIVIOUS ALGORITHMS FOR WAKSMAN NETWORKS

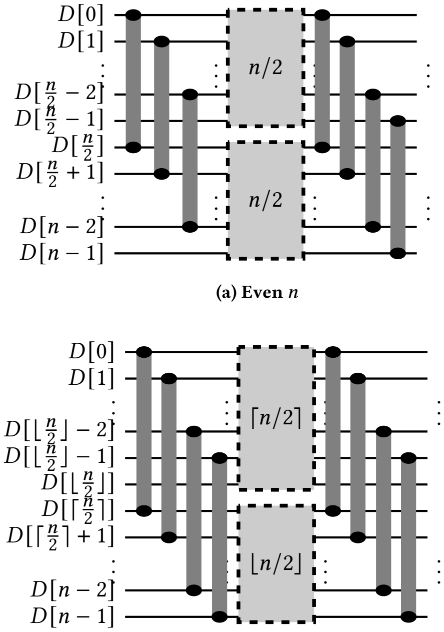
3.1 Preliminaries

When describing algorithms, we use $A[i]$ to indicate the element of array A at index i, with $A[0]$ as the first element. We use $A[i..j]$ to indicate the subarray $(A[i], A[i+1], \dots A[j])$, and we use $|A|$ to denote the number of items in A. We denote bitwise AND with &, bitwise OR with |, and the left (right) shift of x by y bits with $x \ll y$ ($x \gg y$). We assume a global security parameter λ that is available to all algorithms, and we do not require that any algorithm is oblivious to it.

We also make use of several fully oblivious primitives. We assume that random bits can be generated fully obliviously. We assume that arithmetic and bitwise operators are oblivious to their inputs. We denote the oblivious computation of the comparison c as a boolean value with $[\![c]\!]$ (e.g., $[\![x < y]\!]$ has value 1 if $x < y$ and 0 otherwise). We use an oblivious select function, OSelect(x, y, b), which returns y if $b = 1$ and returns x otherwise. OSelect is assumed to be oblivious to its input values. We also use an oblivious swap function, OSwap(A, i, j, b), which swaps the values of $A[i]$ and $A[j]$ if $b = 1$ and does nothing otherwise. OSwap(A, i, j, b) is assumed to be oblivious to A and b (though not to i and j) and, for items of size β, is assumed to have runtime $O(\beta)$. These primitives can be implemented efficiently and obliviously using boolean arithmetic or conditional instructions such as `CMOVZ`.

Some additional functionalities we use require other approaches to provide the necessary obliviousness. We use ORand(x) to select a uniformly random integer in $[0, x-1]$, which we assume is fully oblivious with respect to its input. One efficient oblivious implementation with statistical distance $x2^{-\sigma}$ from uniform is to select σ random bits ρ and compute $x\rho \gg \sigma$. We also require a pseudorandom permutation, PRP(x, k), that applies a PRP to x using key k and is fully oblivious to its inputs. A securely implemented PRP (e.g., AES) must have a constant-time implementation and would thereby provide the required obliviousness. We additionally use a hash map data structure with $O(1)$ amortized insertion and lookup. Given hash map H, HashMapInsert(H, k, v) inserts value v with key k, and HashMapGet(H, k) returns the value inserted for key k. We require that, for any sequence of insertions and lookups, the resulting memory and instruction traces can be simulated without knowing the values v (but knowing the keys k). The standard hash map algorithms are already oblivious in this way.

Although a goal of our algorithms is to provide oblivious shuffling and sorting, we do rely on existing fully oblivious shuffling and sorting algorithms. Our algorithms will provide efficiency improvements over applying those algorithms directly to the data. OShuffle(A) is assumed to put the items in array A in a uniformly random order. OSort(A, B) is assumed to sort array A in place in nondecreasing order and also put B in the same final order as A (e.g., OSort could be used to sort data items in B using keys in A). Both OShuffle and OSort are assumed to be fully oblivious to all of their inputs.


$D[0]$
$D[1]$
$D[\frac{n}{2}-2]$
$D[\frac{n}{2}-1]$
$D[\frac{n}{2}]$
$D[\frac{n}{2}+1]$
$D[n-2]$
$D[n-1]$
$n/2$
$n/2$
$D[\lfloor\frac{n}{2}\rfloor-2]$
$D[\lfloor\frac{n}{2}\rfloor-1]$
$D[\lfloor\frac{n}{2}\rfloor]$
$D[\lceil\frac{n}{2}\rceil]$
$D[\lceil\frac{n}{2}\rceil+1]$
$\lceil n/2\rceil$
$\lfloor n/2\rfloor$

(a) Even n

(b) Odd n

Figure 2: Our recursive construction for Waksman networks of n inputs. Crossbars denote switches. Switches are applied left to right on input array D. Dashed rectangles are Waksman subnetworks for the given numbers of inputs. Note that inputs to the subnetworks are contiguous in D.

3.2 Our Waksman Topology

Our algorithms make use of a modified Waksman topology, which is shown in Figure 2 for a network with n inputs. As with the standard topology (Figure 1), we use a recursive construction with slight differences for even n (Figure 2a) and odd n (Figure 2b). The base cases remain the same: the network is a single switch for $n = 2$, and there are no switches for $n = 1$. In our topology, however, the input switches take inputs separated by $\lceil n/2\rceil$ rather than by 1, and output switches yield outputs separated by $\lceil n/2\rceil$ rather than by 1. The advantage of this topology in our setting is that, after applying the input switches, the inputs to each subnetwork are contiguous in memory rather than fragmented, whereas in the standard topology there is fragmentation that increases with each subnetwork. This design improves the memory locality and thus performance for both setting control bits and applying them to a data array. This topology uses the same number of switches as the standard one. See Section 3.4 for a complete example in the $n = 9$ case.

Figure 3: ControlBits(P, d): Compute Waksman control bits for permutation P at recursion level d. i maps to a value encoded in $P[i]$. Initially call with $d = 0$. Modifies P. Oblivious to values in P but not to $|P|$ or d.

```
1:  $n \leftarrow |P|$
2:  $k \leftarrow \lceil n/2 \rceil$
3:  if $n < 2$ then
4:      return $\emptyset$
5:  ▹ Compute control bits for input-layer switches
6:  $C_{\text{in}} \leftarrow$ InBits$(P, d)$
7:  ▹ Apply input-layer switches to P
8:  if $n > 2$ then
9:      for $i \leftarrow 0, \ldots, k-2$ do
10:         OSwap$(P, i, k+i, C_{\text{in}}[i])$
11: ▹ Reduce P values modulo k, storing bits to undo later
12: for $i \leftarrow 0, \ldots, n-1$ do
13:     $b \leftarrow \llbracket (P[i] \gg d) \geq k \rrbracket$
14:     $P[i] \leftarrow ((P[i] - (\text{OSelect}(0, k, b) \ll d)) \ll 1) \mid b$
15: ▹ Compute control bits for top and bottom subnetworks
16: if $n > 2$ then
17:     $C_{\text{top}} \leftarrow$ ControlBits$(P[0..k-1], d+1)$
18:     $C_{\text{bot}} \leftarrow$ ControlBits$(P[k..n-1], d+1)$
19: ▹ Compute control bits for output-layer switches
20: Create array $C_{\text{out}}$ of length $n-k$.
21: for $i \leftarrow 0, \ldots, n-k-1$ do
22:     $C_{\text{out}}[i] \leftarrow P[i]\ \&\ 1$        ▹Control bit is the last bit of P[i]
23: ▹ Apply output-layer switches to P
24: for $i \leftarrow 0, \ldots, n-k-1$ do
25:     OSwap$(P, i, k+i, C_{\text{out}}[i])$
26: ▹ Undo reduction of P values modulo k
27: for $i \leftarrow 0, \ldots, n-1$ do
28:     $b \leftarrow P[i]\ \&\ 1$
29:     $P[i] \leftarrow (P[i] \gg 1) + (\text{OSelect}(0, k, b) \ll d)$
30: return $(C_{\text{in}}, C_{\text{top}}, C_{\text{bot}}, C_{\text{out}})$
```

3.3 Setting Control Bits

Our fully oblivious algorithm ControlBits(P, d) sets the Waksman control bits to implement a permutation, and it is shown in Figure 3. For $n = |P|$, P must be an arrangement of $\{0, \ldots, n-1\}$, with $P[i] = j$ meaning that i is mapped to j. The initial call should be with a recursion depth of $d = 0$. ControlBits is fully oblivious to P but not n or d. ControlBits is similar to the Lee algorithm [26] for setting control bits. The main differences are (1) the switches follow our modified topology, (2) the algorithm explicitly handles array sizes that are not a power of two, and (3) crucially, setting input switches is modified to make the process oblivious (setting output switches is already oblivious).

The Lee algorithm is described for permutation sizes that are powers of two. First, input switches are set and are applied to their input permutation values, then the subnetwork switches are set recursively, and finally the output switches are set and are applied to their permutation inputs. In the ith subnetwork level (starting at $i = 0$ and incrementing with each subnetwork), the input switches are set so that applying them to the input subarray of P yields as input to each subnetwork a *Complete Residue System modulo* $n/2^{i+1}$, which is a set containing one representative from each residue class modulo $n/2^{i+1}$. That is, if $\hat{P}$ is the subarray of P used as input to a subnetwork, the input switches are set to make $\{x \bmod n/2^{i+1}\}_{x \in \hat{P}} = \{0, \ldots, n/2^{i+1} - 1\}$. After recursively setting the subnetwork switches (which also applies them to elements of P), each output switch is set to the $(\log_2 n - i - 1)$st bit of its input from the top subnetwork, where the zeroth bit is the least-significant bit. As the network topology is fixed, setting the output switches is trivially oblivious. The output switches are then applied to their inputs in P.

ControlBits(P, d) follows a similar process. We first observe that it uses our modified topology. The input switches are applied to items $k = \lceil n/2 \rceil$ apart (Line 10) rather than one apart. Consequently, the subnetwork inputs are contiguous (Lines 17 and 18), which improves memory locality during the recursive calls.

The algorithm is also defined for all n, not just powers of two. To achieve this generalization, we modify the process of modular reduction while setting input switches and of extracting the control bits while setting the output switches. In Lines 12–14, each permutation value is reduced modulo k (accomplished via a shift and a subtraction), and a bit indicating if the value changed is stored at the end (accomplished via a shift and an OR). The result is that at the dth recursive call, each element of P stores a bit array in the d least-significant bits and its reduced values in the remaining (most-significant) bits. After setting the output switches as the least significant bits (Lines 20–22), this reduction process is undone by reversing it (Lines 27–29).

The biggest change to the Lee algorithm comes in setting the input switches (Line 6). In both Lee's algorithm and our oblivious algorithm, each input has a *partner* with which it shares an input switch (in our topology, i and $i \pm k$ are partners). Each permutation value $v \leftarrow P[i] \gg d$ also has an *associate* with which it shares the same residue class modulo k (in our topology, v and $v \pm k$ are associates). A "back-and-forth" process is repeatedly performed between the permutation inputs and outputs. It starts with the input index f that is sent directly to the top subnetwork without entering an input switch (index $\lceil n/2 \rceil - 1$ in our topology). In each iteration, (1) g is set to the fth input value; (2) r is set to the associate of g; (3) s is set to the index of the input that contains r; (4) the switch that s is an input to is set to put r on the output leading to the bottom subnetwork; (5) f is updated to the partner of s; and (6) if f then has the same value it had in the first iteration (yielding a *cycle* during this process), f is set to an arbitrary input index that has not yet had its input switch set, starting a new cycle. Setting the input switch as in Step 4 ensures that the two values with the same residue class are sent to different subnetworks, ultimately producing the desired Complete Residue Systems as inputs to each subnetwork.

We set the input switches obliviously with InBits (Figure 4). The challenging steps to make oblivious are the *forward* map under the given permutation P (Step 1), the *reverse* map under P (Step 3), and starting a new cycle (Step 6). For the forward map, we use CreateForwardLookup (Figure 5) to create a lookup table F that contains the n permutation mappings of P. Each mapping is labeled

Figure 4: InBits(P, d): Compute Waksman control bits for input layer of switches for permutation P at recursion level d.

```
$n \leftarrow |P|$
$k \leftarrow \lceil n/2 \rceil$
Create array $C$ of length $k-1$.                          ▷Control bits
if $n \le 2$ then
    return $C$
Create array $S$ of length $k-1$.       ▷Switch numbers of control bits
Generate $\lambda$ random bits $\kappa_1$.          ▷PRP key for forward lookup
Generate $\lambda$ random bits $\kappa_2$.          ▷PRP key for reverse lookup
$F \leftarrow$ CreateForwardLookup$(P, d, \kappa_1)$
$R \leftarrow$ CreateReverseLookup$(F, \kappa_2)$
$U \leftarrow$ CreateUnselectedCounts$(n, \emptyset)$
▷ Perform initial back-and-forth with fixed input $k-1$
$f, g, \ell \leftarrow$ ForwardOrRand$(F, U, k-1, 0, \kappa_1)$        ▷Map $k-1$
DecUnselectedCounts$(U, \ell)$        ▷Mark $k-1 \longrightarrow g$ as selected
$c \leftarrow f$                ▷Cycle start is $f$ (which contains $k-1$)
$r \leftarrow g +$ OSelect$(-k, k, \llbracket g < k \rrbracket)$        ▷$r$ is the associate of $g$
$s, \ell \leftarrow$ HashMapGet$(R, \mathrm{PRP}(r, \kappa_2))$          ▷Reverse map of $r$
DecUnselectedCounts$(U, \ell)$        ▷Mark $s \longrightarrow r$ map as selected
$f \leftarrow s +$ OSelect$(-k, k, \llbracket s < k \rrbracket)$          ▷$f$ is the partner of $s$
▷ Repeat the back-and-forth process to compute the control bits
for $i \leftarrow 0, \ldots, k-2$ do
    $b \leftarrow \llbracket f = c \rrbracket$              ▷Indicate if current cycle has ended
    ▷ If $b$, choose random unselected $f \longrightarrow g$ map, else map $f$
    $f, g, \ell \leftarrow$ ForwardOrRand$(F, U, f, b, \kappa_1)$
    DecUnselectedCounts$(U, \ell)$        ▷Mark $f \longrightarrow g$ as selected
    $c \leftarrow$ OSelect$(c, f, b)$ ▷If new cycle begun, update cycle start
    $r \leftarrow g +$ OSelect$(-k, k, \llbracket g < k \rrbracket)$      ▷$r$ is the associate of $g$
    $s, \ell \leftarrow$ HashMapGet$(R, \mathrm{PRP}(r, \kappa_2))$        ▷Reverse map of $r$
    DecUnselectedCounts$(U, \ell)$        ▷Mark $s \longrightarrow r$ as selected
    $C[i] \leftarrow \llbracket f \ge k \rrbracket$      ▷Control bit to put $f$ in top subnetwork
    $S[i] \leftarrow f -$ OSelect$(0, k, \llbracket f \ge k \rrbracket)$     ▷Switch number of $f$
    $f \leftarrow s +$ OSelect$(-k, k, \llbracket s < k \rrbracket)$        ▷$f$ is the partner of $s$
OSort$(S, C)$   ▷Order control bits by increasing switch number
return $C$
```

Figure 5: CreateForwardLookup(P, d, κ): Create data structure for forward map under permutation in P at recursion depth d using key κ.

```
$n \leftarrow |P|$
$k \leftarrow \lceil n/2 \rceil$
$m \leftarrow 2k$
Create array $L$ of length $m$.          ▷Random labels for mappings
for $i \leftarrow 0, \ldots, m-1$ do
    $L[i] \leftarrow \mathrm{PRP}(i, \kappa)$
Create array $Q$ of length $m$.              ▷Permutation mappings
for $i \leftarrow 0, \ldots, n-1$ do
    $Q[i] \leftarrow (i, P[i] \gg d)$
if $n \ne m$ then    ▷Add dummy mapping to make total size even
    $Q[n] \leftarrow (n, n)$
OSort$(L, Q)$ ▷Randomize order of mappings in $Q$ by sorting on $L$
return $(L, Q)$
```

Figure 6: ForwardOrRand(F, U, f, b, κ): If $b = 0$, look up forward map from f under permutation encoded in F under key κ. Else (i.e., if $b = 1$), look up a uniformly random mapping in F from among those that U indicates are unselected. Returns mapping input f, output g, and index ℓ.

```
$L, Q \leftarrow F$
$n \leftarrow |L|$
$h \leftarrow \mathrm{PRP}(f, \kappa)$                      ▷Label to search for in $L$
$i \leftarrow 0$                    ▷Start index of range in $L$ and $U$ to search
$j \leftarrow n-1$                  ▷End index of range in $L$ and $U$ to search
$u \leftarrow U[n-1]$        ▷Number of unselected items in search range
$\rho \leftarrow$ ORand$(u)$                    ▷Random unselected mapping
while true do  ▷Binary search for label $h$ or unselected-item $\rho$
    $\ell \leftarrow \lfloor (i+j)/2 \rfloor$
    if i = j then
        break
    $z \leftarrow$ OSelect$(\llbracket L[\ell] < h \rrbracket, \llbracket U[\ell] \le \rho \rrbracket, b)$ ▷Search direction
    if $z = 0$ then          ▷Continue search in first half of range
        $j \leftarrow \ell$
        $u \leftarrow U[\ell]$
    else                      ▷Continue search in last half of range
        $i \leftarrow \ell + 1$
        $u \leftarrow u - U[\ell]$
        $\rho \leftarrow \rho - U[\ell]$
$f, g \leftarrow Q[\ell]$
return $(f, g, \ell)$
```

Figure 7: CreateReverseLookup(F, κ): Create hash map containing reverse map of permutation in forward-lookup structure F using key κ.

```
$L, Q \leftarrow F$
$n \leftarrow |Q|$
Create empty hash map $R$ sized to contain $n$ items.
for $i \leftarrow 0, \ldots, n-1$ do
    $x, y \leftarrow Q[i]$               ▷ $Q[i]$ contains forward map $x \longrightarrow y$
    $z \leftarrow \mathrm{PRP}(y, \kappa)$             ▷Create lookup key $z$ as PRP of $y$
    HashMapInsert$(R, z, (x, i))$ ▷Insert value $(x, i)$ under key $z$
return $R$
```

by a PRP applied to its input, and F is (obliviously) sorted by those labels. This structure allows each mapping to be looked up via a binary search on the labels, which is performed by ForwardOrRand (Figure 6). For the reverse map, we simply store the n inverse permutation mappings of P in a hash map R with a PRP applied to the input value as the lookup key, produced by CreateReverseLookup (Figure 7). Because each permutation mapping is looked up at most once and in either the forward or the reverse direction (but not both), the lookup pattern appears random, which is how it achieves obliviousness. We must use different keys for the PRPs used in F and R because, while each mapping will be looked up in at most one direction, a given integer might be looked up twice (i.e., an f in some iteration might equal an s in some iteration). Also note

Figure 8: CreateUnselectedCounts(n, U): Create data structure recursively storing the total unselected count and the unselected count in the first half. Call initially with $U = \emptyset$.

```
if $U = \emptyset$ then
    Create a new array $U$ of length $n$.
    $U[n-1] \leftarrow n$    ▷Last item contains total unselected count
if $n < 2$ then
    return $U$
$k \leftarrow \lceil n/2 \rceil$
$U[k-1] \leftarrow k$ ▷Midpoint contains unselected count in first half
▷ Recursively add counts to each half
CreateUnselectedCounts($k, U[0..(k-1)]$)
CreateUnselectedCounts($n-k, U[k..(n-1)]$)
return $U$
```

Figure 9: DecUnselectedCounts(U, ℓ): Decrement counts in U to indicate that position ℓ has been selected. Modifies U.

```
$n \leftarrow |U|$
$k \leftarrow \lceil n/2 \rceil$
if $\ell < k$ then                    ▷Selected item is in first half
    $U[n-1] \leftarrow U[n-1] - 1$
    if $n > 1$ then
        DecUnselectedCounts($U[0..(k-1)], \ell$)
else                                  ▷Selected item is in last half
    DecUnselectedCounts($U[k..(n-1)], \ell - k$)
```

that we store the input switch settings in the order they are determined along with their switch numbers (Lines 30–31), and then we obliviously sort them by switch number to put them in the required order (Line 33).

We could have used a hash map for the forward direction except for a complication that arises when a back-and-forth cycle ends. After a cycle ends, the next lookup (if any) is in the forward direction, and we need to look up a mapping that has not yet been queried in either direction. To accomplish this, CreateUnselectedCounts (Figure 8) creates a data structure U to track which mappings have been queried in either direction and to enable the uniformly random selection of an unqueried one. U is an array containing in its middle element the number of unselected mappings in the first half of the forward-lookup table F, and then its two halves contains the same values recursively for the corresponding halves of F. Therefore, ForwardOrRand can also look up a uniformly random unselected mapping while still performing a binary search, making its execution indistinguishable from querying a specific mapping. Maintaining U requires decrementing its counts with each lookup in either direction, which is accomplished by DecUnselectedCounts (Figure 9). Note that DecUnselectedCounts reveals which counts are decremented but maintains obliviousness because doing so reveals no more than which location was queried in F or R, which is both pseudorandom and already apparent in prior memory accesses. We next present an example of running ControlBits.

3.4 Worked Example of ControlBits

Figure 10 illustrates an example of running our ControlBits algorithm to set the switches of a Waksman network using our topology. The examples is on a network with $n = 9$ inputs. In the figure, the switches are shown as crossbars connecting two inputs, and they are applied to values from left to right. Subnetworks are indicated by a dashed rectangle and labeled by a letter in the upper left. For example, the whole (i.e., $n = 9$) network contains a top subnetwork A with 5 inputs and a bottom subnetwork E with 4 inputs, while subnetwork A itself contains a top subnetwork B with 3 inputs and a bottom subnetwork D with 2 inputs.

The control bits of the network in Figure 10 are set to implement the permutation shown: $P = (6, 2, 3, 7, 5, 1, 8, 0, 4)$. A dashed red crossbar indicates a switch that at the end of the algorithm is unset (i.e., a switch with a control bit of zero), and a solid green crossbar indicates a switch that ends up set (i.e., a switch with a control bit of one). The numeric values that appear throughout are the values of P as they are propagated through the network during ControlBits. The notation $\mathbf{x} : y \triangleright_k \mathbf{w} : y\mathbf{z}$ indicates that x gets reduced modulo k to w, y is the sequence of bits recording if prior modulo reductions changed the value or not, and z is 1 if x changed under this modulo reduction (i.e., $z = 1$ if $w = x - k$, and $z = 0$ if $w = x$). The bold values emphasize which components of a given entry in P are potentially changed at this step. Similarly, the notation $\mathbf{w} : y\mathbf{z} \triangleleft_k \mathbf{x} : y$ indicates the reversal of the modulo reduction of x by k that yielded w, where y records the action of modulo reduction before the one being reversed and z indicated if x changed under this reduction.

We follow Figure 10 from left to right to observe the actions of ControlBits. It initially uses InBits to compute control bits for the first set of input switches. The modulus it uses is $k = \lceil 9/2 \rceil = 5$. Starting with input $f = 4$, which is the input that leads to no switch at this level of recursion, it is mapped to $g = P[4] = 5$, the associate is then $s = 5 - k = 0$, its inverse map is $r = 7$ because $P[7] = 0$, and then f is updated to its partner $f = 7 - k = 2$. This back-and-forth process continues and sets control bits as follows:

(1) Set $g = 3$, $r = 8$, $s = 6$. Set the control bit of the switch containing input $f = 2$ (and input 7) to $\llbracket f \geq k \rrbracket = \llbracket 2 \geq 5 \rrbracket = 0$, ensuring that the associate values 5 and 0 (i.e., the prior values of g and s) will be inputs to different subnetworks. Set $f = s - k = 1$
(2) Set $g = 2$, $r = 7$, and $s = 3$. Then set the control bit of the switch containing input $f = 1$ to $\llbracket 1 \geq 5 \rrbracket = 0$. Set $f = 8$.
(3) Set $g = 4$, $r = 9$, and $s = 9$. Then set the control bit of the switch containing input $f = 8$ to $\llbracket 8 \geq 5 \rrbracket = 1$. Set $f = 4$.
(4) The cycle has ended because $f = 4$ was the initial value. The remaining mappings are $0 \longrightarrow 6$ and $5 \longrightarrow 1$, and ForwardOrRand randomly selects $0 \longrightarrow 6$ and sets $f = 0$. Set $g = 6$, $r = 1$, and $s = 5$. Then set the control bit of the switch containing input $f = 0$ to $\llbracket 0 \geq 5 \rrbracket = 0$.

The control bits for the first set of input switches are thus set. The switches are applied to the values in P, and those values are reduced modulo $k = 5$ with bits appended recording any reduction are stored. This process is shown in Figure 10 by the values after the initial input switches. Observe that the reduced values provided to the top subnetwork (i.e., network A) is a Complete Residue System

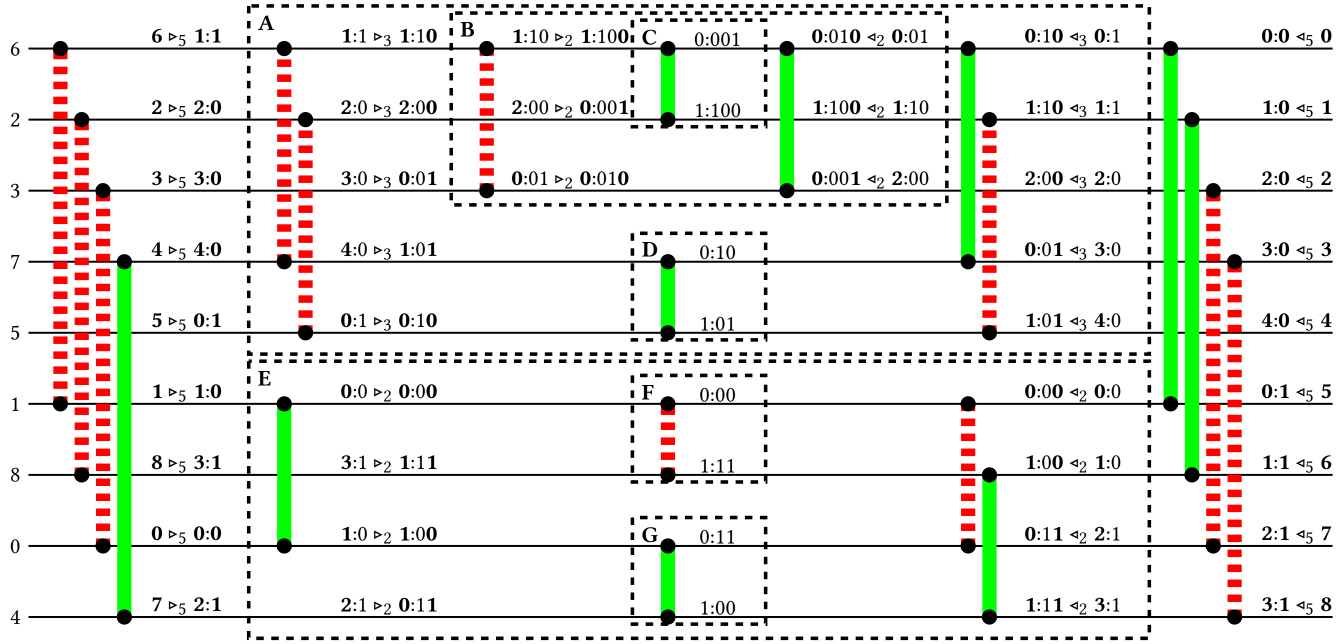

6
2
3
7
5
1
8
0
4
6 $\triangleright_5$ 1:1
2 $\triangleright_5$ 2:0
3 $\triangleright_5$ 3:0
4 $\triangleright_5$ 4:0
5 $\triangleright_5$ 0:1
1 $\triangleright_5$ 1:0
8 $\triangleright_5$ 3:1
0 $\triangleright_5$ 0:0
7 $\triangleright_5$ 2:1
A
1:1 $\triangleright_3$ 1:10
2:0 $\triangleright_3$ 2:00
3:0 $\triangleright_3$ 0:01
4:0 $\triangleright_3$ 1:01
0:1 $\triangleright_3$ 0:10
B
1:10 $\triangleright_2$ 1:100
2:00 $\triangleright_2$ 0:001
0:01 $\triangleright_2$ 0:010
C
0:001
1:100
D
0:10
1:01
0:010 $\triangleleft_2$ 0:01
1:100 $\triangleleft_2$ 1:10
0:001 $\triangleleft_2$ 2:00
0:10 $\triangleleft_3$ 0:1
1:10 $\triangleleft_3$ 1:1
2:00 $\triangleleft_3$ 2:0
0:01 $\triangleleft_3$ 3:0
1:01 $\triangleleft_3$ 4:0
E
0:0 $\triangleright_2$ 0:00
3:1 $\triangleright_2$ 1:11
1:0 $\triangleright_2$ 1:00
2:1 $\triangleright_2$ 0:11
F
0:00
1:11
G
0:11
1:00
0:00 $\triangleleft_2$ 0:0
1:00 $\triangleleft_2$ 1:0
0:11 $\triangleleft_2$ 2:1
1:11 $\triangleleft_2$ 3:1
0:0 $\triangleleft_5$ 0
1:0 $\triangleleft_5$ 1
2:0 $\triangleleft_5$ 2
3:0 $\triangleleft_5$ 3
4:0 $\triangleleft_5$ 4
0:1 $\triangleleft_5$ 5
1:1 $\triangleleft_5$ 6
2:1 $\triangleleft_5$ 7
3:1 $\triangleleft_5$ 8

Figure 10: A Waksman network in our topology for $n = 9$ with control bits implementing permutation $P = (6, 2, 3, 7, 5, 1, 8, 0, 4)$. A dashed red crossbar indicates switch with a control bit of zero, and a solid green crossbar indicates a switch with a control bit of one. The numeric values are the values of P as they are propagated through the network and modified during ControlBits.

(CRS) modulo 5, and the reduced values for the bottom subnetwork (i.e., network E) are a CRS modulo 4; that is, those sets of values are $\{0, 1, 2, 3, 4\}$ and $\{0, 1, 2, 3\}$, respectively.

Control bits are then set and applied recursively in the top and bottom subnetworks—networks A and E, respectively. Network A takes 5 inputs, and network E takes 4 inputs. They each contain subnetworks themselves, with the recursion ending when a network has 1 or 2 inputs. A one-input network is null (i.e., has no switches). A two-input network (e.g., network C) consists of a single output switch, which has its control bit effectively set to the value of its first input (by first appending that value to its attached bit array).

After the subnetworks are set and applied, control bits are computed for the last set of output switches. For each pair of inputs to a switch, the control bit is taken to be the last element of the bit array for the first input. Therefore, we see that the output switch containing input 0 is set to 1, the switch for input 1 is set to 1, the switch for input 2 is set to 0, and the switch for input 3 is set to 0. These switches are applied to the input values, and then the reduction modulo 5 is undone, yielding the original set of values $\{0, 1, \ldots, 8\}$ in sorted order. Note that, because each subnetwork also outputs in sorted order the set of values in its input CRS, the two inputs to a given output switch have the same values modulo 5, and therefore setting the control bit based on the reduction bit of the first input puts those two values in the correct order (and the correct positions) once the reduction modulo 5 is undone.

Figure 11: ApplyPerm(C, D): Apply permutation encoded in Waksman controls bits C to array D. Modifies D. Oblivious to values in C and D but not to $|C|$ or $|D|$.

```
$n \leftarrow |D|$
if $n < 2$ then
    return
$C_{\text{in}}, C_{\text{top}}, C_{\text{bot}}, C_{\text{out}} \leftarrow C$
$k \leftarrow \lceil n/2 \rceil$
if $n > 2$ then
    for $i \leftarrow 0, \ldots, k-2$ do                ▷Apply input-layer switches
        OSwap($D, i, k+i, C_{\text{in}}[i]$)
    ▷ Recursively apply switches in top and bottom subnetworks
    ApplyPerm($C_{\text{top}}, D[0..k-1]$)
    ApplyPerm($C_{\text{bot}}, D[k..n-1]$)
for $i \leftarrow 0, \ldots, n-k-1$ do                 ▷Apply output-layer switches
    OSwap($D, i, k+i, C_{\text{out}}[i]$)
```

3.5 Applying Permutations

Once the control bits are set, it is straightforward to obliviously apply the permutation that they encode. Figure 11 shows the algorithm ApplyPerm, which does so under our Waksman topology. It first applies the input switches, then recursively applies the two subnetworks, and finally applies the output switches. The inverse permutation can also easily be applied, as ApplyInvPerm shows (Figure 12), by applying the output switches first and the input switches last. Because the Waksman networks have a fixed topology for a given n, these algorithms are fully oblivious.

Figure 12: ApplyInvPerm(C, D): Apply inverse of permutation encoded in Waksman controls bits C to array D. Modifies D. Oblivious to values in C and D but not to $|C|$ or $|D|$

```
$n \leftarrow |D|$
if $n < 2$ then
    return
$C_{\text{in}}, C_{\text{top}}, C_{\text{bot}}, C_{\text{out}} \leftarrow C$
$k \leftarrow \lceil n/2 \rceil$
for $i \leftarrow 0, \ldots, n-k-1$ do                ▷Apply output-layer switches
    OSwap$(D, i, k+i, C_{\text{out}}[i])$
if $n > 2$ then
    ▷ Recursively apply switches in top and bottom subnetworks
    ApplyInvPerm$(C_{\text{top}}, D[0..k-1])$
    ApplyInvPerm$(C_{\text{bot}}, D[k..n-1])$
    for $i \leftarrow 0, \ldots, k-2$ do              ▷Apply input-layer switches
        OSwap$(D, i, k+i, C_{\text{in}}[i])$
```

Figure 13: WaksShuffle(D): Put items in D in a uniformly random order. Divided into an offline phase (where only $|D|$ is needed) and an online phase.

```
▷ Begin offline phase
$n \leftarrow |D|$
Create array $P$ of length $n$.
for $i \leftarrow 0, \ldots, n-1$ do
    $P[i] \leftarrow i$
OShuffle$(P)$                ▷Make permutation uniformly random
$C \leftarrow$ ControlBits$(P, 0)$
▷ Begin online phase
ApplyPerm$(C, D)$
```

Figure 14: WaksSort(K, D): Sort D in increasing order according to the corresponding keys in K. Also sort K.

```
$n \leftarrow |D|$
Create array $P$ of length $n$.
for $i \leftarrow 0, \ldots, n-1$ do
    $P[i] \leftarrow i$
OSort$(K, P)$ ▷Make $P$ the inverse of the permutation that sorts $K$
$C \leftarrow$ ControlBits$(P, 0)$
ApplyInvPerm$(C, D)$        ▷Put $D$ in same order as now-sorted $K$
```

4 OBLIVIOUS SHUFFLING AND SORTING

While our oblivious Waksman algorithms may be useful in many settings, we focus on how they can be used for shuffling and sorting. Waksman networks might achieve faster shuffling and sorting because they perform fewer pairwise swaps of data items than other oblivious algorithms. They do, however, carry the additional cost of setting the control bits. Doing so operates only on the *permutation* and, if the permutation does not depend on the data, not the *data items*. Therefore, Waksman networks can admit an *offline* computation phase before the data items are available, in which the control bits are set at a cost independent of the data-item size, and an *online* phase once the data items are available, in which the permutation is applied with relatively few swaps of the data items themselves.

4.1 Shuffling

The goal of shuffling is to put an array of data items in a uniformly random order. Shuffling obliviously can hide the resulting order as well as the data contents. Let D be an array of data items, with $|D| = n$. The fully oblivious WaksShuffle(D) algorithm (Figure 13) chooses a random permutation by applying an existing oblivious shuffle, OShuffle, to the array $(0, \ldots, n-1)$. This permutation is used to set the Waksman control bits. The resulting network is applied to D, accomplishing the shuffle. Despite using an existing oblivious shuffle as a subroutine, this algorithm can improve efficiency if the data items are large because that shuffle is only applied to integers less than n. Moreover, setting the control bits occurs during an offline phase because the permutation does not depend on the data, though the number of items n must be known.

4.2 Sorting

The fully oblivious WaksSort(K, D) algorithm (Figure 14) takes an array of keys K and an array of data items D. Its goal is to sort the items in D by increasing value of their corresponding keys, where item $D[i]$ corresponds to key $K[i]$. It uses an existing oblivious sorting algorithm, OSort, to put the keys in increasing order and simultaneously rearrange the array $P = (0, \ldots, n-1)$ in the same way (i.e., if the ith element of K is moved to position j, the same happens to P). The result is that P contains the inverse permutation needed to sort the items of D by their (original) keys. P is used to set the Waksman control bits, and then the inverse permutation is applied to D with ApplyInvPerm. In a setting where the data items are large compared to the keys, WaksSort can yield a faster sort because the existing oblivious sort is only applied to the keys.

Faster *online* oblivious sorting can be achieved by executing WaksShuffle(D) and then following it with the non-oblivious quicksort. In this sorting algorithm (WaksShuffle+QS), the control bits are set to encode a uniformly random permutation during the offline phase. Then, for the online phase, the Waksman network is applied to D, and quicksort is run on the result. Performing a non-oblivious comparison sort after an oblivious shuffle (assuming keys are distinct) does not violate obliviousness [13]. Also, even with the added cost of non-oblivious sorting, the online phase may be faster than existing oblivious sorting algorithms because both ApplyPerm and quicksort have $O(\beta n \log n)$ runtime (where β is the size of each data item) with small constants.

5 ANALYSIS

We show that WaksShuffle and WaksSort are correct and fully oblivious. By way of doing that, we demonstrate these properties for ControlBits, ApplyPerm, and ApplyInvPerm. These algorithms can be used for applications of Waksman networks outside of shuffling and sorting. We also analyze the efficiency of all of our algorithms. The proofs for all theorems appear in Appendix A.

5.1 Correctness

We first show the correctness of our Waksman-network algorithms. Although these algorithms are similar to prior work, we have made non-trivial modifications for both obliviousness and efficiency.

Theorem 1 shows that ControlBits$(P, 0)$ produces control bits C such that ApplyPerm(C, D) applies P to D, and ApplyInvPerm(C, D) applies the inverse of P. Note the permutation representation that P maps i to $P[i]$.

Theorem 1. *Let P be an array, with $|P| = n$, containing the values $\{0, \ldots, n-1\}$ in some order. Let $Q = (0, \ldots, n-1)$. Let P' be a copy of P (as P' will get modified), and let $C =$ ControlBits$(P', 0)$. Then, if we run ApplyPerm(C, P), then $P = Q$ afterwards, and if instead we run ApplyInvPerm(C, Q), then again $P = Q$ afterwards.*

Theorem 2 shows that WaksShuffle(D) shuffles the items of D. Similarly, Theorem 3 shows that WaksSort(K, D) sorts the keys in K in nondecreasing order and puts the data items in the same order.

Theorem 2. *WaksShuffle(D) puts the items in D in a uniformly random order.*

Theorem 3. *WaksSort(K, D) sorts the items in K in increasing order and also applies that permutation to D.*

5.2 Obliviousness

We next show that our algorithms are fully oblivious with respect to their secret inputs.

Theorem 4 shows that ControlBits(P, d) is fully oblivious with respect to the input permutation P. Therefore, its operation does not reveal any information about the permutation for which control bits are being computed. Note that this obliviousness holds despite the facts that i) the binary search of ForwardOrRand reveals that location of the item being looked up, and ii) the entries in the reverse-lookup hash map R are inserted in the order they appear in the forward-lookup table F and are thus observably linked to them. These properties do not violate obliviousness because the mappings in F are obliviously shuffled, the keys in R are pseudorandom, and a mapping is looked up exactly once and in only one direction (forward or reverse).

Theorem 4. *ControlBits(P, d) is fully oblivious with respect to P.*

Theorem 5 shows that ApplyPerm(C, D) and ApplyInvPerm(C, D) are fully oblivious with respect to both the input control bits C and the input data D. This property guarantees that the memory and instruction traces hide all information about what permutation is being applied.

Theorem 5. *ApplyPerm(C, D) and ApplyInvPerm(C, D) are fully oblivious with respect to C and D.*

Theorems 6 and 7 show that WaksShuffle and WaksSort are fully oblivious with respect to their inputs. They thus hide the values of the data items and the permutation that is applied.

Theorem 6. *WaksShuffle(D) is fully oblivious with respect to D.*

Theorem 7. *WaksSort(K, D) is fully oblivious with respect to K and D.*

5.3 Efficiency

Our algorithms are designed to be practically efficient rather than asymptotically optimal. For an input of n data items each of size β (or for sorting, where a key and data item have total size β), it is possible to perform oblivious shuffling and sorting in $O(\beta n \log n)$ time with a sorting network [1], but the oblivious shuffling algorithm (ORShuffle [41]) and sorting algorithm (bitonic sort [4, 15]) that are fastest in practice have a $O(\beta n \log^2 n)$ runtime. We use these as our OShuffle and OSort subroutines for practicality, but we note that using a $O(\beta n \log n)$ choice here would improve the asymptotic runtime of our algorithms. Our analysis also assumes that the computation can apply instructions on machine words of size $O(\log n)$ in constant time. For obliviously setting Waksman control bits, the fastest existing algorithm currently is from Nassimi-Sahni [32], as made oblivious by Bernstein [6], which has a runtime of $O(n \log^4 n)$.

Theorem 8 shows that the runtime of ControlBits is $O(n \log^3 n)$.

Theorem 8. *Let $|P| = n$. The runtime of ControlBits$(P, 0)$ is $O(n \log^3 n)$.*

The oblivious sorts in ControlBits are major costs in theory and in practice. Let $k = \lceil n/2 \rceil$. There are $\lceil \log_2 n \rceil$ levels of recursion in ControlBits, and in each recursive call an OSort is performed on $2k$ (fixed-size, independent of the item size β) elements to create the forward lookup table, and another OSort is performed on $k - 1$ elements to sort the input control bits. These sorts take time $O(n \log^2 n)$. All other operations in a given recursive call take $O(n \log n)$ time, dominated by the k calls to ForwardOrRand and $2k$ calls to DecUnselectedCounts, where each of those calls performs a $O(\log n)$ binary search. If OSort were implemented with a $O(n \log n)$ algorithm, then ControlBits would have a $O(n \log^2 n)$ runtime.

Theorem 9 shows that, for data items of size β each, the runtimes of ApplyPerm and ApplyInvPerm are both $O(\beta n \log n)$.

Theorem 9. *Let $|D| = n$, and let each item in D be of size β. The runtime of ApplyPerm(C, D) and ApplyInvPerm(C, D) is $O(\beta n \log n)$.*

The expensive operations in both algorithms are applying the switches to the data items, and each resulting OSwap takes time $O(\beta)$, while there are at most $n \log_2 n$ switches total.

Theorem 10 shows that the offline runtime of WaksShuffle is $O(n \log^3 n)$, while the online runtime for items of size β is $O(\beta n \log n)$.

Theorem 10. *Let $|D| = n$, and let each item in D be of size β. The offline runtime of WaksShuffle(D) is $O(n \log^3 n)$, and the online runtime of WaksShuffle(D) is $O(\beta n \log n)$.*

The main cost during the offline phase is ControlBits, which Theorem 8 shows has a $O(n \log^3 n)$ runtime, and the runtime would become $O(n \log^2 n)$ if a $O(n \log n)$ OSort were used. The online phase is simply ApplyPerm. The β factor only appears in the online phase, and so if β is large relative to n (i.e., $\beta = \omega(\log n)$), the total runtime of WaksShuffle can be faster than directly applying the existing oblivious shuffle OShuffle. We can estimate the potential speedup in the online phase compared to the practically fastest ORShuffle [41] (which has no offline phase) by comparing the number of OSwap operations performed on the data items. In both algorithms, the only operations on the data items are OSwaps, and these operations dominate the runtime of the (online) computation. ORShuffle performs approximately $(n/4)((\log_2 n) + 1)\log_2 n$ OSwaps, and the online phase of WaksShuffle performs approximately $n \log_2 n - n + 1$, where in both cases these counts are exact when n is a power of two. Therefore, the WaksShuffle online phase performs a little over $((\log_2 n) + 1)/4$ times fewer OSwaps than ORShuffle. For $n = 2^{20}$ items, for example, the online speedup would be about 5.3×.

Theorem 11 shows that, when data keys are of size α and data items are of size β, the total runtime of WaksSort is $O(n \log^3 n + \alpha n \log^2 n + \beta n \log n)$.

Theorem 11. *Let $|K| = n$, $|D| = n$, each item in K be of size α, and each item in D be of size β. The (online) runtime of WaksSort(K, D) is $O(n \log^3 n + \alpha n \log^2 n + \beta n \log n)$.*

All of WaksSort is online because the keys are needed to determine the permutation used to set the Waksman control bits. The $O(n \log^3 n)$ term comes from the call to ControlBits, and it would be $O(n \log^2 n)$ if OSort were instead $O(n \log n)$. The $\alpha n \log^2 n$ term is from applying OSort to the keys K, and it would similarly reduce to $O(\alpha n \log n)$ with a $O(n \log n)$ OSort. The $\beta n \log n$ term is from ApplyInvPerm. Note that if the keys are small relative to the items (i.e., $\alpha = O(\beta/\log n)$), and the items are large relative to n (i.e., $\beta = \omega(\log n)$), then the overall runtime is $O(\beta n \log n)$, which compares favorably to the $O(\beta n \log^2 n)$ runtime of the practically efficient bitonic sort. Bitonic sort also only operates on data items through OSwaps, and it uses the same number of them as ORShuffle. Therefore, for large items we again expect to see a speedup factor in practice of a little over $((\log_2 n) + 1)/4$ for WaksSort over bitonic sort. On the other hand, when the keys are relatively large, for example when they are simply the data items themselves, we expect no runtime improvement, as a large fraction of the runtime would be consumed by simply running OSort on the keys to determine the permutation P. WaksShuffle+QS, however, can yield a faster online sort for any size of keys. It has a $O(n \log^3 n)$ offline runtime, and its online runtime is $O((\alpha + \beta) n \log n)$, better than the $O((\alpha + \beta) n \log^2 n)$ online runtime for bitonic sort.

We also observe that ApplyPerm and the online phase of WaksShuffle have relatively low parallel step complexity. In ApplyPerm, applying an input or output layer of switches to items of any size can be performed in a single step, and so ApplyPerm can be executed in $2\lceil \log_2 n \rceil - 1$ parallel steps. The online phase of WaksShuffle has the same step complexity. Thus WaksShuffle improves on the $(\lceil \log_2 n \rceil + 1)\lceil \log_2 n \rceil/2$ step complexity of ORShuffle by a factor of a little more than $(\lceil \log_2 n \rceil + 1)/4$. For $n = 2^{20}$ items, for example, the speedup in online step complexity would be about 5.3×.

6 IMPLEMENTATION

We compare WaksShuffle and WaksSort against four different algorithms for shuffling and sorting. The first and obvious candidate is (i) a bitonic sorting network [4] for sorting (frequently used in the TEE literature [14, 23, 30, 34]) and bitonic shuffle for shuffling. Our implementation builds on top of the software artifact of Sasy et al. for oblivious shuffling algorithms in TEEs [41, 42], which presents two fully oblivious shuffle algorithms, namely (ii) ORShuffle and (iii) BORPStream. Their work demonstrates that ORShuffle always outperforms the then-state-of-the-art bitonic shuffle. On the other hand, BORPStream can be optimized to either minimize the total time to shuffle (V1) or minimize the time to complete a shuffle once all items are available (V2); BORPStream-V2 yields faster shuffling than ORShuffle when items to shuffle arrive in a streaming fashion, which is similar to the offline/online setting but requires processing time between inputs rather than before them.

Finally, we also compare our oblivious control bit setting algorithm for Waksman networks against (iv) Nassimi-Sahni's control bit setting algorithm [32]. In recent work, Bernstein presented an oblivious version of the Nassimi-Sahni control bit setting algorithm, making it a suitable candidate for fully oblivious shuffling and sorting within TEEs. We implement this algorithm ([6, §7, Fig 7.1]) and evaluate its performance against WaksShuffle and WaksSort. The underlying Waksman network of our Nassimi-Sahni implementation intentionally uses the standard Waksman network layout [48], so that we can contrast the performance improvements obtained by our locality-optimized Waksman network layout. We detail the results of this comparison at the end of Section 7.1.

Since we use the definition of full obliviousness of Sasy et al. [41], to ensure that our implementation is fully oblivious, we use the Fully Oblivious Assembly Verifier (FOAV) tool they provide. Using this tool we flag variables containing public (non-secret) values as "safe" within our WaksSort and WaksShuffle implementation. FOAV verified 598 out of 688 of the conditional branches generated in the final binary to be safe. Similar to prior work, we manually inspect the remainder to ensure that these are safe as well. The bulk of the conditional branches that FOAV could not mark as safe arise from branches within the C++ implementation of vectors and hash tables. However, since non-repeating inputs fed into the hash table operations are the outputs of the PRP there is no secret data dependence underlying any usage of the hash tables themselves. Similarly for vectors, the conditional branches are associated to the size of vectors which are publicly known and not hidden.

We provide additional details on our implementation-level optimizations in Appendix B. Our implementation is open-source and publicly available at https://crysp.uwaterloo.ca/software/obliv/.

7 EXPERIMENTS

All our experiments use a single core of a 2.3 GHz Intel 8380. Our machine runs Intel SGX2 and has 16 GB of Processor Reserved Memory. All our reported experiments and benchmarks are on a single-threaded implementation, but, as we note in Section 5.3, our algorithms can easily be parallelized to further reduce latency. In the experiments reported below we fix the number of items n to 2^{20} and vary the size of each item β to explore performance across different item sizes. We performed these experiments for a variety of n between 2^{15} and 2^{20}, and results for other n appear in Appendix C. Those results are qualitatively identical to those presented in this section.

For shuffling, we run experiments to contrast the performance of the four shuffling algorithms listed in Section 6. For sorting, WaksSort and Nassimi-Sahni generate control bits to permute the data to its sorted state. Neither of these can provide a meaningful offline phase, as the permutation to sort is dependent on the data. WaksShuffle+QS, on the other hand, can set the control bits for WaksShuffle in an offline phase, then apply the Waksman network

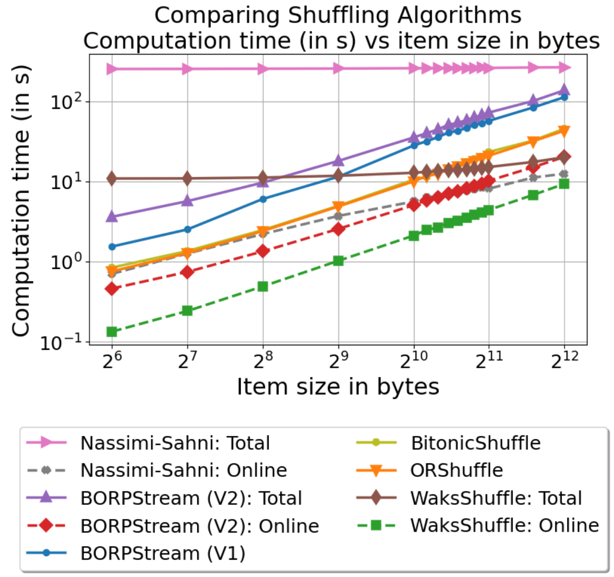


Figure 15: Comparing time taken for shuffling against item size with $n = 2^{20}$ items. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.

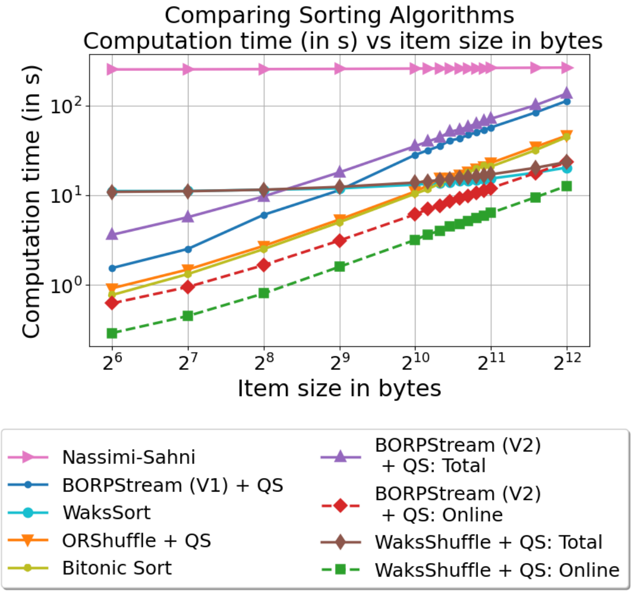


Figure 16: Comparing time taken for sorting against item size with $n = 2^{20}$ items. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.

(to shuffle) and perform a non-oblivious quicksort in an online phase. ORShuffle and BORPStream can similarly be converted into offline/online sorting algorithms by using this oblivious-shuffle-then-sort paradigm.

7.1 Comparing Shuffling Algorithms

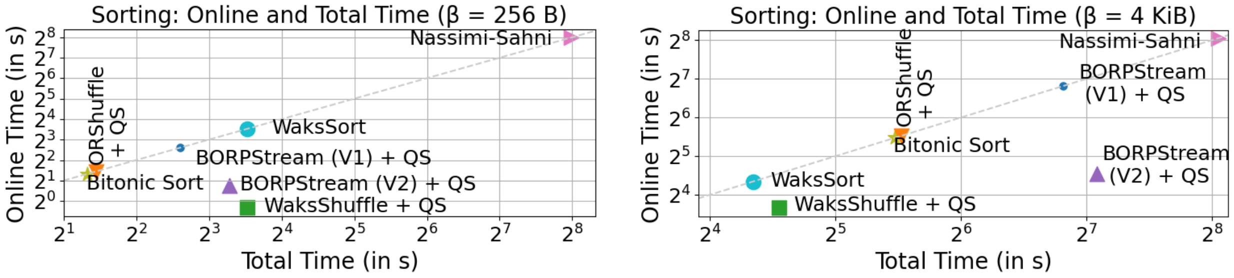
BORPStream's main advantage is its ability to split the computation into two phases: i) a first phase partially shuffles items as they become available by routing those items through a well-parameterized butterfly routing network to random destination buckets, and ii) a second "online" phase compacts (i.e., discards dummy items from) each bucket and individually shuffles each resulting bucket. This two-phase division significantly reduces the time to complete a shuffle in settings where the items to shuffle stream in over a long period of time. If all the data to shuffle is available up front, BORPStream offers little advantage as the total work done by BORPStream is more than that of ORShuffle. However, Sasy et al. note that BORPStream V1 (tuned to minimize total time) does outperform ORShuffle for large enough item sizes due to its memory locality advantage, as the second phase shuffles small subsets of items at a time.

In comparing our work with BORPStream, we give BORPStream every advantage we can. In particular, recall that WaksShuffle works in an offline/online model, implying that it can always reap the low online cost to shuffle, independent of how the data becomes available. Nonetheless, we compare our online time with BORPStream's online time assuming items are made available in a streaming fashion, and ignoring the cost of its first phase.

Figure 15 presents the results of comparing shuffling algorithms, and it demonstrates that WaksShuffle requires the least online time. In comparison with BORPStream (V2)'s online time, at the smallest and largest item sizes of 64 B and 4 KiB that our experiments cover, WaksShuffle's online time provides a 3.5× and 2.2× speedup respectively. However, note that BORPStream can only support this online cost in the streaming setting. In numerous application scenarios, all the data is available simultaneously (like oblivious database joins [23] or oblivious file systems [14]), and in such cases BORPStream's cost falls back to its V1 variant's total time. In comparison with BORPStream V1, our online time provides speedups of 11.5× and 12× respectively for 64 B and 4 KiB item sizes. Similarly, WaksShuffle demonstrates a 4.6–5.6× online speedup over ORShuffle. Our improvements arise from the fact that applying the Waksman network (once the control bits are generated) is a $O(\beta n \log n)$ process in comparison to the online (and total) costs of all the other algorithms which require $O(\beta n \log^2 n)$ operations.

The oblivious Nassimi-Sahni control bit setting algorithm incurs a $O(n \log^4 n)$ cost and has higher hidden constants than the ControlBits of WaksShuffle, due to the larger number of oblivious sort invocations it requires. The cost of setting control bits is agnostic of the underlying item size, though, shown by the total runtimes that are nearly constant as the item sizes increase for both WaksShuffle and Nassimi-Sahni in Figure 15. However, from Figure 15 we also observe that ControlBits of WaksShuffle is one to two orders of magnitude faster than that of Nassimi-Sahni.

Indeed, for large enough item sizes, the entirety of WaksShuffle is faster than all the other existing shuffling algorithms even without leveraging the offline/online split, a feat that remained out of reach for the Nassimi-Sahni algorithm across all our experiments. To determine this crossover point we test additional points close to this crossover, and we observe that the crossover point occurs at



(a) Sorting with a small item size (256 bytes) with $n = 2^{20}$

(b) Sorting with a large item size (4096 bytes) with $n = 2^{20}$

Figure 17: Scatter plots comparing online and total times (in s) for sorting algorithms with $n = 2^{20}$. WaksShuffle+QS consistently incurs the least online time irrespective of item size. For large enough items (β > 1400 B), the total times of WaksSort and WaksShuffle+QS are lower than all other fully oblivious sorting algorithms.

1400 B, a modest size of practical relevance for several applications. We also experimentally observe that this crossover point varies only slightly when n is varied between 2^{15} and 2^{20} (see Appendix C).

In terms of the online cost, we intentionally set up the Nassimi-Sahni implementation with the standard Waksman network topology to demonstrate the performance benefits of our topology.[2] We observe that the performance improvements from our topology are larger for smaller items, yielding up to a 5.3× improvement for 64 B items and tapering down to a 1.3× speedup at 4 KiB.

7.2 Comparing Sorting Algorithms

We present the results of our sorting experiments in Figure 16. As mentioned earlier, while we can generate control bits to use the Waksman network to sort a set of items, doing so makes the control bit setting algorithm a part of the online phase, as the permutation that sorts the items depends on the keys of the items. However, for large enough problem sizes, as with shuffling above, eventually the total time for WaksSort (which is almost coincident to that of WaksShuffle+QS) is lower than all currently known oblivious sorting algorithms. Furthermore, the shuffle-then-sort paradigm enables us to generate control bits (the expensive component) of the Waksman network in an offline phase, leaving just applying the Waksman permutation network followed by a non-oblivious sort in the online phase to sort the data. We evaluate ORShuffle and BORPStream for sorting in the same manner.

While we present BORPStream's online cost in Figure 16, recall from Section 7.1 that this offline/online split for BORPStream only applies in settings where items to shuffle arrive intermittently and is not a generic offline/online split like that supported by WaksShuffle+QS. Assuming that the application setting is compatible for BORPStream, the online time taken by WaksShuffle+QS still demonstrates a 1.8–2.2× speedup over BORPStream's online time to sort. If all the items to sort are available simultaneously, then the best competing algorithm is in fact Bitonic Sort, over which the online time of WaksShuffle+QS provides a 2.7–3.5× speedup as the item size increases from 64 B to 4 KiB.

7.3 Summarizing the State of Affairs

To provide a revised and concrete view of the state of affairs for fully oblivious sorting algorithms within TEEs, we present scatter plots detailing the online and total time tradeoffs in Figure 17. The corresponding figures for shuffling algorithms are qualitatively identical and are shown in Appendix C. As seen in Figures 17a and 17b, the online time to sort for WaksShuffle+QS is consistently and significantly lower than that of all other existing sorting algorithms. Furthermore, as the item sizes get larger, eventually (β >1400 B) WaksSort even has the lowest total times, with WaksShuffle+QS close behind. The algorithms' online times translate to query latency, and the total times translate to server CPU costs, which can determine the monetary cost of deploying these algorithms.

8 RELATED WORK

8.1 Oblivious Shuffling Algorithms

While oblivious shuffling algorithms specifically designed for TEEs have been proposed [8, 33], these algorithms are not fully oblivious as they assume portions of the enclave memory as private and unobservable to the adversary, which we now know to be untrue [24]. We refer the interested reader to Sasy et al. [41, §7.2], which provides comprehensive details on these algorithms and explains why attempting to fix these vulnerabilities leads to solutions no better than bitonic shuffle.

8.2 Oblivious Sorting Algorithms

Ebbesen's thesis [15] experimentally compares the performance of data-oblivious sorting algorithms and demonstrates bitonic sort to be the best deterministic oblivious sorting technique. In the TEE setting, bitonic sorting networks have become the standard approach for oblivious sorting [14, 23, 30, 34, 36, 51]. Recently, data-oblivious sorting algorithms have received renewed interest [3, 28, 38]. These works, however, are theoretical in nature, and while they demonstrate $O(n \log n)$ oblivious sorting algorithms, under the hood they leverage either the AKS network [1] or expander graphs [35], both of which have large constants hidden in the asymptotics that make them impractical.

[2] Instantiating Nassimi-Sahni with our topology would result in an online time matching our algorithm and a negligible change in its total time.

8.3 Waksman Networks

Beauquier and Darrot [5] show that Waksman networks can be extended to work for numbers of items n that are not powers of two. Their work gives Waksman network constructions for such non-power-of-two cases, and it compares the number of required switches to simply using the original Waksman networks for n', where n' is the next power of two greater than n.

Recently, Holland et al. [19] use Waksman networks for efficient permutation of data in the client-server model. Their work is in the setting where a client outsources their data to a cloud server and wishes to permute the data obliviously. To this end, they propose using Waksman networks and provide a control bit setting algorithm to do so with minimal I/O overheads. The control bit setting algorithm is the same as Waksman's "looping" algorithm, but, in order to minimize I/O and client storage overheads, they observe that the client can both configure the Waksman network and route items through it layer by layer. Their insight is that to do so the client need only store at most $2n$ switch settings locally (instead of $n \log_2 n$). They observe that each recursive Waksman subnetwork requires only its ancestors for correct switch setting, thus one only needs to store switch settings along one recursive path of the Waksman network at a time; i.e., once the algorithm recurses into a bottom subnetwork, the control bits of its top subnetwork can be released. As for the exterior (outer) switches, they pack them along with the items in the first half of the Waksman network, thus not requiring to store these switch settings either, nor to perform any switch configuration operations while routing items through the latter half of the Waksman network. Their algorithm to set control bits has a $O(n \log^2 n)$ complexity, but their model requires private unobservable memory ($O(n)$) at the client side, which by the definition of fully oblivious is not permissible in our setting, requiring us to design a novel algorithm.

The existence of unobservable memory makes the general multiparty computation (MPC) setting easier as well. In MPC, we can assume that some parties are honest and thus have local computation (including memory accesses and instructions) that is unobservable to the adversary. In the TEE setting, as we have argued, that assumption is not valid. Therefore, MPC protocols exist that take time linear in the number of items [16, 37], whereas the best fully oblivious shuffling algorithms take superlinear time.

Some applications of Waksman networks in the MPC setting have been proposed. Zahur et al. [50] revisit the original square-root ORAM by Goldreich and Ostrovsky [17] and leverage Waksman networks to replace the oblivious sorting primitive for efficiency improvements within the ORAM. Smart and Alaoui [45] demonstrate how to convert traditional single-party protocols into secret-shared counterparts, and as an application present an MPC protocol to shuffle data by setting and applying the control bits of a Waksman permutation network. In their setting, the control bits are generated for a secret-shared permutation produced by the parties participating in the MPC protocol. Both instances of these MPC-based Waksman networks use the original looping algorithm to set control bits [48], modified to work over secret-shared permutations.

Waksman networks have also been used by code-based cryptosystems [7, 11]. They are used to obliviously apply a secret permutation during decryption. The Waksman control bits are generated during key generation and are also generated obliviously (e.g., with Nassimi-Sahni) to prevent key leakage. Our ControlBits algorithm may speed up this key-generation process in these cryptosystems.

9 CONCLUSION

In this work we revisit Waksman networks and design novel fully oblivious algorithms to set their control bits and permute data items with them. We use these algorithms to create fully oblivious algorithms for shuffling and sorting data, two frequently used primitives in privacy-preserving systems using TEEs. The total overheads of our algorithms are lower than the current state-of-the-art algorithms for oblivious shuffling and sorting of moderately sized items (item size exceeding 1400 B). Furthermore, in the offline/online model, our algorithms provide a $O(\log n)$ asymptotic factor speedup, and they yield $> 5\times$ and $> 2.7\times$ concrete speedups in the online costs of shuffling and sorting, respectively, for any item size.

ACKNOWLEDGMENTS

This work was supported by the Office of Naval Research. We thank the Ontario Graduate Scholarships program, NSERC (CRDPJ-534381), and the Royal Bank of Canada for supporting this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi. 1983. An $O(n \log n)$ Sorting Network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC).*

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. https://software.intel.com/content/www/us/en/develop/articles/innovative-technology-for-cpu-based-attestation-and-sealing.html. Accessed May 2023.

[3] Gilad Asharov, Wei-Kai Lin, and Elaine Shi. 2022. Sorting Short Keys in Circuits of Size $o(n \log n)$. *SIAM J. Comput.* (2022).

[4] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of American Federation of Information Processing Societies (AFIPS).*

[5] Bruno Beauquier and Eric Darrot. 2002. On Arbitrary Size Waksman Networks and Their Vulnerability. *Parallel Processing Letters* (2002).

[6] Daniel J. Bernstein. 2020. Verified fast formulas for control bits for permutation networks. Cryptology ePrint Archive, Paper 2020/1493. https://eprint.iacr.org/2020/1493

[7] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al. 2017. Classic McEliece: conservative code-based cryptography. *NIST submissions* 1, 1 (2017).

[8] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. 2017. Prochlo: Strong Privacy for Analytics in the Crowd. In *Symposium on Operating Systems Principles (SOSP).*

[9] Feng Chen, Shuang Wang, Xiaoqian Jiang, Sijie Ding, Yao Lu, Jihoon Kim, S Cenk Sahinalp, Chisato Shimizu, Jane C Burns, Victoria J Wright, Eileen Png, Martin L Hibberd, David D Lloyd, Hai Yang, Amalio Telenti, Cinnamon S Bloss, Dov Fox, Kristin Lauter, and Lucila Ohno-Machado. 2016. PRINCESS: Privacy-protecting Rare disease International Network Collaboration via Encryption through Software guard extensionS. *Bioinformatics* (2016).

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy (EuroS&P).*

[11] Tung Chou. 2017. McBits revisited. In *International Conference on Cryptographic Hardware and Embedded Systems.*

[12] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained.* Technical Report.

[13] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. 2017. Privacy-Preserving Computation with Trusted Computing via Scramble-then-Compute. *Proceedings on Privacy Enhancing Technologies (PoPETs)* (2017).

[14] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP).*

[15] Kris Vestergaard Ebbesen. 2015. *On the Practicality of Data-oblivious Sorting.* Master's thesis. Aarhus Universitet, Datalogisk Institut.

[16] Saba Eskandarian and Dan Boneh. 2022. Clarion: Anonymous Communication from Multiparty Shuffling Protocols. In *Network and Distributed System Security Symposium, (NDSS).*

[17] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)* (1996).

[18] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2012. Practically Efficient Multi-Party Sorting Protocols from Comparison Sort Algorithms. In *Proceedings of the 15th International Conference on Information Security and Cryptology (ICISC).*

[19] William Holland, Olga Ohrimenko, and Anthony Wirth. 2022. Efficient Oblivious Permutation via the Waksman Network. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (AsiaCCS).*

[20] Intel. 2018. Q3 2018 Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html. Accessed August 2023.

[21] Intel. 2019. Intel Processors Voltage Settings Modification Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html. Accessed August 2023.

[22] Intel. 2020. 2020.2 IPU - Intel RAPL Interface Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html. Accessed August 2023.

[23] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proceedings of the VLDB Endowment* (2020).

[24] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security Symposium.*

[25] Kyungsook Yoon Lee. 1985. On the Rearrangeability of $2(\log_2 N) - 1$ Stage Permutation Networks. *IEEE Trans. Comput.* (1985).

[26] Kyungsook Yoon Lee. 1987. A New Benes Network Control Algorithm. *IEEE Trans. Comput.* (1987).

[27] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium.*

[28] Wei-Kai Lin and Elaine Shi. 2022. Optimal Sorting Circuits for Short Keys. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).*

[29] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (S&P).*

[30] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *IEEE Symposium on Security and Privacy (S&P).*

[31] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy (S&P).*

[32] David Nassimi and Sartaj Sahni. 1982. Parallel Algorithms to Set Up the Benes Permutation Network. *IEEE Trans. Comput.* (1982).

[33] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. 2014. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming (ICALP).*

[34] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium.*

[35] Nicholas Pippenger. 1993. Self-Routing Superconcentrators. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC).*

[36] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. 2020. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *USENIX Security Symposium.*

[37] A Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. 2023. Ruffle: Rapid 3-party shuffle protocols. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 3 (2023).

[38] Vijaya Ramachandran and Elaine Shi. 2021. Data Oblivious Algorithms for Multicores. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).*

[39] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium.*

[40] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Network and Distributed System Security Symposium (NDSS).*

[41] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[42] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. https://crysp.uwaterloo.ca/software/obliv/. Software artifact.

[43] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2023. Waks-On/Waks-Off: Fast Oblivious Offline/Online Shuffling and Sorting with Waksman Networks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[44] Sajin Sasy and Olga Ohrimenko. 2019. Oblivious Sampling Algorithms for Private Data Analysis. In *Advances in Neural Information Processing Systems (NeurIPS).*

[45] Nigel P Smart and Younes Talibi Alaoui. 2019. Distributing any Elliptic Curve Based Protocol. In *Proceedings of the 17th IMA International Conference on Cryptography and Coding (IMACC).*

[46] Afonso Tinoco, Sixiang Gao, and Elaine Shi. 2023. EnigMap: External-Memory Oblivious Map for Secure Enclaves. In *USENIX Security Symposium.*

[47] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium.*

[48] Abraham Waksman. 1968. A Permutation Network. *Journal of the ACM (JACM)* (1968).

[49] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P).*

[50] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square-Root ORAM: Efficient Random access in Multi-Party Computation. In *2016 IEEE Symposium on Security and Privacy (S&P).*

[51] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

A PROOFS

A.1 Correctness

The following are proofs of the theorems from Section 5.1.

Theorem 1. *Let P be an array, with $|P| = n$, containing the values $\{0, \ldots, n-1\}$ in some order. Let $Q = (0, \ldots, n-1)$. Let P' be a copy of P (as P' will get modified), and let $C =$ ControlBits$(P', 0)$. Then, if we run ApplyPerm(C, P), then $P = Q$ afterwards, and if instead we run ApplyInvPerm(C, Q), then again $P = Q$ afterwards.*

Proof. We first show that the inputs to each Waksman subnetwork, shifted right by d, contain a Complete Residue System (CRS). That is, for each call ControlBits(P, d), when either of the two calls ControlBits$(\hat{P}, d+1)$ are made, where $\hat{P}$ is a subarray of P and $|\hat{P}| = n'$, the values $\{x \gg d+1 | x \in \hat{P}\}$, taken as integers, form the set $\{0, \ldots, n'-1\}$. We recursively assume it is true for the ControlBits(P, d) call, and the base case (i.e., ControlBits$(P, 0)$) holds by the requirement that P is initially a permutation on $\{0, \ldots, n-1\}$.

In the InBits(P, d) call, CreateForwardLookup(P, d) inserts the values $(i, P[i] \gg d)$ into the forward lookup table F, with (n, n) added if n is odd, where $n = |P|$. Let $k = \lceil n/2 \rceil$. Then starting from element $f = k-1$, which is necessarily in input to the top subnetwork, the sequence is calculated that $f \longrightarrow g$ in F (i.e., $(f, g) \in F$), values g and r are associates (i.e., have the same value modulo k), $s \longrightarrow r$ under F, and f is updated to the partner of s. In each iteration, the switch that the initial f is an input to (if any) is already set to make $P[f]$ (i.e., g) an input to the top subnetwork. Therefore, to make a CRS to both subnetworks, the associate of $P[f]$ (i.e., r), which has the same value modulo k, must be sent to the bottom subnetwork. f is updated to be the input that shares a switch with r, and then that switch is set to send $P[f]$ to the top subnetwork and r to the bottom subnetwork, as required. If a cycle is encountered during this process, a new f can be arbitrarily chosen to be sent to the top subnetwork, and the process continues.

At the end, each pair of associates in P are sent to opposite subnetworks. In the case that n is odd, the element (n, n) in F can only be queried in the reverse direction with $r = n$ because the partner of n is $k - 1$, which is the initial selection for f. Therefore, this element will always end a cycle, and the value modulo k of $k - 1$ will always be sent to the top subnetwork. The top subnetwork thus obtains $\{0, \ldots, k - 1\}$ as inputs modulo k, and the bottom subnetwork obtains $\{0, \ldots, (n - k) - 1\}$ as inputs modulo k, giving each a CRS.

We next show that the outputs from each Waksman subnetwork, shifted right by d, are the sorted values $(0, \ldots, n - 1)$ (i.e., a sorted CRS). We recursively assume this claim is true for the outputs of the two calls ControlBits$(\hat{P}, d + 1)$, where $\hat{P}$ is a subarray of P.

When $n = 1$, we have already shown that the shifted input is 0, and there is no output switch applied, and so the claim holds in this case. When $n = 2$, we have already shown that the shifted inputs are $\{0, 1\}$, and in ControlBits the process of shifting left and applying an OR simply moves those values to the last bits. Then the first of these two values is used to set the control bit for the only output switch, which only flips the values if they appear in the order $(1, 0)$. Then bits are then moved back to the most significant position via a right shift and an addition, yielding the desired sorted output.

For $n > 2$, we use the recursive assumption that the output of each recursive ControlBits call is a sorted CRS. We have shown that the input to the current ControlBits call was a CRS, and therefore the least-significant bits of the outputs of the recursive ControlBits calls indicate if the value changed when reduced modulo k. Each output switch takes as inputs the outputs in the same position from the subnetworks, and because those are sorted they must have been partners modulo k. Therefore, one of their least-significant bits is a zero and the other is a one. The values are flipped only if the first input to the output switch has a least-significant bit of one. Thus, after applying all output switches, the first k outputs contain a CRS (ignoring the final $d + 1$ bits) in sorted order with least-significant bits of zero, and the remaining $n - k$ outputs contain a CRS (ignoring the final $d + 1$ bits) in sorted order with least-significant bits of one. The final right shifts and additions turns these values into a sorted CRS (ignoring the final d bits), as desired.

Thus, the control bits are set such that applying the resulting network to P sorts it. That is, with $Q = (0, \ldots, n - 1)$, after running ApplyPerm(C, P), $P = Q$. Because ApplyInvPerm simply applies the switches in the reverse order that ApplyPerm does, it applies the inverse permutation, and so, after instead running ApplyInvPerm(C, Q), $Q = P$. □

Theorem 2. *WaksShuffle(D) puts the items in D in a uniformly random order.*

Proof. WaksShuffle first creates the array $P = (0, \ldots, n - 1)$. Then, by assumption, the OShuffle(P) call puts P in a uniformly random order, which makes P a uniformly random permutation. By Theorem 2, the call to ControlBits$(P, 0)$ followed by ApplyPerm(C, D) permutes the items in D according to P. Therefore, WaksShuffle puts D in a uniformly random order. □

Theorem 3. *WaksSort(K, D) sorts the items in K in increasing order and also applies that permutation to D.*

Proof. WaksSort first creates the array $P = (0, \ldots, n - 1)$. Then, by assumption, the OSort(K, P) call sorts K and P such that the items in K are nondecreasing and the same permutation is applied to P. If, by sorting K, the key in position i in K is moved to position j in K, then the item in position i in P, which has value i, is moved to position j in P. That is, P now is such that $P[j] \rightarrow j$ is the permutation used to sort K, or, put another way, P, interpreted as the permutation $i \rightarrow P[i]$ contains the inverse of the permutation used to sort K. Therefore, by Theorem 2, the call to ControlBits$(P, 0)$ followed by ApplyInvPerm(C, D) permutes the items in D according to P. □

A.2 Obliviousness

The following are proofs of the theorems from Section 5.2.

Theorem 4. *ControlBits(P, d) is fully oblivious with respect to P.*

Proof. Let $n = |P|$ and $k = \lceil n/2 \rceil$. We will construct a simulator $\mathcal{S}$ that, for all P and d, and conditional on an actual output C (which $\mathcal{S}$ does not get), produces indistinguishable memory and execution traces given only n and d. Such simulators exist by assumption for the oblivious primitives we use, such as for OSelect, OSwap, OShuffle, and OSort. We recursively assume that such a simulator can be produced for smaller d. For $d \leq 2$, the required traces for ControlBits(P, d) can easily be produced because a deterministic sequence of operations is performed that depends only on n and d (note that InBits returns early in this case).

For $n > 2$, we can use our recursive assumption to simulate the traces for the recursive calls to ControlBits. Other than those calls and the call to InBits, the memory accesses and instructions executed by ControlBits depend only on n and d. Therefore we can focus on InBits(P, d), and we claim that it is fully oblivious with respect to P (but not d).

The obliviousness of InBits relies on the pseudorandomness of the PRP. We (conceptually) replace the calls to PRP with key κ_1 (i.e., those appearing in CreateForwardLookup and ForwardOrRand) with queries to a genuinely random permutation Π_1 on $(0, \ldots, 2k - 1)$ (or on any larger set containing $\{0, \ldots, 2k - 1\}$). The resulting executions are computationally indistinguishable by reduction to the pseudorandomness assumption. That is, otherwise an adversary could win the game defining pseudorandomness by simulating the execution while forwarding the PRP calls to the challenger, as the resulting executions would be distinguishable. Similarly, we replace the calls to PRP under κ_2 (i.e., those in CreateReverseLookup and sent to HashMapGet) with a genuinely random permutation Π_2 on $(0, \ldots, 2k - 1)$ (or on any larger set containing $\{0, \ldots, 2k - 1\}$).

In InBits, P is first used in CreateForwardLookup(P, d, κ). We claim that CreateForwardLookup is fully oblivious with respect to P and κ. Fix some output $F = (L, Q)$. The traces from creating and initially populating the arrays L and Q can be produced using the PRP simulator, which are indistinguishable even conditional on the values $L[i]$ that are fixed given F. The traces from the OSort(L, Q) call can be produced using its simulator, again even conditional on the values of L and Q, which are fixed given F. This proves the obliviousness claim for CreateForwardLookup.

We now consider CreateReverseLookup(F, κ), which is the next operation in InBits. We would *not* be able to say that CreateReverseLookup is fully oblivious to F and κ because its hash map insertions depend on them. However, $\mathcal{S}$ can nonetheless simulate the traces that result from the call to CreateReverseLookup, conditional on C, by choosing a random permutation (which is not necessarily the same as Π_2) and using its values on $(0, \ldots, n-1)$ (in order) as the values z used as keys in HashMapInsert$(R, z, (x, i))$. HashMapInsert is by assumption oblivious with respect to the value inserted (here, (x, i)), and so its simulator can be used to produce its traces. $\mathcal{S}$ also actually runs HashMapInsert, starting with an empty R and using the z values it has chosen, an arbitrary value for each x, and the i taking values $(0, \ldots, n-1)$ in order. Let z_i be the ith z value, which will be needed to produce traces for the hash map queries.

The next call in InBits is to CreateUnselectedCounts$(n, \emptyset)$. This function is deterministic and is only given n, and so its traces can easily be produced by $\mathcal{S}$.

InBits continues with a sequence of back-and-forth iterations. In each iteration, the traces from assignments, arithmetic computations, and OSelect calls are easily produced. The more involved arguments are for the calls to ForwardOrRand, HashMapGet, and DecUnselectedCounts. These arguments will use the following fact about the back-and-forth sequence: each of the n mappings under P (and also the dummy (n, n) mapping for odd n) is looked up only once and in either the forward or reverse direction (but not both). This fact would actually hold whenever partners and associates are defined to each produce a matching of the values $\{0, \ldots, k-1\}$.

Consider each call to ForwardOrRand(F, U, f, b, κ_1). Regardless of the permutation P, the control bits for P that we have sampled, and the previous traces that have been produced, each call looks like a random lookup of an unselected element. That is, given the items that have been looked up in the previous or forward and reverse lookups, a subsequent forward lookup looks like a binary search for a uniformly random unselected item in F. Note that this is true in part because F and R use different random permutations on the lookup values, Π_1 and Π_2, respectively. Therefore, because each ForwardOrRand call is performed on a different value f, its location in F is uncorrelated to a lookup of the same value in R, which is possible. However, it is not possible to do a forward lookup of a *mapping* (ignoring its direction) that has been looked up already in either the forward or reverse direction. Moreover, if ForwardOrRand is called with $b = 1$, then U ensures that a uniformly random is genuinely looked up. This fact holds even given P and C has been fixed, because although P determines when cycles end and C which mappings begin a new cycle, the location of the mappings is uniformly random at each step. Thus, the traces of each forward lookup can be generated by $\mathcal{S}$ as a uniformly random unselected item.

Now consider each HashMapGet$(R, \mathrm{PRP}(r, \kappa_2))$ call. It will yield a value s such that $s \longrightarrow r$ under the forward mapping. That mapping appears at some random unselected location i in F, and during the creation of R the reverse mapping was the ith insertion into R using a key z_i. We have already simulated that R and the z_i values, and so we can simulate the traces for HashMapGet given z_i. We can simulate choosing the correct z_i value simply by choosing a uniformly random index from among those containing unselected mappings. This simulation is accurate because, given P and C and the simulation so far, the next mapping is unselected and therefore appears in a uniformly random unselected item in F (or, put another way, Π_1 seen as an oracle is at that point forced to choose a value for an unqueried value). Then, once the simulator selects where it appears in F, the previous choices force the key z_i to use as HashMapGet(R, z_i). Now given both arguments, the traces from HashMapGet can be simulated by executing that function.

For the calls in InBits to DecUnselectedCounts(U, ℓ), we observe that the location ℓ can be computed from the traces (simulated or not) of the lookups preceding the DecUnselectedCounts call (forward or reverse). In the forward direction, ℓ is the final element accessed during the binary search, and, in the reverse direction, ℓ corresponds to the z_ℓ value that has been used to call HashMapGet. The creation of U has been simulated, and U is only modified by DecUnselectedCounts. Therefore, $\mathcal{S}$ can simulate the traces of each DecUnselectedCounts(U, ℓ) call by executing DecUnselectedCounts on the values for U and ℓ that $\mathcal{S}$ has effectively already chosen.

Finally, InBits calls OSort(S, C). The traces of this call, conditioned on C, can be sampled knowing only $|S|$ and $|C|$ using the OSort simulator. This finishes the trace simulation for InBits(P, d), proving the desired claim that it is fully oblivious with respect to P. □

Theorem 5. *ApplyPerm(C, D) and ApplyInvPerm(C, D) are fully oblivious with respect to C and D.*

Proof. ApplyPerm and ApplyInvPerm each perform a deterministic sequence of OSwap calls that only depend on $n = |D|$. Therefore, the simulator can produce the traces for these calls knowing only n, making use of the OSwap simulator, which is assumed to be oblivious to the values of its data and swap-bit inputs. □

Theorem 6. *WaksShuffle(D) is fully oblivious with respect to D.*

Proof. Let D' be the shuffled output of WaksShuffle, which does depend on some unknown random bits. D' is unknown to the simulator $\mathcal{S}$. $\mathcal{S}$ can produce the traces to create the initial P, and then for the OShuffle(P) call it uses an assumed simulator to produce its traces without knowing P or what order P must end up in, given D'. By Theorem 4, there exists a simulator that $\mathcal{S}$ can use to produce the traces for the ControlBits$(P, 0)$ call, not knowing the control bits C that must be produced, given D'. Finally, by Theorem 5, there exists a simulator that $\mathcal{S}$ can use to produce the traces for the ApplyPerm(C, D) call, knowing only $|C|$ and $|D|$. □

Theorem 7. *WaksSort(K, D) is fully oblivious with respect to K and D.*

Proof. Let K' and D' be the sorted outputs of WaksSort(K, D). K' and D' are unknown to the simulator $\mathcal{S}$. $\mathcal{S}$ can produce the traces to create the initial P, and then for the OSort(K, P) call it uses an assumed simulator to produce its traces without knowing K or P or what order K (and P) must end up in, given K'. By Theorem 4, there exists a simulator that $\mathcal{S}$ can use to produce the traces for the ControlBits$(P, 0)$ call, not knowing the control bits C that must be produced, given K'. Finally, by Theorem 5, there exists a simulator that $\mathcal{S}$ can use to produce the traces for the ApplyInvPerm(C, D) call, knowing only $|C|$ and $|D|$. □

A.3 Efficiency

The following are proofs of the theorems from Section 5.3.

Theorem 8. *Let $|P| = n$. The runtime of ControlBits$(P, 0)$ is $O(n \log^3 n)$.*

Proof. Let $k = \lceil n/2 \rceil$. Let the runtime of ControlBits$(P, 0)$ be $T(n)$.

We first consider the runtime of the call to InBits(P, d). Its call to CreateForwardLookup(P, d, κ_1) takes time $O(n \log^2 n)$ by assumption on the runtime of OSort. The CreateReverseLookup call takes time $O(n)$ by assumption on the $O(n)$ time to create an empty hash map on n items and the $O(1)$ amortized time for each HashMapInsert call. The CreateUnselectedCounts call takes $O(n)$ time by a straightforward inductive argument. Each ForwardOrRand call and DecUnselectedCounts call performs a binary search and thus takes $O(\log n)$ time, and each HashMapGet call takes $O(1)$ amortized time, and therefore the k back-and-forth iterations take $O(n \log n)$ total time. The final OSort(S, C) takes time $O(n \log^2 n)$ time by assumption on the runtime of OSort and because $|S| = |C| = k - 1$. Therefore the InBits call takes $O(n \log^2 n)$ time.

Subsequently, ControlBits applies input the $k-1$ input switches and reduces the n values in P modulo k, which together take $O(n)$ total time. The recursive calls to ControlBits take $T(k)$ and $T(n - k)$ time. Then, extracting the $n - k$ control bits takes $O(n)$ total time. Finally, applying the $n - k$ output switches and undoing the modulo reduction of P takes $O(n)$ total time.

Therefore,

$$T(n) = O(n \log^2 n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor).$$

For n a power of 2, we have that

$$T(n) = O(n \log^2 n) + 2T(n/2)$$

so by the Master theorem, $T(n) = O(n \log^3 n)$. If n is not a power of 2, let $N > n$ be the smallest power of 2 greater than n (so that $N < 2n$). Then by monoticity, $T(n) \leq T(N) = O(N \log^3 N) = O(2n \log^3 (2n)) = O(n \log^3 n)$. □

Theorem 9. *Let $|D| = n$, and let each item in D be of size β. The runtime of ApplyPerm(C, D) and ApplyInvPerm(C, D) is $O(\beta n \log n)$.*

Proof. Let $k = \lceil n/2 \rceil$, and let $T(n)$ be the runtime of ApplyPerm(C, D). ApplyPerm first executes $k - 1$ OSwap calls, which by assumption each take $O(\beta)$ time. Then it calls ApplyPerm on k items and on $n - k$ items. Finally, it executes $n - k$ OSwap calls, each of which take $O(\beta)$ time. Therefore,

$$T(n) = O(\beta n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor).$$

The same reasoning as in the proof of Theorem 8 above yields that $T(n) = O(\beta n \log n)$, as required. The same argument applies to ApplyInvPerm. □

Theorem 10. *Let $|D| = n$, and let each item in D be of size β. The offline runtime of WaksShuffle(D) is $O(n \log^3 n)$, and the online runtime of WaksShuffle(D) is $O(\beta n \log n)$.*

Proof. In the offline phase of WaksShuffle, $O(n)$ time is required to create P, and by Theorem 8, $O(n \log^3 n)$ time is used by ControlBits. Therefore, the offline runtime is $O(n \log^3 n)$. In the online phase, by Theorem 9, $O(\beta n \log n)$ time is used by ApplyPerm. □

Theorem 11. *Let $|K| = n$, $|D| = n$, each item in K be of size α, and each item in D be of size β. The (online) runtime of WaksSort(K, D) is $O(n \log^3 n + \alpha n \log^2 n + \beta n \log n)$.*

Proof. WaksSort is entirely online. $O(n)$ time is required to create P, $O(\alpha n \log^2 n)$ time is assumed to be used by OSort(K, P), ControlBits uses $O(n \log^3 n)$ time by Theorem 8, and ApplyInvPerm uses $O(\beta n \log n)$ time by Theorem 9. Therefore the (online) runtime of WaksSort is $O(n \log^3 n + \alpha n \log^2 n + \beta n \log n)$. □

B IMPLEMENTATION OPTIMIZATIONS

We refine our implementation of WaksShuffle and WaksSort with several optimizations. To ensure fair comparisons we apply matching optimizations (whenever applicable as detailed below) for our Nassimi-Sahni implementation. First, in the SGX context performing memory allocations are expensive since they require an Asynchronous Enclave eXit (AEX) operation [12]. Hence we perform a one-time memory allocation procedure that allocates memory for all the Waksman subnetworks upfront with one AEX, rather than $n \log n$ recursive memory allocations. We implement this optimization for ControlBits of our Waksman algorithms and for Nassimi-Sahni as well.

For ControlBits, we instantiate both the forward and reverse lookup tables with standard C++ hash maps, and the PRP function with AES-ECB. Since the PRP has to generate unique random labels for the repeated switch number inputs it receives from different Waksman subnetworks, in our implementation every Waksman subnetwork is assigned a unique id. This subnetwork id serves as the high 64 bits of the input to our PRP, and the switch number occupies the low 64 bits, ensuring unique random labels for the repeated inputs from different Waksman subnetworks.

Since our PRP relies on AES-ECB and uses the same key for all PRP operations, we optimize our PRP usage by maintaining the AES expanded key state and using it to perform PRPs as inputs to permute are available, instead of repeating the key expansion step for each PRP operation individually. Note that this optimization does not extend to the Nassimi-Sahni algorithm as it does not make use of a PRP. Finally to pick a random integer from a range, needed for ForwardOrRand in InBits, we use the simple technique we detailed in Section 3.1.

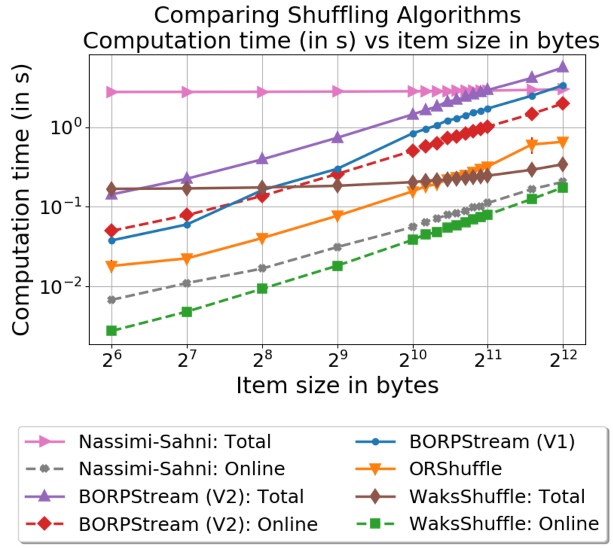


Figure 18: Comparing time taken for shuffling against item size with $n = 2^{15}$ items. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.

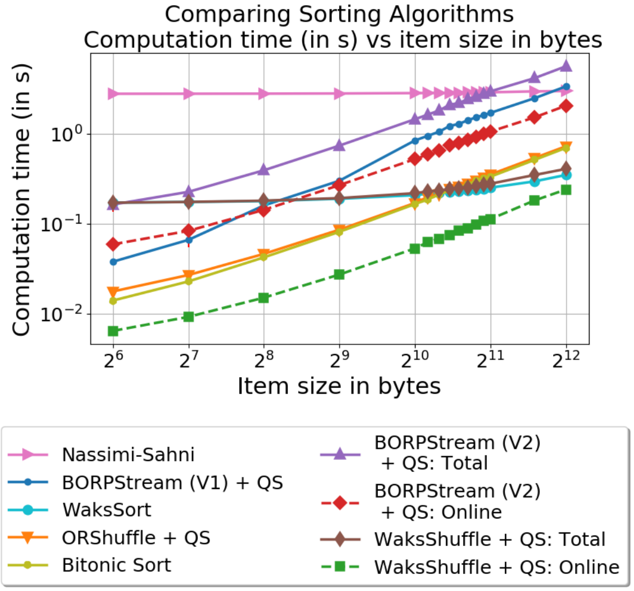


Figure 19: Comparing time taken for sorting against item size with $n = 2^{15}$ items. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.

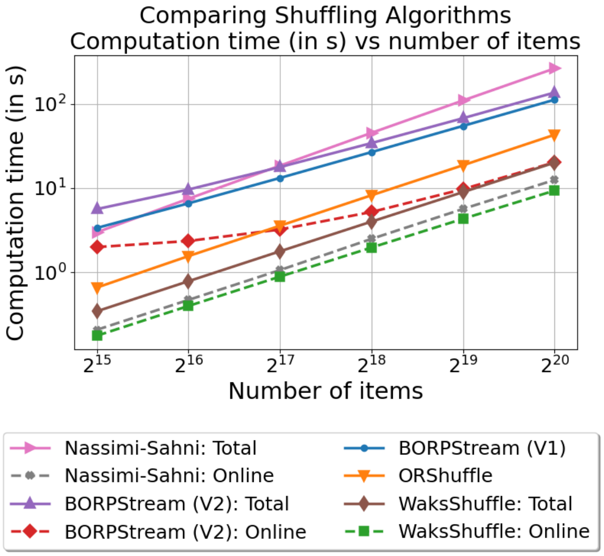


Figure 20: Comparing time taken for shuffling against number of items with an item size of $\beta = 2^{12}$ bytes. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.

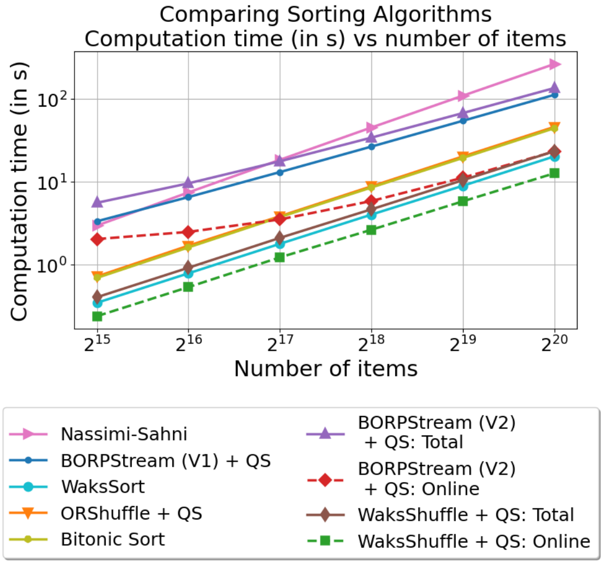
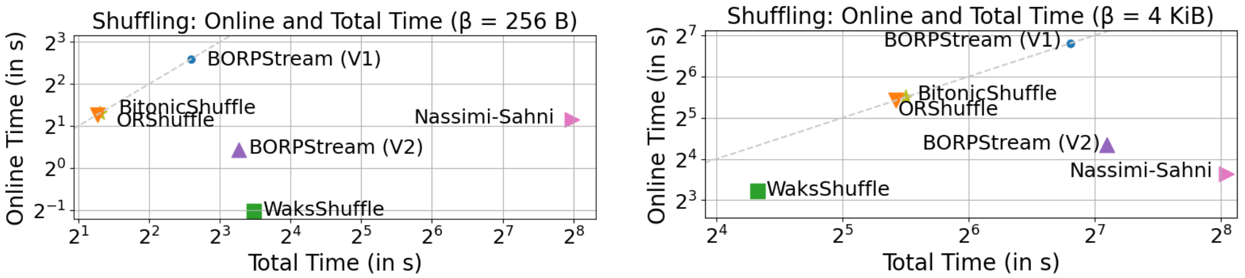


Figure 21: Comparing time taken for sorting against number of items with an item size of $\beta = 2^{12}$ bytes. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.



(a) Shuffling with a small item size (256 bytes).

(b) Shuffling with a large item size (4096 bytes).

Figure 22: Scatter plots comparing online and total time taken (in s) for shuffling algorithms with $n = 2^{20}$. WaksShuffle consistently incurs the least online time irrespective of item size. For large enough item size ($\beta > 1400$ B), the total time of WaksShuffle is lower than all other fully oblivious shuffling algorithms.

C ADDITIONAL EXPERIMENTAL RESULTS

C.1 Varying the Item Size

Figures 18 and 19 show shuffle and sort times for varying item sizes and $n = 2^{15}$ items. We observe qualitatively similar results to those for $n = 2^{20}$ items. In particular, we see the crossover points, at which the total times for WaksShuffle and WaksSort become the lowest, are still at an item size of about $\beta = 1400$ bytes. We also see that the online times for WaksShuffle and WaksShuffle+QS remain the lowest.

C.2 Varying the Number of Items

Figures 20 and 21 show the shuffle and sort times for varying numbers of items and an item size of $\beta = 2^{12}$ bytes. For all n tested, WaksShuffle and WaksSort have the lowest total times for shuffling and sorting, respectively. Similarly, WaksShuffle and WaksShuffle+QS always have the lowest online times for shuffling and sorting, respectively.

C.3 Scatter Plots for Oblivious Shuffling

In Section 7.3, we presented plots showing the online and total times of oblivious sorting algorithms within TEEs. Here, we do the same for oblivious shuffling algorithms, and see an analogous result. Figure 22a shows shuffling algorithms with items of size 256 B, and Figure 22b shows 4 KiB items. Notably, WaksShuffle always incurs the least online overhead for shuffling items irrespective of the item size β; for large enough problem sizes ($\beta > 1400$ B) the total cost of WaksShuffle is lower than that of all other comparable shuffling algorithms.