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 circumventing 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 shuffling 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× and > 3× speedups), when permuting \(n\) items each of size \(\beta\).

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., \(\beta > 1400\) B), and the performance gap only widens as the item sizes increase. Furthermore, WaksShuffle improves the online cost of oblivious shuffling by > 5× for shuffling \(2^{20}\) items of any size; similarly WaksShuffle+QS, our other sorting algorithm, provides > 2.7× speedups in the online cost of oblivious sorting.

CCS CONCEPTS
• Security and privacy → Domain-specific security and privacy architectures.

KEYWORDS
oblivious algorithms; trusted execution environments; SGX

ACM Reference Format:

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 [29, 45], telemetry [8], collaborative machine learning [33], privacy-preserving statistics [43], file systems [13], database queries [22], and video analytics [35], 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, 46] and voltage variations [28, 30], and Intel has issued patches to mitigate them [19–21]. In contrast, memory and control-flow side-channel attacks [23, 26, 48] are out of scope of the SGX security model.

However, some defenses have been developed against such side channels [38, 39]. Sasy et al. [40] 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

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

\(^{An}\) extended version of this paper is available. [42]
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/onlinemodel, 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/onlinemodel can be useful to reduce query latency for a system that answers queries on existing data [13, 43]. 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.

Therefore, in this work we present novel fully oblivious shuffling and sorting algorithms in the offline/onlinemodel. 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 phase 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 [33], where the items to shuffle may be images, and database queries [22], where the items to sort may be entire rows of database tables. Therefore, even without the offline/onlinesplit, 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 [47]. 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 $O(\beta n \log n)$ cost of permuting $n$ items each of size $\beta$ 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 [12, 17], 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 [47].

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 $\beta > 1400$ 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/onlinemodel, 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 [47] 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 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 [47] 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 [31] 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 [14], the Bernstein algorithm has runtime $O(n \log^4(n))$.

Lee [24, 25] 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 [22, 40, 45], we adopt full obliviousness [40] 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 [23], 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. [40]. Let an algorithm $A$ be a sequence of instructions with direct or indirect memory arguments. Let $E(A, x_1, \ldots, x_k)$ 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 $E$ is a random function. Let $E \in \mathcal{E}(A, x_1, \ldots, x_k)$ be a random execution. We denote its output as $O(E)$. We denote its instruction trace as $I(E)$, 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 $M(E)$, which consists of a sequence of pairs $(b, \ell)$, one for each executed instruction, with $b \in \{0, 1\}$ indicating a read or write and $\ell$ 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 [40], with $|k| = \{1, \ldots, k\}$:

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

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 $(A[i], A[i+1], \ldots, A[j])$, 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 $\lambda$ 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 $\llbracket c \rrbracket$ (e.g., $\llbracket x < y \rrbracket$ 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 $\beta$, is assumed to have runtime $O(\beta)$. These primitives can be implemented efficiently and obliviously using boolean arithmetic or conditional instructions such as CMVNZ.

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 $x^{2-\sigma}$ from uniform is to select $\sigma$ random bits $p$ and compute $xp \gg \sigma$. We also require a pseudo-random 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.

### 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.

### 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 [25] 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).
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 ± k are partners). Each permutation value \( v \) has \( P[i] \gg d \) 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,
Figure 4: InBits$(P, d)$: Compute Waksman control bits for input
layer of switches for permutation $P$ at recursion level $d$.

1. \( n \leftarrow |P| \)
2. \( k \leftarrow \lfloor n/2 \rfloor \)
3. Create array $C$ of length $k - 1$. \( \triangleright \) Control bits
4. if $n \leq 2$ then return $C$
6. Create array $S$ of length $k - 1$. \( \triangleright \) Switch numbers of control bits
7. Generate $\lambda$ random bits $k_1$. \( \triangleright \) PRP key for forward lookup
8. Generate $\lambda$ random bits $k_2$. \( \triangleright \) PRP key for reverse lookup
9. $F \leftarrow \text{CreateForwardLookup}(P, d, k_1)$
10. $R \leftarrow \text{CreateReverseLookup}(F, k_2)$
11. $U \leftarrow \text{CreateUnselectedCounts}(n, 0)$
12. \( \triangleright \) Perform initial back-and-forth with fixed input $k - 1$
13. $f, g, \ell \leftarrow \text{ForwardORRand}(F, U, k - 1, 0, k_1)$ \( \triangleright \) Map $k - 1$
14. $d \text{UnselectedCounts}(U, \ell) \leftarrow \text{Mark } k - 1 \rightarrow g \text{ as selected}$
15. $c \leftarrow f$ \( \triangleright \) Cycle start is $f$ (which contains $k - 1$)
16. $r \leftarrow g + \text{OSelect}(\kappa$, $k, [g < k])$ \( \triangleright \) $r$ is the associate of $g$
17. $s, \ell \leftarrow \text{HashMapGet}(R, \text{PRP}(r, k_2))$ \( \triangleright \) Reverse map of $r$
18. $d \text{UnselectedCounts}(U, \ell) \leftarrow \text{Mark } s \rightarrow r \text{ as map as selected}$
19. $f \leftarrow s + \text{OSelect}(\kappa$, $k, [s < k])$ \( \triangleright \) $f$ is the partner of $s$
20. \( \triangleright \) Repeat the back-and-forth process to compute the control bits
21. for $i \leftarrow 0, \ldots, k - 2$ do
22. \( b \leftarrow [f = c] \) \( \triangleright \) Indicate if current cycle has ended
23. if $b$, choose random unselected $f \rightarrow g \text{ map, else map } f$
24. $f, g, \ell \leftarrow \text{ForwardORRand}(F, U, f, b, k_1)$
25. $d \text{UnselectedCounts}(U, \ell) \leftarrow \text{Mark } f \rightarrow g \text{ as selected}$
26. $c \leftarrow \text{OSelect}(c, f, b) \) (If new cycle begin, update cycle start)
27. $r \leftarrow g + \text{OSelect}(\kappa$, $k, [g < k])$ \( \triangleright \) $r$ is the associate of $g$
28. $s, \ell \leftarrow \text{HashMapGet}(R, \text{PRP}(r, k_2))$ \( \triangleright \) Reverse map of $r$
29. $d \text{UnselectedCounts}(U, \ell) \leftarrow \text{Mark } s \rightarrow r \text{ as selected}$
30. $C[i] \leftarrow [f \geq k] \) \( \triangleright \) Control bit to put $f$ in top subnetwork
31. $S[i] \leftarrow f - \text{OSelect}(0, k, [f \geq k])$ \( \triangleright \) Switch number of $f$
32. $f \leftarrow s + \text{OSelect}(\kappa$, $k, [s < k])$ \( \triangleright \) $f$ is the partner of $s$
33. $\text{OSort}(S, C)$ \( \triangleright \) Order control bits by increasing switch number
34. return $C$

1. $n \leftarrow |P|$
2. $k \leftarrow \lfloor n/2 \rfloor$
3. $m \leftarrow 2k$
4. Create array $L$ of length $m$. \( \triangleright \) Random labels for mappings
5. for $i \leftarrow 0, \ldots, m - 1$ do
6. $L[i] \leftarrow \text{PRP}(i, k)$
7. Create array $Q$ of length $m$. \( \triangleright \) Permutation mappings
8. for $i \leftarrow 0, \ldots, n - 1$ do
9. $Q[i] \leftarrow (i, P[i] \gg d)$
10. if $n \neq m$ then \( \triangleright \) Add dummy mapping to make total size even
11. $Q[n] \leftarrow (n, n)$
12. $\text{OSort}(L, Q)$ \( \triangleright \) Randomize order of mappings in $Q$ by sorting on $L$
13. return $(L, Q)$

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

1. $n \leftarrow |P|$
2. $k \leftarrow \lfloor n/2 \rfloor$
3. $m \leftarrow 2k$
4. Create array $L$ of length $m$. \( \triangleright \) Random labels for mappings
5. for $i \leftarrow 0, \ldots, m - 1$ do
6. $L[i] \leftarrow \text{PRP}(i, k)$
7. Create array $Q$ of length $m$. \( \triangleright \) Permutation mappings
8. for $i \leftarrow 0, \ldots, n - 1$ do
9. $Q[i] \leftarrow (i, P[i] \gg d)$
10. if $n \neq m$ then \( \triangleright \) Add dummy mapping to make total size even
11. $Q[n] \leftarrow (n, n)$
12. $\text{OSort}(L, Q)$ \( \triangleright \) Randomize order of mappings in $Q$ by sorting on $L$
13. return $(L, Q)$

Figure 6: ForwardORRand$(F, U, f, b, k)$: If $b = 0$, look up forward
map from $f$ under permutation encoded in $F$ under key $k$. 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 $\ell$.

1. $L, Q \leftarrow F$
2. $n \leftarrow |L|$
3. $h \leftarrow \text{PRP}(f, k)$ \( \triangleright \) Label to search for in $L$
4. $i \leftarrow 0$ \( \triangleright \) Start index of range in $L$ and $U$ to search
5. $j \leftarrow n - 1$ \( \triangleright \) End index of range in $L$ and $U$ to search
6. $u \leftarrow U[n - 1]$ \( \triangleright \) Number of unselected items in search range
7. $\rho \leftarrow \text{Rand}(u)$ \( \triangleright \) Random unselected mapping
8. while true do \( \triangleright \) Binary search for label $h$ or unselected-item $\rho$
9. $\ell \leftarrow [(i + j)/2]$ \( \triangleright \) Search direction
10. if $i = j$ then \( \triangleright \) Continue search in first half of range
11. break
12. $z \leftarrow \text{OSelect}(L[\ell] \leq h, [U[\ell] \leq \rho], b)$ \( \triangleright \) Search direction
13. if $z = 0$ then \( \triangleright \) Continue search in last half of range
14. $j \leftarrow \ell$
15. $u \leftarrow U[\ell]$
16. else \( \triangleright \) Continue search in last half of range
17. $i \leftarrow \ell + 1$
18. $u \leftarrow u - U[\ell]$
19. $\rho \leftarrow \rho - U[\ell]$
20. $f, g \leftarrow Q[\ell]$
21. return $(f, g, \ell)$

labels. This structure allows each mapping to be looked up via a bi-
ary search on the labels, which is performed by ForwardORRand
(Figure 6). For the reverse map, we simply store the $n$ inverse per-
mutation 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 $F$.
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 for a complication that arises when a back-and-forth cycle ends. Starting with input $s_1$, the associate $w_1$ is 1 if $x = s_1$, and $z_1$ is 0 if $w = s_1$. The bold values emphasize which components of a network contain 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 ComputeControlBits. The notation $x : y \rightarrow_k w : yz$ 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 $w : yz \rightarrow_{k'} x : y$ 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 $z$ indicates if $x$ changed under this reduction.

We follow Figure 10 from left to right to observe the actions of ComputeControlBits. It initially uses InBits to compute control bits for the first set of input switches. The modulus it uses is $k = [9/2] = 5$. 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:

3.4 Worked Example of ComputeControlBits

Figure 10 illustrates an example of running our ComputeControlBits 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 ComputeControlBits. The notation $x : y \rightarrow_k w : yz$ 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 $w : yz \rightarrow_{k'} x : y$ 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 $z$ indicates if $x$ changed under this reduction.

We follow Figure 10 from left to right to observe the actions of ComputeControlBits. It initially uses InBits to compute control bits for the first set of input switches. The modulus it uses is $k = [9/2] = 5$. 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:
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 \texttt{ControlBits}.

(1) Set \( g = 3, r = 8, s = 6 \). Set the control bit of the switch containing input \( f = 2 \) (and input 7) to \([f \geq k] = [2 \geq 5] = 0\), 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 \([1 \geq 5] = 0 \). 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 \([8 \geq 5] = 1 \). Set \( f = 4 \).

(4) The cycle has ended because \( f = 4 \) was the initial value.

The remaining mappings are \( 0 \rightarrow 6 \) and \( 5 \rightarrow 1 \), and \texttt{ForwardRand} randomly selects \( 0 \rightarrow 6 \) 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 \([0 \geq 5] = 0 \).

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 (CRS) modulo 5, and the reduced values for the bottom subnetwork (i.e., network E) is 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.

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 \texttt{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 \texttt{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 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|.

```plaintext
1: n ← |D|
2: if n < 2 then return
3: C_in, C_top, C_bot, C_out ← C
4: k ← ⌈n/2⌉
5: if n > 2 then
6: for i ← 0, ..., n − k − 1 do
7: OSwap(D, i, k + i, C_in[i])  // Apply input-layer switches
8: Recursively apply switches in top and bottom subnetworks
9: ApplyPerm(C_top, D[0..k − 1])
10: ApplyPerm(C_bot, D[k..n − 1])
11: for i ← 0, ..., n − k − 1 do
12: OSwap(D, i, k + i, C_out[i])  // Apply output-layer switches
13: OSwap(D, i, k + i, C_in[i])
```

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|.

```plaintext
1: n ← |D|
2: if n < 2 then return
3: C_in, C_top, C_bot, C_out ← C
4: k ← ⌈n/2⌉
5: if n > 2 then
6: for i ← 0, ..., n − k − 1 do
7: OSwap(D, i, k + i, C_out[i])  // Apply output-layer switches
8: Recursively apply switches in top and bottom subnetworks
9: ApplyInvPerm(C_top, D[0..k − 1])
10: ApplyInvPerm(C_bot, D[k..n − 1])
11: for i ← 0, ..., n − k − 2 do
12: OSwap(D, i, k + i, C_in[i])  // Apply input-layer switches
13: OSwap(D, i, k + i, C_out[i])
```

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, ..., 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, ..., 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 [12]. Also, even with the added cost of non-oblivious sorting, the online phase may be faster than existing oblivious sorting algorithms because both $\text{APPLYPERM}$ and quicksort have $O(\beta n \log n)$ runtime (where $\beta$ is the size of each data item) with small constants.

5 ANALYSIS

We show that $\text{WaksShuffle}$ and $\text{WaksSort}$ are correct and fully oblivious. By way of doing that, we demonstrate these properties for $\text{CONTROLBITS}$, $\text{APPLYPERM}$, and $\text{APPLYINVPERM}$. 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 of the extended version of this paper [42].

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. 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 $C = \text{CONTROLBITS}(P', 0)$. Then, if we run $\text{APPLYPERM}(C, P)$, then $P = Q$ afterwards, and if instead we run $\text{APPLYINVPERM}(C, Q)$, then again $P = Q$ afterwards.

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

Theorem 2. $\text{WaksShuffle}(D)$ puts the items in $D$ in a uniformly random order.

Theorem 3. $\text{WaksSort}(K, D)$ 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. $\text{CONTROLBITS}(P, d)$ 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 $\text{FORWARDORRAND}$ 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. $\text{CONTROLBITS}(P, d)$ is fully oblivious with respect to $P$.

Theorem 5 shows that $\text{APPLYPERM}(C, D)$ and $\text{APPLYINVPERM}(C, D)$ 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. $\text{APPLYPERM}(C, D)$ and $\text{APPLYINVPERM}(C, D)$ are fully oblivious with respect to $C$ and $D$.

Theorems 6 and 7 show that $\text{WaksShuffle}$ and $\text{WaksSort}$ 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. $\text{WaksShuffle}(D)$ is fully oblivious with respect to $D$.

Theorem 7. $\text{WaksSort}(K, D)$ 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 $\beta$ (or for sorting, where a key and data item have total size $\beta$), 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 [40]) and sorting algorithm (bitonic sort [4, 14]) that are fastest in practice have a $O(\beta n \log^2 n)$ runtime. We use these as our $\text{OSHUFFLE}$ and $\text{OSORT}$ 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 [31], as made oblivious by Bernstein [6], which has a runtime of $O(n \log^3 n)$.

Theorem 8. $\text{WaksShuffle}(D)$ is $O(n \log^3 n)$.

The oblivious sorts in $\text{CONTROLBITS}$ 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 $\text{CONTROLBITS}$, and in each recursive call an $\text{OSORT}$ is performed on $2k$ (fixed-size, independent of the item size $\beta$) elements to create the forward lookup table, and another $\text{OSORT}$ 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 $\text{FORWARDORRAND}$ and $2k$ calls to $\text{DECSSELECTEDCOUNTS}$, where each of those calls performs an $O(\log n)$ binary search. If $\text{OSORT}$ were implemented with a $O(n \log n)$ algorithm, then $\text{CONTROLBITS}$ would have a $O(n \log^5 n)$ runtime.

Theorem 9 shows that, for data items of size $\beta$ each, the runtimes of $\text{APPLYPERM}$ and $\text{APPLYINVPERM}$ are both $O(\beta n \log n)$.

Theorem 9. Let $|D| = n$, and let each item in $D$ be of size $\beta$. The runtime of $\text{APPLYPERM}(C, D)$ and $\text{APPLYINVPERM}(C, D)$ is $O(\beta n \log n)$.

The expensive operations in both algorithms are applying the switches to the data items, and each resulting $\text{OSWAP}$ 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 $\beta$ is $O(\beta n \log n)$.

**Theorem 10.** Let $|D| = n$, and let each item in $D$ be of size $\beta$. 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 $\beta$ factor only appears in the online phase, and so if $\beta$ 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 ORShuffle. We can estimate the potential speedup in the online phase compared to the practically fastest ORShuffle [40] (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 5.3x.

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

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

All of WaksSort is online because the keys are needed to determine the permutation used to set the Waksman control bits. The $O(n \log^2 n)$ 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 $an \log^2 n$ term is from applying OSORT to the keys $K$, and it would similarly reduce to $O(an \log n)$ with a $O(n \log n)$ OSORT. The $\beta n \log n$ term is from APPLYINVRPerm. Note that if the keys are small relative to the items (i.e., $a = O(\beta \log n)$), 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 $O(n \log^2 n)$, better than the $O(n \log^3 n)$ 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 $2(\log_2 n) - 1$ parallel steps. The online phase of WaksShuffle has the same step complexity. Thus WaksShuffle improves on the $(\log_2 n + 1)/2$ step complexity of ORShuffle by a factor of a little more than $(\log_2 n + 1)/4$. For $n = 2^{20}$ items, for example, the speedup in online step complexity would be about 5.3x.

### 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 [13, 22, 29, 33]) and bitonic shuffle for shuffling. Our implementation builds on top of the software artifact of Sasy et al. for oblivious shuffling algorithms in TEEs [40, 41], 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 [31]. 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 [47], 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. [40], 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 of the extended version of this paper [42]. 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 $\beta$ 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 of the extended version of this paper [42]. In the interest of space, we do not present them all here as the results are qualitatively identical.

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 (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 3.5x and 2.2x 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 [22] or oblivious file systems [13]), 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
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 ($\beta > 1400\ B$), the total times of WaksSort and WaksShuffle+QS are lower than all other fully oblivious sorting algorithms.

speedups of $11.5\times$ and $12\times$ respectively for 64 B and 4 KiB item sizes. Similarly, WaksShuffle demonstrates a $4.6\sim5.6\times$ 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 in- 
curs a $O(n \log^3 n)$ cost and has higher hidden constants than the ControlBits of WaksShuffle, due to the larger number of obli-
vorous 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 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 of the extended version of this paper [42]).

In terms of the online cost, we intentionally set up the Nassimi-
Sahni implementation with the standard Waksman network topo-
ylogy to demonstrate the performance benefits of our topology.\footnote{Instantiating Nassimi-Sahni with our topology would result in an online time matching our algorithm and a negligible change in its total time.} We observe that the performance improvements from our topology are larger for smaller items, yielding up to a $5.3\times$ improvement for 64 B items and tapering down to a $1.3\times$ 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 $1.8\sim2.2\times$ speedup over BORPStream’s online time to sort. If all the items to sort are available simultaneously, then the best competing algorithm in fact Bitonic Sort, over which the online time of WaksShuffle+QS provides a $2.7\sim3.5\times$ 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 of the extended version of this paper [42]. 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 ($\beta > 1400\ B$) 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, 32], 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 [23]. We refer the interested reader to Sasy et al. [40, §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 [14] 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 [13, 22, 29, 33, 35, 50]. Recently, data-oblivious sorting algorithms have received renewed interest [3, 27, 37]. 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 [34], both of which have large constants hidden in the asymptotics that make them impractical.

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. [18] 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 [15, 36], 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. [49] revisit the original square-root ORAM by Goldreich and Ostrovsky [16] and leverage Waksman networks to replace the oblivious sorting primitive for efficiency improvements within the ORAM. Smart and Aloufi [44] 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 [47], 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 CONTROL-Bits 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
