Data-Oblivious Data Structures

John C. Mitchell and Joe Zimmerman

Department of Computer Science, Stanford University, Stanford, US
{mitchell,jzim}@cs.stanford.edu

Abstract

An algorithm is called *data-oblivious* if its control flow and memory access pattern do not depend on its input data. Data-oblivious algorithms play a significant role in secure cloud computing, since programs that are run on secret data—as in fully homomorphic encryption or secure multi-party computation—must be data-oblivious. In this paper, we formalize three definitions of data-obliviousness that have appeared implicitly in the literature, explore their implications, and show separations. We observe that data-oblivious algorithms often compose well when viewed as data structures. Using this approach, we construct data-oblivious stacks, queues, and priority queues that are considerably simpler than existing constructions, as well as improving constant factors. We also establish a new upper bound for oblivious data compaction, and use this result to show that an "offline" variant of the Oblivious RAM problem can be solved with $O(\log n \log\log n)$ expected amortized time per operation— as compared with $O(\log^2 n/\log\log n)$, the best known upper bound for the standard online formulation.

1998 ACM Subject Classification D.4.6 Security and Protection, E.1 Data Structures, F.1.1 Models of Computation, F.1.2 Modes of Computation

Keywords and phrases Data-oblivious algorithms, Data-oblivious data structures, Oblivious RAM, Secure multi-party computation, Secure cloud computing

Digital Object Identifier 10.4230/LIPIcs.STACS.2014.554

1 Introduction

An algorithm is called *data-oblivious*, or just *oblivious*, if its control flow and memory access pattern do not depend on its input data. Data-obliviousness has compelling applications in cryptography and security. When an untrusted cloud server computes on encrypted data, the computation must be oblivious, since by assumption the server cannot learn anything about the data. Obliviousness also subsumes standard notions of security against side-channel attacks. If an adversary cannot distinguish between different primitive operations, then by making a program oblivious, we ensure that the adversary gains no information from any other source—including standard side channels such as timing, cache performance, and termination behavior. In addition, deterministic oblivious algorithms correspond directly to uniform families of Boolean circuits, and so they are useful in designing hardware implementations whose structure must be fixed before the input is known.

There has been a large body of work on the Oblivious RAM problem, which concerns a general translation of RAM programs to versions that are data-oblivious in a certain sense [7, 15, 10, 17, 6], as well as on other general data-oblivious simulations [9, 16]. In addition, there has been some work on developing data-oblivious solutions to specific problems [4, 8], but this class of questions has received relatively little attention in the past, particularly with respect to the prospect of composable data-oblivious data structures. In this work:

- We formalize three definitions of data-obliviousness—including formulations that apply to modern cloud computing settings, in which algorithms need only be oblivious with respect to part of the data.

31st Symposium on Theoretical Aspects of Computer Science (STACS'14).
Editors: Ernst W. Mayr and Natacha Portier; pp. 554–565

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


- We give new results on data-oblivious algorithms and simple composable data structures, including new bounds for data compaction and an "offline" variant of Oblivious RAM.
- We give a number of refinements of existing results, including simpler and tighter constructions for oblivious stacks, queues, and priority queues.

2 The formal computation model

We will consider algorithms executed on a word RAM, with a word size of $\Theta(\log n)$, where n is the size of the input (or the capacity of the data structure, as appropriate), and the entire memory consists of $\text{poly}(n)$ words.[1] The RAM has a constant number of public and secret registers, and can perform arbitrary operations (of circuit size $\text{poly}(\log n)$) on a constant number of registers in constant time. (Data-oblivious algorithms are sometimes also studied in the external memory model. In the language of that model, our results assume a block size of $B = \Theta(\log n)$, and a cache of a constant number of blocks, $M = O(B)$; this is the traditional setting of Oblivious RAM [7].)

We will further assume that the machine's memory is divided into *public* and *secret* regions, and that an operation is prohibited if it could cause values originating from secret regions to influence those in public regions. The formal mechanism for this restriction depends on the machine specification, but in general, we assume standard techniques from (branching-sensitive) static information flow control [14, 13]. For example, we may specify that (i) a register is tagged as secret when its value is read from a secret memory location or is updated based on another secret register; (ii) a register does not become public again until it is directly overwritten by a constant or the contents of a public memory location; and (iii) that no secret register may be involved in a write instruction to a public memory location. The notion of obliviousness is then parameterized over which part of the input is secret: we say that an algorithm is oblivious if it never executes any instruction that sets its program counter or memory-address operands based on a secret register. Thus, if the entire memory is designated as secret, we recover the standard definition of oblivious computation.

We will also assume that each of the registers and memory locations is labeled as *deterministic* or *nondeterministic*, with the restriction that values labeled *nondeterministic* can never influence those labeled *deterministic*.[2] We assume that machines have a finite number of registers of every type (public/deterministic, public/nondeterministic, secret/deterministic, secret/nondeterministic), and we will measure memory usage of all types together. When machines have access to randomness, we will say that they are provided with two read-only one-way-infinite tapes of uniformly random bits, with one labeled public and one labeled secret, and both labeled nondeterministic.

We can now state three definitions, in increasing order of computational power.

1. **Deterministically data-independent**: *The RAM can only set its control flow based on registers that are both public and deterministic.* This definition corresponds directly to uniform families of Boolean circuits.
2. **Data-independent**: *The RAM can only set its control flow based on registers that are public (though these registers may be either deterministic or nondeterministic).* Even

[1] Our choice of such a powerful machine model is motivated by the thin asymptotic complexity margins inherent in the study of obliviousness. As we note in Section 2.1, any program can be made oblivious with overhead linear in its time complexity, and thus it would not make sense to study a simpler model such as the Turing machine, which may already incur linear overhead just by virtue of being limited to sequential memory access.

[2] As above, this restriction can be effected using standard static information-flow control rules.

though its structure varies based on random bits, the computation can still proceed without seeing the secret data.

3. **Data-oblivious**: *The RAM can only set its control flow based on public registers, but it is equipped with an additional* "declassification" operation *that moves a value from any secret register to any public register. However, the distribution of all declassified values (along with the point during execution at which each declassification occurs) is independent*[3] *of all secret inputs.* Here, the algorithm can see the secret data and branch on it, but its control flow and memory access pattern still must not reveal any secret values. This definition captures the setting of Oblivious RAM with constant client-side storage [7, 10, 3]; the physical memory locations accessed by the Oblivious RAM protocol must be independent of the logical memory locations requested.

Qualitatively, the fundamental difference between these definitions is that *data-independent* machines must satisfy a purely static constraint on the kinds of operations that are permitted on secret locations, while *data-oblivious* machines must only satisfy a dynamic condition, describing the actual effects of declassification operations at runtime. This distinction is fairly subtle. For instance, Beame and Machmouchi [1] give a lower bound for oblivious branching programs that was originally believed to apply to Oblivious RAM; however, as indicated by the authors in a supplementary note [2], their result applies only to data-independent algorithms, not to data-oblivious algorithms such as Oblivious RAM.

We also remark that, in general, deterministic data-independence is a very strong condition. For hardware implementations, it may be the case that the control flow of the algorithm is fixed a priori and cannot depend on any random bits at runtime. In virtually all applications in cryptography and security, however, no such restriction exists in practice. Even in the case of fully homomorphic encryption, the server operating on ciphertexts may generate and encrypt its own random bits.[4] Thus, for the remainder of this paper, we will use the term *data-independent* to refer to probabilistically data-independent algorithms, unless otherwise indicated.

Finally, we will sometimes want to enable the machine to execute cryptographic operations, such as evaluating a pseudorandom function (PRF), at unit cost. While this feature is standard in the Oblivious RAM setting [7, 10], it introduces subtle difficulties. The largest cryptographic key that will fit in the machine's $O(1)$ registers is of size $\Theta(\log n)$—which cannot possibly provide security better than $2^{\Theta(\log n)} = \text{poly}(n)$. So instead, whenever cryptographic operations are relevant, we introduce an additional security parameter λ, and assume that the adversary's advantage is negligible in any relevant cryptographic game when instantiated with λ as the security parameter.[5] In such settings, when we write $O(t(n))$, it should be taken to mean $O(t(n) \cdot (1 + \text{poly}(\lambda / \log n)))$, i.e., $O(t(n))$ standard RAM operations plus $O(t(n))$ cryptographic operations.

---

[3] We extend this definition in the natural way to *statistically data-oblivious*, in which the distributions for two different secret inputs are statistically close; or *computationally data-oblivious*, in which the distributions are computationally indistinguishable. In this case, we refer to the original definition as *perfectly data-oblivious*.

[4] Indeed, even when the server must provide proof that it conducted the computation correctly, it can still simulate the same effect by evaluating a pseudorandom function homomorphically.

[5] In practical settings, this may require a superpolynomial hardness assumption, since we generally always have $\lambda < n$. An alternate approach, suggested by Goldreich and Ostrovsky [7], is to give the machine (but not the adversary) unit-cost access to a random oracle.

2.1 General program transformations

We note that there is a general upper bound on the cost of making an arbitrary algorithm data-independent:

▶ **Theorem 1.** *Any RAM program whose running time is at most $T(n)$, and refers to memory addresses bounded by $S(n)$,* [6] *may be translated to a deterministically data-independent RAM program that uses $\Theta(S(n))$ space and runs in time $\Theta(T(n)S(n))$.*

Proof. The translation is simply by brute force. At each time step, iterate over the entire finite program, and for each possible pair ⟨instruction, memory location⟩, compute the Boolean value b corresponding to whether (a) the program counter resides at the given instruction and (b) the given memory location is requested by that instruction. Then compute the result of executing the instruction, and set the resulting register and/or memory location accordingly, by "arithmetizing" the branch: $A_\ell := (b \cdot \text{new\_value}) + ((1 - \text{b}) \cdot \text{A}_\ell)$. ◀

If the algorithm need only be data-oblivious, not necessarily data-independent, then the upper bound is considerably better. A long line of work on the Oblivious RAM problem [7, 15, 10, 17, 6] has produced increasingly efficient data-oblivious simulations of RAMs, achieving the following complexity bound [10]:

▶ **Theorem 2** (Kushilevitz, Lu, and Ostrovsky, 2012)**.** *Assuming one-way functions exist, there is an Oblivious RAM that requires $O(\log^2 n / \log \log n)$ amortized overhead per operation and uses $\Theta(n)$ space.*

2.2 Data-oblivious data structures

We extend the formal treatment above to data structures, by considering each operation on a data structure as an algorithm taking its current internal state and a query as input, and producing as output its result and subsequent internal state.[7]

As a simple example, we consider the case of the array: we must produce an algorithm that takes a query tuple (read/write, index, value), along with some current memory state, and returns a new memory state (and, if the operation was a read, the result of the query). For simplicity, we restrict our attention to arrays that are *operation-secret* as well as *data-secret*—i.e., the identity of the operation being performed (read or write), is labeled as secret, in addition to the values in the array.[8] (Of course, we also restrict our attention to arrays that are *index-secret*—i.e., the index i is labeled as secret—since otherwise the solution is trivial; an ordinary random-access array suffices.)

For general arrays, it is clear that we cannot hope for a nontrivial data-independent construction:

▶ Proposition 3. Any data-independent array of size n requires $\Omega(n)$ space and $\Omega(n)$ time per operation.

Proof. Immediate by an information-theoretic argument. ◀

[6] Technically, we also require T and S to be time- and space-constructible by an oblivious machine, since we must decide when to stop the simulation without inspecting the simulated data values.

[7] Here we remark that our notion of data-oblivious data structures is also distinct from that of Micciancio [12]: under our definition, the obliviousness criterion applies not to the physical representation of the data structure, but rather to the algorithms that implement its operations. Data-obliviousness is also distinct from cache-obliviousness (as in external memory models).

[8] One could also weaken the requirements so that arrays may be *operation-public*, and describe the complexity of reads and writes separately, but this does not change the situation up to constant factors, since we can just execute both a read and a write whenever either is requested.

We may consider relaxing the requirements by specifying only that the array be data-oblivious, rather than data-independent. In this case, the setting coincides with that of Oblivious RAM, and Theorem 2 gives an efficient construction.

Alternatively, we can relax the requirements not by changing the model, but by restricting the features of the data structure itself. Indeed, we now show that for many common uses of arrays—notably, stacks, queues, and priority queues—it is possible to do considerably better than the naive translation, even in the data-independent setting.

3 Stacks and queues, via composition

In some of the earliest work that considers obliviousness as an explicit goal, Pippenger and Fischer give a simulation of a multitape Turing machine by a deterministic data-independent two-tape Turing machine, running in time $O(T(n) \log T(n))$, where $T(n)$ bounds the running time of the original machine [16]. Initially, constructions such as the Pippenger-Fischer simulation were used to refine the time hierarchy theorem, as they permit a more efficient universal Turing machine simulation than the naive $\Theta(T(n)^2)$.

However, from the perspective of oblivious algorithms (even executed on a RAM), we find that the Turing machine tape is a useful data structure in its own right. As above, we will restrict our attention to the nontrivial case, that of *operation-secret* Turing machine tapes: the identity of each operation (i.e., $\{\text{read}, \text{write}\} \times \{\text{left}, \text{right}\}$), as well as each tape symbol, is deemed secret for the purpose of obliviousness. We state the following results:

▶ **Theorem 4.** *There exists a deterministic data-independent Turing machine tape with (pre-specified, public) length n, using $\Theta(n)$ space and $O(\log n)$ amortized time per operation. The tape may be taken to be either (a) toroidal, so that attempting to move off the right end wraps around to the left end, and vice versa; or (b) bounded, so that attempting to move off either end results in no head movement.*

Proof. Follows directly from the Pippenger-Fischer simulation [16]. ◀

▶ **Corollary 5.** *There exists a deterministic data-independent stack with (pre-specified, public) capacity n, using $\Theta(n)$ space and $\Theta(\log n)$ amortized time per operation.*

Proof. Follows from Theorem 4, since a stack may be implemented by writing the elements in sequence on a Turing machine tape, leaving the head at the end. ◀

Fischer et al. also showed that it is possible to simulate a Turing machine with multiple heads using a Turing machine with multiple tapes (but only one head on each tape), with only a constant factor slowdown [5]. This result enables a straightforward construction of efficient oblivious queues, by composition with the stack of the previous section:

▶ **Theorem 6.** *There exists a deterministic data-independent queue with (pre-specified, public) capacity n, using $\Theta(n)$ space and $\Theta(\log n)$ amortized time per operation.*

Proof. Given a two-headed Turing machine tape, a queue can trivially be implemented with overhead $\Theta(1)$ (and no extra space), by keeping the front of the queue at one head and the back of the queue at the other. A two-headed Turing machine tape can be implemented by a constant number of (non-oblivious) single-headed Turing machine tapes, also with linear space and only a constant factor slowdown [5, 11]. Thus, if we use the oblivious Turing machine tape of Pippenger and Fischer (Theorem 4) to implement each, the entire construction uses only $\Theta(\log n)$ amortized time per operation (and linear space). ◀

4 Stacks and queues, directly

In the previous section, we illustrated the ability to build data-oblivious data structures by composition of other data structures. We now show that one may also proceed from first principles, sometimes obtaining constructions that are much simpler and have better constant factors. Generally, efficient data-oblivious data structures share the following traits:

- Data locality – the Turing machine tape, for example, can only visit a neighborhood of size $O(k)$ within k steps.
- Self-similarity or isotropy – the "local context" at any point in the data structure must appear the same, so that the structure can operate obliviously on local data, and can obliviously shift data so that the new local context is correct.

Proceeding from these principles, we arrive at the following results:

▶ **Theorem 7.** *There exists a deterministic data-independent stack with (pre-specified, public) capacity n, requiring linear space and amortized time at most $\sim 8\lceil \lg n \rceil$ per operation.*

▶ **Theorem 8.** *There exists a deterministic data-independent queue with (pre-specified, public) capacity n, requiring linear space and amortized time at most $\sim 11.5\lceil \lg n \rceil$ per operation.*

At a high level, our constructions implement a "b-structure", which operates on b-word blocks, in terms of (i) a buffer of a constant number of b-word blocks, kept fairly close to half-full; and (b) a "$2b$-structure", defined inductively, into which blocks are pushed or pulled as the local buffer becomes too imbalanced. We defer the details to the extended version of this work. We note that the overhead of our oblivious stack is significantly better than the $24\lceil \lg n \rceil$ of the Pippenger-Fischer simulation (Corollary 5), and even yields a tighter oblivious Turing machine tape ($16\lceil \lg n \rceil$, by implementing the tape using two stacks). The constant-factor improvement in the oblivious queue is even more significant, since Theorem 6 invokes not only the Pippenger-Fischer simulation but also a simulation of a two-headed machine.

5 Priority queues

As another example of data structure composition, we now show how to build efficient oblivious priority queues (both operation-secret and operation-public), using oblivious queues as a key component. As above, we will be concerned only with data-secret priority queues, since the data-public case is trivial. However, unlike the Turing machine tape, the oblivious priority queue harbors a nontrivial distinction between operation-secret and operation-public structures, and we will consider both. In what follows, we write a to denote the capacity of the priority queue, n the number of items in the priority queue at any given time, and m the total number of operations that have been performed.

First, we note that if the priority queue is operation-secret, then the time bounds must depend only on m and a (since n depends on whether the operations performed were insertions or deletions.) We also note that in general:

▶ **Lemma 9.** *Given an operation-public priority queue such that remove-min and insert both run in amortized time $f(a, n, m)$ (where f is monotonically increasing and poly(a, n, m)), there exists an operation-secret priority queue such that both operations run in amortized time $O(f(m, m, m))$.*

Proof. By performing dummy operations; we defer the details to the extended version. ◀

▶ **Theorem 10.** *There exists a deterministic data-independent, operation-secret priority queue, using* $\Theta(\min(a, m))$ *space and* $\Theta(\log^2(\min(a, m)))$ *amortized time per operation.*

Proof. For simplicity, we assume the capacity of the priority queue is a power of two, $a = 2^k$, and that all elements are distinct. We use a hierarchical series of k oblivious queues (Section 8), as in standard Oblivious RAM constructions; the i^{th} queue has capacity 2^{i+1}.

We maintain the following invariant: at the beginning of any operation, queue i contains up to 2^i items in sorted order. The operations are then implemented as follows:[9]

1. To remove the minimum: first, find the minimum by examining the front of each queue.[10] Then, for each queue, if its front is the minimum, pop it; otherwise, do nothing. (Since all elements are distinct, only one of the queues will actually be popped.)
2. To insert an item: first, let l be the deepest level such that 2^l divides m (where m, as above, is the number of operations performed so far). Create a temporary buffer queue B of size 2^{l+1}, holding only the new element. Then, for each level $i \in (0, \ldots, l)$, merge the contents of queue i into B. This can be done using the standard merge algorithm: pop whichever of the two queues currently yields the smaller element, accumulating the results in a new buffer B', and finally replace B with B'.

By the properties of the oblivious queue, operation (1) takes amortized time $\Theta(\log(2^i))$ on level i, and thus the total running time is $\sum_{i=0}^{l-1} \Theta(i) = \Theta(l^2) = \Theta(\log^2 \min(a, m))$ (where l is the index of the largest occupied level). On the other hand, operation (2) merges a queue of size $\Theta(2^i)$ into its successor (taking time $\Theta(2^i \log 2^i)$) once after every 2^i operations, and thus the amortized cost of each operation is $\sum_{i=0}^{l}(1/2^i)(\Theta(2^i \log 2^i)) = \Theta(l^2) = \Theta(\log^2 \min(a, m))$ also. ◀

▶ **Theorem 11.** *There exists a deterministic data-independent operation-public priority queue, using* $\Theta(n)$ *space and* $\Theta(\log^2 n)$ *amortized time per operation, where* n *is the size of the priority queue prior to each operation.*

Proof. Similar to the proof of Theorem 10; we defer the details to the extended version. ◀

Theorems 10 and 11 establish efficient constructions of oblivious priority queues in both the operation-secret and the operation-public models. Evidently, for series of operations in which $m = O(n)$ (e.g., inserting $2m/3$ items followed by removing $m/3$ items), the performance in the operation-secret case is no worse asymptotically than in the operation-public case; since the operation-secret queue cannot reveal the pattern of operations, in a sense, this is the best we can hope for, without also obtaining a faster operation-public queue. Thus, the distinction between operation-public and operation-secret priority queues turns out not to matter, at least in the context of our best known constructions.

The constructions above compare favorably with the generic solution in the data-oblivious setting: i.e., representing the priority queues as min-heaps, and using Oblivious RAM to serve as the underlying array. In this case, using the Oblivious RAM of Kushilevitz et al. [10], we would spend $\Theta(\log^3 n / \log\log n)$ time, and $\Theta(\log^2 n)$ sequential communication rounds, per operation (and we would incur either a large constant factor, if the Oblivious RAM uses the AKS sorting network; or a moderate constant factor, with some small probability of error, if the Oblivious RAM uses Goodrich's randomized Shellsort). In contrast, our constructions

[9] In fact, since the priority queue is operation-secret, of course, we simulate both (1) and (2) when either operation is requested.

[10] Strictly speaking, this is not an operation of the queue interface above, but it is a straightforward extension since the pop operation is not destructive.

are not just data-oblivious but data-independent; they operate deterministically, and costs only $\Theta(\log^2 n)$ time per operation, with a very small constant factor.

We note that in recent independent work, Toft [18] has also described a data-oblivious priority queue with the same asymptotic bounds, albeit by a different method. The constructions of this section are much simpler, however, due to the composition of data-oblivious primitives.

6 Data compaction and the partition problem

Goodrich [8] describes a problem called *data compaction*: given an array of length n in which r of the elements are marked as "distinguished", construct a new array of size $O(r)$ containing only the distinguished items. If the resulting array is required to be of size exactly r, the compaction is said to be *tight*; otherwise it is *loose*. If the items are also required to be in their original order, the compaction is additionally said to be *order-preserving*. Goodrich proves the following:

▶ **Theorem 12** (Goodrich, 2011). *There exists a deterministically data-independent algorithm for tight order-preserving data compaction running in time $\Theta(n \log n)$.*

▶ **Theorem 13** (Goodrich, 2011). *There is a data-oblivious algorithm that runs in time $O(n \log^* n)$ and achieves loose data compaction with high probability when $r < n/4$.*

However, for the next section, we will need fast tight compaction. To that end, we first rephrase the problem of tight compaction in terms of the *partition problem*: given an array of n items each tagged with a single bit, separate the items so that all those tagged with a 0 appear to the left of those tagged with a 1.[11]

Now, we will show:

▶ **Theorem 14.** *There is a data-independent algorithm that runs in time $\Theta(n \log \log n)$ and achieves tight data compaction (i.e., solves the partition problem) with high probability.*

Proof. First, we note that we can easily count which tag (0 or 1) constitutes a majority, and obliviously decide whether to invert the tags (and reverse the array) based on that information. Thus, it suffices to give an algorithm to extract a constant fraction of whichever tag constitutes a majority (placing the extracted items at either the beginning or the end of the array, as appropriate), since we may then recur on the rest of the array. At any stage in this recursion, let n denote the size of the entire array, and n' denote the size of the subarray being partitioned at this point.

Now, without loss of generality, suppose the subarray contains more zeroes than ones. Let A denote the leftmost $n'/3$ cells, and B the rightmost $2n'/3$. Let s, the *bucket size*, be $(\log n)^k$ for an arbitrary constant $k > 1$. Then:

1. For each cell a_i in A, pick c cells b_i uniformly at random from B (for some constant c to be determined), and, if b_i contains a 0, swap a_i with b_i.
2. Divide A into $n'/3s$ buckets of s cells each. Partition each bucket using any $\Theta(s \log s)$ algorithm (e.g., Theorem 12).

[11] To show the equivalence, we note that in order to solve the data compaction problem, we can simply mark the distinguished cells with 0, run a partition algorithm, and return the appropriate prefix of the original array. Conversely, to solve the partition problem, we run data compaction twice: once with the 0 elements marked as distinguished, and once with the array in reverse order and the 1 elements marked as distinguished. Then, iterate over the two resulting arrays in parallel, and at each index, copy from whichever array has a distinguished item.

3. Extract and concatenate all of the first halves of the buckets (i.e., $n'/3s$ half-buckets, each of size $s/2$, for a total of $n'/6$ items).

Evidently, this procedure performs a total of $cn'/3+(n'/3s)(s\log s)+n'/6 = \Theta(n'\log s) = \Theta(n'\log\log n)$ operations, giving a running time of $\Theta(n'\log\log n)$ for a subarray of size n'. Since $T(n') = T(5n'/6) + \Theta(n'\log\log n)$ (and noting that we may stop the recursion at $n' = O(s)$, solving the base case directly by the algorithm of Theorem 12 with $T(s) = O(s\log s) = s\log\log n$), we have an overall running time of $T(n) = \Theta(n\log\log n)$.

For correctness, we now claim that for any given subarray of size n', after the final step of the iteration, the first $n'/6$ cells of A are all 0 with high probability. To show this, we first note that since zeroes constitute a majority of the array, B (the rightmost $2n'/3$ cells) must contain at least $n'/6$ zeroes at all times. Thus, in step 1, a_i will become zero with probability at least $(n'/6)/(2n'/3) = 1/4$ on every potential swap, independently of any other events. Since c swaps are performed, the probability that a_i will become zero after the entire step is at least $1-(3/4)^c$; we choose c so that this success probability is bounded away from $1/2$. Thus, by Hoeffding's inequality (and a union bound), the probability that there are more than $s/2$ ones in any bucket (and, thus, the probability that we fail to extract $n'/6$ zeroes after partitioning and splitting) is at most $(n'/3s)e^{-2(2/3-1/2)^2 s} = (n'/6)e^{-\Theta(s)}$. Even after executing this procedure at every level of the recursion (as n' decreases from n down to s), again by a union bound, the overall probability of error is at most $\sum_{n'}(n'/6)e^{-\Theta(s)} = ne^{-\Theta(s)} = n^{-\Theta(\lg^{k-1} n)}$. ◀

Crucially, because this partitioning algorithm is based on swaps (and is data-independent, not just data-oblivious), we can also use it to "un-partition", or intersperse, items:

▶ **Theorem 15.** *Given an array of r items and an array of r binary tags, there is a data-independent algorithm that runs in time $\Theta(r\log\log n)$ and, with probability $1-1/n^{\omega(1)}$, produces a permutation of the array such that every element that was originally in the left half now occupies a location that was tagged with 0. If one-way functions exist, this algorithm can use $O(n)$ words of memory; otherwise, it uses $O(n\log\log n)$ words.*

Proof. Run the algorithm of Theorem 14 on the tag array, and record a "trace" consisting of each pair of indices that it decided to swap, accompanied by a (secret) bit indicating whether those items were actually swapped. (These indices may also include empty scratch cells, not initially containing either a 0 or a 1 item.) Then, run the trace in reverse on the array of actual items.

The space used by the trace is $O(r\log\log n)$ bits, which requires $o(r)$ words; plus $O(r\log\log n)$ pairs of indices, which requires $O(r\log\log n)$ words. However, assuming one-way functions (and hence PRFs), the index pairs can be taken to be the output of a PRF on a short seed and a time step, in which case the entire assembly requires only linear space. ◀

▶ **Lemma 16.** *Given two arrays of size r, each permuted according to a distribution that is computationally indistinguishable from uniform, there is a data-independent algorithm that runs in time $\Theta(r\log\log n)$ and results in a permutation that is computationally indistinguishable from uniform on all $2r$ elements.*

Proof. Follows from Theorem 15. ◀

7 Offline Oblivious RAM

We now show that the data compaction problem is closely related to the question of data-oblivious arrays (i.e., Oblivious RAM). Intuitively, the hierarchical Oblivious RAM

constructions of Goldreich and Ostrovsky [7] and of Kushilevitz et al. [10] must remember (and hide from the adversary) two things: (i) the "level", or epoch, in which each item resides; and (ii) the permutation of the items within each epoch. As a result, they must use techniques such as oblivious cuckoo hashing via sorting [10] or logarithmic-sized buckets [7] in order to make accesses to a level look the same whether an item is present or not. However, we can separate these two issues by considering a variant of the problem in which the RAM need not remember when an item was last accessed—perhaps because the sequence of memory addresses is known in advance, and can be preprocessed before the queries are made.

More precisely, we define a *primed array* to be a data structure that has the same interface as an array, except that each request (read or write) is accompanied by the (logarithm of the) time since the requested cell was last accessed. We note that a primed array generalizes an "offline" array with preprocessed queries, since the preprocessor may annotate each query with the preceding access time to that cell. Now, we can show:

▶ **Theorem 17.** *Assuming one-way functions exist, there is a (computationally) data-oblivious primed array with expected amortized overhead $O((p(n)/n)\log n)$ per operation, where $p(n)$ is the time required for oblivious partitioning (or data compaction) of n items.*

Proof. As in other hierarchical Oblivious RAM constructions, we maintain a series of levels of increasing power-of-two sizes, in which level i contains between zero and $r = 2^i$ items, and is "shuffled" into the next larger level (in a sense specified below) and after every epoch of length r. Here, level i consists of a dictionary D_i of size $\Theta(r)$, mapping each of $2r$ keys to a data item, where each key is either an element of $\{1, \ldots, n\}$ or one of the dummy values $\{\text{dummy}_1^{t_i}, \ldots, \text{dummy}_r^{t_i}\}$, where t_i is the time of the most recent shuffle at level i (superscripts elided below for clarity). (D_i may be implemented by a cuckoo hash table, or even a standard linked-list-based hash table, since we will not depend on its hiding the access pattern—only on the fact that it implements a dictionary in expected amortized constant time.) Further, during any fixed epoch, level i will have an associated injective PRF $\psi_i^{t_i}$ (again, we drop the time index for clarity, and just write ψ_i). For each logical address x present at level i, $D_i(\psi_i(x))$ is the item stored at x; while for each dummy index dummy_s, the key $\psi_i(\text{dummy}_s)$ is present in D_i (and is mapped to a dummy item). Now, upon receiving a query, we find out how long ago the cell was accessed. This tells us in which level it resides (since, as in standard Oblivious RAM constructions, we will "promote" an item to the top level whenever it is accessed). Suppose it resides at level i. We then access a real item in that level only, by querying D_i at $\psi_i(x)$, and query for a unique dummy item in all other levels (e.g., $\psi_j(\text{dummy}_{t \bmod 2^j})$).

When it is time for a level to be merged into the next level down, we first extract (via oblivious partitioning) the items from both levels that have not yet been accessed during this epoch—possibly including some real items and some dummy items (which we overwrite with new dummy indices appropriate to the new, merged level). Then, we randomly intersperse these two lists of items (Section 6). Finally, we fill the resulting $2r$ items into the new level's hash table, keyed by the values of a new PRF (evaluated on either the items' addresses, for real items, or on their new dummy indices).

During this process, the adversary sees the values of the new PRF on all addresses remaining in both levels, appearing in the order they were in after the partition operations. Since these addresses are all distinct, their images under the PRF are indistinguishable from an independently uniform set of size $2r$. Hence the information obtained by the adversary is completely described by the correspondence between these values and the PRF image value queried on each level's dictionary when an item is retrieved (recall that these can match only once, since as soon as an item is found, it is promoted to the top level; while dummy

indices are used only once per epoch). This reveals only the order that an item had when it was initially inserted into its current level during a merge. Thus, if we assume inductively that the ordering of unvisited items in each of the higher levels was indistinguishable from a uniform permutation (of their order of insertion), then it follows from Lemma 16 that the resulting ordering in the current level is also indistinguishable from a uniform permutation, and so is independent of the address request sequence, as desired. ◀

▶ **Corollary 18.** *Assuming one-way functions exist, there is an offline Oblivious RAM with expected amortized overhead* $O(\log n \log\log n)$ *per operation.*

Proof. Immediate from Theorems 14 and 17. ◀

8 Lower bounds via communication complexity

For most problems, it seems very difficult to establish lower bounds on oblivious algorithms, since any nontrivial slowdown due to obliviousness would imply nontrivial uniform circuit lower bounds. For oblivious data structures, however, this is not the case. In fact, by viewing the Oblivious RAM as an oblivious data structure, we immediately obtain a communication complexity lower bound that generalizes both that of Pippenger and Fischer [16], for oblivious Turing machine tapes, and that of Goldreich and Ostrovsky [7], for Oblivious RAMs (we defer the proof to the extended version of this work):

▶ **Theorem 19.** *Any data-oblivious Turing machine tape of length* n *requires* $\Omega(\log n)$ *expected amortized time per operation.*

Now Theorem 19 immediately entails Ostrovsky's lower bound for Oblivious RAM (originally proven by a combinatorial argument [7]):

▶ **Corollary 20.** *Any Oblivious RAM (i.e., any data-oblivious array) with* n *memory words requires* $\Omega(\log n)$ *expected amortized time per operation.*

Proof. Use the array to implement a Turing machine tape and invoke Theorem 19. ◀

Acknowledgements. We thank Dan Boneh and Valeria Nikolaenko for helpful discussions, and the anonymous reviewers for their comments. This work was supported by DARPA PROCEED, under contract #N00014-11-1-0276-P00002, the National Science Foundation, and the Air Force Office of Scientific Research. During this work, Joe Zimmerman has been further supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program (2012-2013), and by the National Science Foundation (NSF) through the National Science Foundation Graduate Research Fellowship Program (GRFP) (2013-2014).

References

1 Paul Beame and Widad Machmouchi. Making branching programs oblivious requires superlogarithmic overhead. In *IEEE Conference on Computational Complexity*, pages 12–22. IEEE Computer Society, 2011.

2 Paul Beame and Widad Machmouchi. Making RAMs oblivious requires superlogarithmic overhead. *ECCC*, 2011.

3 Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure Oblivious RAM without random oracles. In Yuval Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2011.

4 David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 13–22, New York, NY, USA, 2010. ACM.

5 Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Real-time simulation of multihead tape units. *J. ACM*, 19(4):590–607, October 1972.

6 Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

7 Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

8 Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 379–388, New York, NY, USA, 2011. ACM.

9 F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape turing machines. *J. ACM*, 13(4):533–546, October 1966.

10 Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *SODA*, pages 143–156. SIAM, 2012.

11 Benton L. Leong and Joel I. Seiferas. New real-time simulations of multihead tape units. *J. ACM*, 28(1):166–180, January 1981.

12 Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 456–464, New York, NY, USA, 1997. ACM.

13 John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. Information-flow control for programming on encrypted data. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 45–60. IEEE, June 2012.

14 Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000.

15 Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 502–519, Berlin, Heidelberg, 2010. Springer-Verlag.

16 Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, April 1979.

17 Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious RAM protocol. *IACR Cryptology ePrint Archive*, 2013:280, 2013.

18 Tomas Toft. Secure data structures based on multi-party computation. In Cyril Gavoille and Pierre Fraigniaud, editors, *PODC*, pages 291–292. ACM, 2011.