# Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation

Michael T. Goodrich[1] and Michael Mitzenmacher[2]

[1] University of California, Irvine
[2] Harvard University

**Abstract.** We describe schemes for the *oblivious RAM simulation* problem with a small logarithmic or polylogarithmic amortized increase in access times, with a very high probability of success, while keeping the external storage to be of size $O(n)$.

## 1 Introduction

Suppose Alice owns a large data set, which she outsources to an honest-but-curious server, Bob. For the sake of privacy, Alice can, of course, encrypt the cells of the data she stores with Bob. But encryption is not enough, as Alice can inadvertently reveal information about her data based on how she accesses it. Thus, we desire that Alice's access sequence (of memory reads and writes) is *data-oblivious*, that is, the probability distribution for Alice's access sequence should depend only on the size, $n$, of the data set and the number of memory accesses. Formally, we say a computation is data-oblivious if $\mathbf{Pr}(S \,|\, \mathcal{M})$, the probability that Bob sees an access sequence, $S$, conditioned on a specific configuration of his memory (Alice's outsourced memory), $\mathcal{M}$, satisfies $\mathbf{Pr}(S \,|\, \mathcal{M}) = \mathbf{Pr}(S \,|\, \mathcal{M}')$, for any memory configuration $\mathcal{M}' \neq \mathcal{M}$ such that $|\mathcal{M}'| = |\mathcal{M}|$. In particular, Alice's access sequence should not depend on the values of any set of memory cells in the outsourced memory that Bob maintains for Alice. To provide for full application generality, we assume outsourced data is indexed and we allow Alice to make arbitrary indexed accesses to this data for queries and updates. That is, let us assume this outsourced data model is as general as the random-access machine (RAM) model.

Most computations that Alice would be likely to perform on her outsourced data are not naturally data-oblivious. We are therefore interested in this paper in simulation schemes that would allow Alice to make her access sequence data-oblivious with low overhead. For this problem, which is known as *oblivious RAM simulation* [5], we are primarily interested in the case where Alice has a relatively small private memory, say, of constant size or size that is $O(n^{1/r})$, for a given constant $r > 1$.

*Our Results.* In this paper, we show how Alice can perform an oblivious RAM simulation, with very high probability[1], with an amortized time overhead of $O(\log n)$ and

[1] We show that our simulation fails to be oblivious with negligible probability; that is, the probability that the algorithm fails can be shown to be $O\left(\frac{1}{n^{\alpha}}\right)$ for any $\alpha > 0$. We say an event holds with *very high probability* if it fails with negligible probability.

with $O(n)$ storage overhead purchased from Bob, while using a private memory of size $O(n^{1/r})$, for a given constant $r > 1$. With a constant-sized memory, we show that she can do this simulation with overhead $O(\log^2 n)$, with a similarly high probability of success. At a high level, our result shows that Alice can leverage the privacy of her small memory to achieve privacy in her much larger outsourced data set of size $n$. Interestingly, our techniques involve the interplay of some seemingly unrelated new results, which may be of independent interest, including an efficient MapReduce parallel algorithm for cuckoo hashing and a novel deterministic data-oblivious external-memory sorting algorithm.

*Previous Related Results.* Goldreich and Ostrovsky [5] introduce the oblivious RAM simulation problem and show that it requires an overhead of $\Omega(\log n)$ under some reasonable assumptions about the nature of such simulations. For the case where Alice has only a constant-size private memory, they show how Alice can easily achieve an overhead of $O(n^{1/2} \log n)$, with $O(n)$ storage at Bob's server, and, with a more complicated scheme, how Alice can achieve an overhead of $O(\log^3 n)$ with $O(n \log n)$ storage at Bob's server.

Williams and Sion [15] study the oblivious RAM simulation problem for the case when the data owner, Alice, has a private memory of size $O(n^{1/2})$, achieving an expected amortized time overhead of $O(\log^2 n)$ using $O(n \log n)$ memory at the data provider. Incidentally, Williams *et al.* [16] claim an alternative method that uses an $O(n^{1/2})$-sized private memory and achieves $O(\log n \log \log n)$ amortized time overhead with a linear-sized outsourced storage, but some researchers (e.g., see [14]) have raised concerns with the assumptions and analysis of this result.

The results of this paper were posted by the authors in preliminary form as [7]. Independently, Pinkas and Reinman [14] published an oblivious RAM simulation result for the case where Alice maintains a constant-size private memory, claiming that Alice can achieve an expected amortized overhead of $O(\log^2 n)$ while using $O(n)$ storage space at the data outsourcer, Bob. Unfortunately, their construction contains a flaw that allows Bob to learn Alice's access sequence, with high probability, in some cases, which our construction avoids.

Ajtai [1] shows how oblivious RAM simulation can be done with a polylogarithmic factor overhead without cryptographic assumptions about the existence of random hash functions, as is done in previous papers [5,14,15,16], as well as any paper that derives its security or privacy from the random oracle model (including this paper). A similar result is also given by Damgård *et al.* [2]. Although these results address interesting theoretical limits of what is achievable without random hash functions, we feel that the assumption about the existence of random hash functions is actually not a major obstacle in practice, given the ubiquitous use of cryptographic hash functions.

## 2 Preliminaries

*A Review of Cuckoo Hashing.* Pagh and Rodler [13] introduce *cuckoo hashing*, which is a hashing scheme using two tables, each with $m$ cells, and two hash functions, $h_1$ and $h_2$, one for each table, where we assume $h_1$ and $h_2$ can be modeled as random hash functions for the sake of analysis. The tables store $n = (1 - \epsilon)m$ keys, where one key can be held in each cell, for a constant $\epsilon < 1$. Keys can be inserted or deleted over time; the requirement is that at most $n = (1-\epsilon)m$ distinct keys are stored at any time. A stored key $x$ should be located at either $h_1(x)$ or $h_2(x)$, and, hence, lookups take constant time. On insertion of a new key $x$, cell $h_1(x)$ is examined. If this cell is empty, $x$ is placed in this cell and the operation is complete. Otherwise, $x$ is placed in this cell and the key $y$ already in the cell is moved to $h_2(y)$. This may in turn cause another key to be moved, and so on. We say that a failure occurs if, for an appropriate constant $c_0$, after $c_0 \log n$ steps this process has not successfully terminated with all keys located in an appropriate cell. Suppose we insert an $n$th key into the system. Well-known attributes of cuckoo hashing include:

- The expected time to insert a new key is bounded above by a constant.
- The probability a new key causes a failure is $\Theta(1/n^2)$.

Kirsch, Mitzenmacher, and Wieder introduce the idea of utilizing a *stash* [10]. A stash can be thought of as an additional memory where keys that would otherwise cause a failure can be placed. In such a setting, a failure occurs only if the stash itself overflows. For $n$ items inserted in a two-table cuckoo hash table, the total failure probability can be reduced to $O(1/n^{k+1})$ for any constant $k$ using a stash that can hold $k$ keys. For our results, we require a generalization of this result to stashes that are larger than constant sized.

## 3 MapReduce Cuckoo Hashing

*The MapReduce Paradigm.* In the MapReduce paradigm (e.g., see [4,9]), a parallel computation is defined on a set of values, $\{x_1, x_2, \ldots, x_n\}$, and consists of a series of *map*, *shuffle*, and *reduce* steps:

- A map step applies a *mapping function*, $\mu$, to each value, $x_i$, to produce a key-value pair, $(k_i, v_i)$. To allow for parallel execution, the function, $\mu(x_i) \rightarrow (k_i, v_i)$, must depend only on $x_i$.
- A shuffle step takes all the key-value pairs produced in the previous map step, and produces a set of lists, $L_k = (k; v_{i_1}, v_{i_2}, \ldots)$, where each such list consists of all the values, $v_{i_j}$, such that $k_{i_j} = k$ for a key $k$ assigned in the map step.
- A reduce step applies a *reduction function*, $\rho$, to each list, $L_k = (k; v_{i_1}, v_{i_2}, \ldots)$, formed in the shuffle step, to produce a set of values, $w_1, w_2, \ldots$. The reduction function, $\rho$, is allowed to be defined sequentially on $L_k$, but should be independent of other lists $L_{k'}$ where $k' \neq k$.

Since we are using a MapReduce algorithm as a means to an end, rather than as an end in itself, we allow values produced at the end of a reduce step to be of two types: *final values*, which should be included in the output of such an algorithm when it completes and are not included in the set of values given as input to the next map step, and *non-final values*, which are to be used as the input to the next map step. Thus, for our purposes, a MapReduce computation continues performing map, shuffle, and reduce steps until the last reduce step is executed, at which point we output all the final values produced over the course of the algorithm.

In the MUD version of this model [4], which we call the *streaming-MapReduce* model, the computation of $\rho$ is restricted to be a streaming algorithm that uses only $O(\log^c n)$ working storage, for a constant $c \geq 0$. Given our interest in applications to data-oblivious computations, we define a version that further restricts the computation of $\rho$ to be a streaming algorithm that uses only $O(1)$ working storage. That is, we focus on a streaming-MapReduce model where $c = 0$, which we call the *sparse-streaming-MapReduce* model. In applying this paradigm to solve some problem, we assume we are initially given a set of $n$ values as input, for which we then perform $t$ steps of map-shuffle-reduce, as specified by a sparse-streaming-MapReduce algorithm, $\mathcal{A}$.

Let us define the *message complexity* of a MapReduce to be the total size of all the inputs and outputs to all the map, shuffle, and reduce steps in a MapReduce algorithm. That is, if we let $n_i$ denote the total size of the input and output sets for the $i$th phase of map, shuffle, and reduce steps, then the message complexity of a MapReduce algorithm is $\sum_i n_i$.

Suppose we have a function, $f(i, n)$, such that $n_i \leq f(i, n)$, for each phase $i$, over all possible executions of a MapReduce algorithm, $\mathcal{A}$, that begins with an input of size $n$. In this case, let us say that $f$ is a *ceiling function* for $\mathcal{A}$. Such a function is useful for bounding the worst-case performance overhead for a MapReduce computation.

*A MapReduce Algorithm for Cuckoo Hashing.* Let us now describe an efficient algorithm for setting up a cuckoo hashing scheme for a given set, $X = \{x_1, x_2, \ldots, x_n\}$, of items, which we assume come from a universe that can be linearly ordered in some arbitrary fashion. Let $T_1$ and $T_2$ be the two tables that we are to use for cuckoo hashing and let $h_1$ and $h_2$ be two candidate hash functions that we are intending to use as well.

For each $x_i$ in $X$, recall that $h_1(x_i)$ and $h_2(x_i)$ are the two possible locations for $x_i$ in $T_1$ and $T_2$. We can define a bipartite graph, $G$, commonly called the *cuckoo graph*, with vertex sets $U = \{h_1(x_i)\colon\ x_i \in X\}$ and $W = \{h_2(x_i)\colon\ x_i \in X\}$ and edge set $E = \{(h_1(x_i), h_2(x_i))\colon\ x_i \in X\}$. That is, for each edge $(u, v)$ in $E$, there is an associated value $x_i$ such that $(u, v) = (h_1(x_i), h_2(x_i))$, with parallel edges allowed. Imagine for a moment that an oracle identifies for us each connected component in $G$ and labels each node $v$ in $G$ with the smallest item belonging to an edge of $v$'s connected component. Then we could initiate a breadth-first search from the node $u$ in $U$ such that $h_1(x_i) = u$ and $x_i$ is the smallest item in $u$'s connected component, to define a BFS tree $T$ rooted at $u$. For each non-root node $v$ in $T$, we can store the item $x_j$ at $v$, where $x_j$ defines the edge from $v$ to its parent in $T$.

If a connected component $C$ in $G$ is in fact a tree, then this breadth-first scheme will accommodate all the items associated with edges of $C$. Otherwise, if $C$ contains some non-tree edges with respect to its BFS tree, then we pick one such edge, $e$. All other non-tree edges belong to items that are going to have to be stored in the stash. For the one chosen non-tree edge, $e$, we assign $e$'s item to one of $e$'s endvertices, $w$, and we perform a "cuckoo" action along the path, $\pi$, from $w$ up to the root of its BFS tree, moving each item on $\pi$ from its current node to the parent of this node on $\pi$. Therefore, we can completely accommodate all items associated with unicyclic subgraphs or tree subgraphs for their connected components. All other items are stored in the stash. For cuckoo graphs corresponding to hash tables with load less than $1/2$, with high probability there are no components with two or more non-tree edges, and the stash further increases the probability that, when such edges exist, they can be handled.

Unfortunately, we don't have an oracle to initiate the above algorithm. Instead, we essentially perform the above algorithm in parallel, starting from all nodes in $U$, assuming they are the root of a BFS tree. Whenever we discover a node should belong to a different BFS tree, we simply ignore all the work we did previously for this node and continue the computation for the "winning" BFS tree (based on the smallest item in that connected component). Consider an efficient MapReduce algorithm for performing $n$ simultaneous breadth-first searches such that, any time two searches "collide," the search that started from a lower-numbered vertex is the one that succeeds. We can easily convert this into an algorithm for cuckoo hashing by adding steps that process non-tree edges in a BFS search. For the first such edge we encounter, we initiate a reverse cuckoo operation, to allocate items in the induced cycle. For all other non-tree edges, we allocate their associated items to the stash.

Intuitively, the BFS initiated from the minimum-numbered vertex, $v$, in a connected component propagates out in a wave, bounces at the leaves of this BFS tree, returns back to $v$ to confirm it as the root, and then propagates back down the BFS tree to finalize all the members of this BFS tree. Thus, in time proportional to the depth of this BFS tree (which, in turn, is at most the size of this BFS tree), we will finalize all the members of this BFS tree. And once these vertices are finalized, we no longer need to process them any more. Moreover, this same argument applies to the modified BFS that performs the cuckoo actions. Therefore, we process each connected component in the cuckoo graph in a number of iterations that is, in the worst-case, equal to three times the size of each such component (since the waves of the BFS move down-up-down, passing over each vertex three times).

To bound both the time for the parallel BFS algorithm to run and to bound its total work, we require bounds on the component sizes that arise in the cuckoo graph. Such bounds naturally appear in previous analyses of cuckoo hashing. In particular, the following result is proven in [10][Lemma 2.4].

**Lemma 1.** *Let $v$ be any fixed vertex in the cuckoo graph and let $C_v$ be its component. Then there exists a constant $\beta \in (0,1)$ such that for any integer $k \geq 0$,*

$$\mathbf{Pr}(|C_v| \geq k) \leq \beta^k.$$

More detailed results concerning the asymptotics of the distribution of component sizes for cuckoo hash tables can be found in, for example [3], although the above result is sufficient to prove linear message-complexity overhead.

Lemma 1 immediately implies that the MapReduce BFS algorithm (and the extension to cuckoo hashing) takes $O(\log n)$ time to complete with high probability.

**Lemma 2.** *The message complexity of the MapReduce BFS algorithm is $O(n)$ with very high probability.*

*Proof.* The message complexity is bounded by a constant times $\sum_v |C_v|$, which in expectation is

$$\mathbf{E}\left[\sum_v |C_v|\right] = \sum_v \mathbf{E}[|C_v|] \leq 2m \sum_{k\geq 0} \mathbf{Pr}(C_v \geq k) \leq 2m \sum_{k\geq 0} \beta^k = O(m).$$

To prove a very high probability bound, we use a variant of Azuma's inequality specific to this situation. If all component sizes were bounded by say $O(\log^2 n)$, then a change in any single edge in the cuckoo graph could affect $\sum_v |C_v|$ by only $O(\log^4 n)$, and we could directly apply Azuma's inequality to the Doob martingale obtained by exposing the edges of the cuckoo graph one at a time. Unfortunately, all component sizes are $O(\log^2 n)$ only with very high probability. However, standard results yield that one can simply add in the probability of a "bad event" to a suitable tail bound, in this case the bad event being that some component size is larger than $c_1 \log^2 n$ for some suitable constant $c_1$. Specifically, we directly utilize Theorem 3.7 from [12], which allows us to conclude that if the probability of a bad event is a superpolynomially small $\delta$, then

$$\mathbf{Pr}\left(\sum_v |C_v| \geq \sum_v \mathbf{E}[|C_v|] + \lambda\right) \leq e^{-(2\lambda^2)/(nc_2 \log^4 n)} + \delta,$$

where $c_2$ is again a suitably chosen constant. Now choosing $\lambda = n^{2/3}$, for example, suffices. □

## 4 Simulating a MapReduce Algorithm Obliviously

Our simulation is based on a reduction to oblivious sorting.

**Theorem 1.** *Suppose $\mathcal{A}$ is a sparse-streaming-MapReduce algorithm that runs in at most $t$ map-shuffle-reduce steps, and suppose further that we have a ceiling function, $f$, for $\mathcal{A}$. Then we can simulate $\mathcal{A}$ in a data-oblivious fashion in the RAM model in time $O(\sum_{i=1}^{t}$ o-sort$(f(i,n)))$, where o-sort$(n)$ is the time needed to sort $n$ items in a data-oblivious fashion.*

*Proof.* Let us consider how we can simulate the map, shuffle, and reduce steps in phase $i$ of algorithm $\mathcal{A}$ in a data-oblivious way. We assume inductively that we store the input values for phase $i$ in an array, $X_i$. Let us also assume inductively that $X_i$ can store values that were created as final values the step $i - 1$. A single scan through the first $f(i,n)$ values of $X_i$, applying the map function, $\mu$, as we go, produces all the key-value pairs for the map step in phase $i$ (where we output a dummy value for each input value that is final or is itself a dummy value). We can store each computed value, one by one, in an oblivious fashion using an output array $Y$. We then obliviously sort $Y$ by keys to bring together all key-value pairs with the same key as consecutive cells in $Y$ (with dummy values taken to be larger than all real keys). This takes time $O($o-sort$(f(i,n))$. Let us then do a scan of the first $f(i,n)$ cells in $Y$ to simulate the reduce step. As we consider each item $z$ in $Y$, we can keep a constant number of state variables as registers in our

RAM, which collectively maintain the key value, $k$, we are considering, the internal state of registers needed to compute $\rho$ on $z$, and the output values produced by $\rho$ on $z$. This size bound is due to the fact that $\mathcal{A}$ is a sparse-streaming-MapReduce algorithm. Since the total size of this state is constant, the total number of output values that could possibly be produced by $\rho$ on an input $z$ can be determined a priori and bounded by a constant, $d$. So, for each value $z$ in $Y$, we write $d$ values to an output array $Z$, according to the function $\rho$, padding with dummy values if needed. The total size of $Z$ is therefore $O(d\,f(i,n))$, which is $O(f(i,n))$. Still, we cannot simply make $Z$ the input for the next map-shuffle-reduce step at this point, since we need the input array to have at most $f(i,n)$ values. Otherwise, we would have an input array that is a factor of $d$ too large for the next phase of the algorithm $\mathcal{A}$. So we perform a data-oblivious sorting of $Z$, with dummy values taken to be larger than all real values, and then we copy the first $f(i,n)$ values of $Z$ to $X_{i+1}$ to serve as the input array for the next step to continue the inductive argument. The total time needed to perform step $i$ is $O(\text{o-sort}(f(i,n))$. When we have completed processing of step $t$, we concatenate all the $X_i$'s together, flagging all the final values as being the output values for the algorithm $\mathcal{A}$, which can be done in a single data-oblivious scan. Therefore, we can simulate each step of $\mathcal{A}$ in a data-oblivious fashion and produce the output from $\mathcal{A}$, as well, at the end. Since we do two sorts on arrays of size $O(f(i,n))$ in each map-shuffle-reduce step, $i$, of $\mathcal{A}$, this simulation takes time $O(\sum_{i=1}^{t} \text{o-sort}(f(i,n)))$. □

We can show that by combining this result with Lemma 2 we get the following:

**Theorem 2.** *Given a set of $n$ distinct items and corresponding hash values, there is a data-oblivious algorithm for constructing a two-table cuckoo hashing scheme of size $O(n)$ with a stash of size $s$, whenever this stash size is sufficient, in $O(\text{o-sort}(n+s))$ time.*

*External-Memory Data-Oblivious Sorting.* In this section, we give our efficient external-memory oblivious sorting algorithm. Recall that in this model memory is divided between an internal memory of size $M$ and an external memory (like a disk), which initially stores an input of size $N$, and that the external memory is divided into blocks of size $B$, for which we can read or write any block in an atomic action called an I/O. In this context, we say that an external-memory sorting algorithm is *data-oblivious* if the sequence of I/Os that it performs is independent of the values of the data it is processing. So suppose we are given an unsorted array $A$ of $N$ comparable items stored in external memory. If $N \le M$, then we copy $A$ into our internal memory, sort it, and copy it back to disk. Otherwise, we divide $A$ into $k = \lceil (M/B)^{1/3} \rceil$ subarrays of size $N/k$ and recursively sort each subarray. Thus, the remaining task is to merge these subarrays into a single sorted array.

Let us therefore focus on the task of merging $k$ sorted arrays of size $n = N/k$ each. If $nk \le M$, then we copy all the lists into internal memory, merge them, and copy them back to disk. Otherwise, let $A[i,j]$ denote the $j$th element in the $i$th array. We form a set of $m$ new subproblems, where the $p$th subproblem involves merging the $k$ sorted subarrays defined by $A[i,j]$ elements such that $j \bmod m = p$, for $m = \lceil (M/B)^{1/3} \rceil$. We form these subproblems by processing each input subarray and filling in the portions of the output subarrays from the input, sending full blocks to disk when they fill up (which will happen at deterministic moments independent of data values), all of which uses $O(N/B)$ I/Os. Then we recursively solve all the subproblems. Let $D[i, j]$ denote the $j$th element in the output of the $i$th subproblem. That is, we can view $D$ as a two-dimensional array, with each row corresponding to the solution to a recursive merge.

**Lemma 3.** *Each row and column of $D$ is in sorted order and all the elements in column $j$ are less than or equal to every element in column $j + k$.*

*Proof.* The lemma follows from Theorem 1 of Lee and Batcher [11]. □

To complete the $k$-way merge, then, we imagine that we slide an $m \times k$ rectangle across $D$, from left to right. We begin by reading into internal memory the first $2k$ columns of $D$. Next, we sort this set of elements in internal memory and we output the ordered list of the $km$ smallest elements (holding back a small buffer of elements if we don't fill up the last block). Then we read in the next $k$ columns of $D$ (possibly reading in $k$ additional blocks for columns beyond this, depending on the block boundaries), and repeat the sorting of the items in internal memory and outputting the smallest $km$ elements in order. At any point in this algorithm, we may need to have up to $2km + (m+2)B$ elements in internal memory, which, under a reasonable tall cache assumption (say $M > 3B^4$), will indeed fit in internal memory. We continue in this way until we process all the elements in $D$. Note that, since we process the items in $D$ from left to right in a block fashion, for all possible data values, the algorithm is data-oblivious with respect to I/Os.

Consider the correctness of this method. Let $D_1, D_2, \ldots, D_l$ denote the subarrays of $D$ of size $m \times k$ used in our algorithm. By a slight abuse of notation, we have that $D_1 \leq D_3$, by Lemma 3. Thus, the smallest $mk$ items in $D_1 \cup D_2$ are less than or equal to the items in $D_3$. Likewise, these $mk$ items are obviously less than the largest $mk$ items in $D_1 \cup D_2$. Therefore, the first $mk$ items output by our algorithm are the smallest $mk$ items in $D$. Inductively, then, we can now ignore these smallest $mk$ items and repeat this argument with the remaining items in $D$. Thus, we have the following.

**Theorem 3.** *Given an array $A$ of size $N$ comparable items, we can sort $A$ with a data-oblivious external-memory algorithm that uses $O((N/B) \log^2_{M/B}(N/B))$ I/Os, under a tall-cache assumption ($M > 3B^4$).*

**Theorem 4.** *Given a set of $n$ distinct items and corresponding hash values, there is a data-oblivious algorithm for constructing a two-table cuckoo hashing scheme of size $O(n)$ with a stash of size $s = O(\log n)$ whenever this stash size is sufficient, using a private memory of size $O(n^{1/r})$, for a given fixed constant $r > 1$, in $O(n + s)$ time.*

*Proof.* Combine Theorems 2 and 3, with $N = n + s$, $B = 1$, and $M \in O(n^{1/r})$. □

## 5 Oblivious RAM Simulations

Our data-oblivious simulation of a non-oblivious algorithm on a RAM follows the general approach of Goldreich and Ostrovsky [5], but differs from it in some important ways, most particularly in our use of cuckoo hashing. We assume throughout that Alice encrypts the data she outsources to Bob using a probabilistic encryption scheme, so that multiple encryptions of the same value are extremely likely to be different. Thus, each time she stores an encrypted value, there is no way for Bob to correlate this value to other values or previous values. So the only remaining information that needs to be protected is the sequence of accesses that Alice makes to her data.

Our description simultaneously covers two separate cases: the constant-sized private memory case with very high probability amortized time bounds, and the case of private memory of size $O(n^{1/r})$ for some constant $r > 1$. The essential description is the same for these settings, with slight differences in how the hash tables are structured as described below.

We store the $n$ data items in a hierarchy of hash tables, $H_k$, $H_{k+1}$, . . ., $H_L$, where $k$ is an initial starting point for our hierarchy and $L = \log n$. Each table, $H_i$, has capacity for $2^i$ items, which are distributed between "real" items, which correspond to memory cells of the RAM, plus "dummy" items, which are added for the sake of obliviousness to make the number of items stored in $H_i$ be the appropriate value. The starting table, $H_k$, is simply an array that we access exhaustively with each RAM memory read or write. The lower-level tables, $H_{k+1}$ to $H_l$, for $l$ determined in the analysis, are standard hash tables with $H_i$ having $2^{i+1}$ buckets of size $O(\log n)$, whereas higher-level tables, $H_{l+1}$ to $H_L$, are cuckoo hash tables, with $H_i$ having $(1 + \epsilon)2^{i+2}$ cells and a stash of size $s$, where $s$ is determined in the analysis and $\epsilon > 0$ is a constant. The sizes of the hash tables in this hierarchy increase geometrically; hence the total size of all the hash tables is proportional to the size of $H_L$, which is $O(n)$. Our two settings will differ in the starting points for the various types of hash tables in the hierarchy as well as the size of the stash associated with the hash tables.

For each $H_i$ with $i < L$ we keep a count, $d_i$, of the number of times that $H_i$ has been accessed since first being constructed as an "empty" hash table containing $2^i$ dummy values, numbered consecutively from $-1$ to $-2^i$. For convenience, in what follows, let us think of each hash table $H_i$ with $i > l$ as being a standard cuckoo hash table, with a stash of size $s = \Theta(\log n)$ chosen for the sake of a desired superpolynomially-small error probability. Initially, every $H_i$ is an empty cuckoo hash table, except for $H_L$, which contains all $n = 2^L$ initial values plus $2^L$ dummy values.

We note that there is, unfortunately, some subtlety in the geometric construction of hash tables in the setting of constant-sized private memory with very high probability bounds. A problem arises in that for small hash tables, of size say polylogarithmic in $n$, it is not clear that appropriate very high probability bounds, which required failures to occur with probability inverse superpolynomial in $n$, hold with logarithmic sized stashes. Such results do not follow from previous work [10], which focused on constant-sized stashes. However, to keep the simulation time small, we cannot have larger than a logarithmic-sized stash (if we are searching it exhaustively), and we require small initial levels to keep the simulation time small.

We can show that we can cope with the problem by extending results from [10] that logarithmic sized stashes are sufficient to obtain the necessary probability bounds for hash tables of size that are polylogarithmic in $n$. In order to start our hierarchy with small, logarithmic-sized hash tables, we simply use standard hash tables as described above for levels $k+1$ to $l = O(\log \log n)$ and use cuckoo hash tables, each with a stash of size $s = O(\log n)$, for levels $l+1$ to $L$.

*Access Phase.* When we wish to make an access for a data cell at index $x$, for a read or write, we first look in $H_k$ exhaustively to see if it contains an item with key $x$. Then, we initialize a flag, "found," to **false** iff we have not found $x$ yet. We continue by performing an access[2] to each hash table $H_{k+1}$ to $H_L$, which is either to $x$ or to a dummy value (if we have already found $x$).

Our privacy guarantee depends on us never repeating a search. That is, we never perform a lookup for the same index, $x$ or $d$, in the same table, that we have in the past. Thus, after we have performed the above lookup for $x$, we add $x$ to the table $H_k$, possibly with a new data value if the access action we wanted to perform was a write.

*Rebuild Phase.* With every table $H_i$, we associate a *potential*, $p_i$. Initially, every table has zero potential. When we add an element to $H_k$, we increment its potential. When a table $H_i$ has its potential, $p_i$, reach $2^i$, we reset $p_i = 0$ and empty the unused values in $H_i$ into $H_{i+1}$ and add $2^i$ to $p_{i+1}$. There are at most $2^i$ such unused values, so we pad this set with dummy values to make it be of size exactly $2^i$; these dummy values are included for the sake of obliviousness and can be ignored after they are added to $H_{i+1}$. Of course, this could cause a cascade of emptyings in some cases, which is fine.

Once we have performed all necessary emptyings, then for any $j < L$, $\sum_{i=1}^{j} p_i$ is equal to the number of accesses made to $H_j$ since it was last emptied. Thus, we rehash each $H_i$ after it has been accessed $2^i$ times. Moreover, we don't need to explicitly store $p_i$ with its associated hash table, $H_i$, as $d_i$ can be used to infer the value of $p_i$.

The first time we empty $H_i$ into an empty $H_{i+1}$, there must have been exactly $2^i$ accesses made to $H_{i+1}$ since it was created. Moreover, the first emptying of $H_i$ into $H_{i+1}$ involves the addition of $2^i$ values to $H_{i+1}$, some of which may be dummy values.

When we empty $H_i$ into $H_{i+1}$ for the second time it will actually be time to empty $H_{i+1}$ into $H_{i+2}$, as $H_{i+1}$ would have been accessed $2^{i+1}$ times by this point—so we can simply union the current (possibly padded) contents of $H_i$ and $H_{i+1}$ together to empty both of them into $H_{i+2}$ (possibly with further cascaded emptyings). Since the sizes of hash tables in our hierarchy increase geometrically, the size of our final rehashing problem will always be proportional to the size of the final hash table that we want to construct. Every $n = 2^L$ accesses we reconstruct the entire hierarchy, placing all the current values into $H_L$. Thus, the schedule of table emptyings follows a data-oblivious pattern depending only on $n$.

*Correctness and Analysis.* To the adversary, Bob, each lookup in a hash table, $H_i$, is to one random location (if the table is a standard hash table) or to two random locations and the $s$ elements in the stash (if the table is a cuckoo hash table), which can be

1. a search for a real item, $x$, that is not in $H_i$,
2. a search for a real item, $x$, that is in $H_i$,
3. a search for a dummy item, $d_i$.

[2] This access is actually two accesses if the table is a cuckoo hash table.

Moreover, as we search through the levels from $k$ to $L$ we go through these cases in this order (although for any access, we might not ever enter case 1 or case 3, depending on when we find $x$). In addition, if we search for $x$ and don't find it in $H_i$, we will eventually find $x$ in some $H_j$ for $j > i$ and then insert $x$ in $H_k$; hence, if ever after this point in time we perform a future search for $x$, it will be found prior to $H_i$. In other words, we will never repeat a search for $x$ in a table $H_i$. Moreover, we continue performing dummy lookups in tables $H_j$, for $j > i$, even after we have found the item for cell $x$ in $H_i$, which are to random locations based on a value of $d_i$ that is also not repeated. Thus, the accesses we make to any table $H_i$ are to locations that are chosen independently at random. In addition, so long as our accesses don't violate the possibility of our tables being used in a valid cuckoo hashing scheme (which our scheme guarantees with very high probability ) then all accesses are to independent random locations that also happen to correspond to locations that are consistent with a valid cuckoo scheme. Finally, note that we rebuild each hash table $H_i$ after we have made $2^i$ accesses to it. Of course, some of these $2^i$ accesses may have successfully found their search key, while others could have been for dummy values or for unsuccessful searches. Nevertheless, the collection of $2^i$ distinct keys used to perform accesses to $H_i$ will either form a standard hash table or a cuckoo graph that supports a cuckoo hash table, with a stash of size $s$, w.v.h.p. Therefore, with very high probability, the adversary will not be able to determine which among the search keys resulted in values that were found in $H_i$, which were to keys not found in $H_i$, and which were to dummy values.

Each memory access involves at most $O(s \log n)$ reads and writes, to the tables $H_k$ to $H_L$. In addition, note that each time an item is moved into $H_k$, either it or a surrogate dummy value may eventually be moved from $H_k$ all the way to $H_L$, participating in $O(\log n)$ rehashings, with very high probability. In the constant-memory case, by Theorem 2, each data-oblivious rehashing of $n_i$ items takes $O((n_i+s)\log(n_i+s))$ time. In addition, in this case, we use a stash of size $s \in O(\log n)$ and set $l = k + O(\log\log n)$. In the case of a private memory of size $O(n^{1/r})$, each data-oblivious rehashing of $n_i$ items takes $O(n_i)$ time, by Theorem 4. In addition, in this case, we can use a constant-size stash (i.e., $s = O(1)$), but start with $k = (1/r)\log n$, so that $H_k$ fits in private memory (with all the other $H_i$'s being in the outsourced memory). The use of a constant-sized stash, however, limits us to a result that holds with high probability, instead of with very high probability.

To achieve very high probability in this latter case, we utilize a technique suggested in [6]. Instead of using a constant-sized stash in each level, we combine them into a single logarithmic-sized stash, used for all levels. This allows the stash at any single level to possibly be larger than any fixed constant, giving bounds that hold with very high probability. Instead of searching the stash at each level, Alice must load the entire stash into private memory and rewrite it on each memory access. Therefore, we have the following.

**Theorem 5.** *Data-oblivious RAM simulation of a memory of size $n$ can be done in the constant-size private-memory case with an amortized time overhead of $O(\log^2 n)$, with very high probability. Such a simulation can be done in the case of a private memory of size $O(n^{1/r})$ with an amortized time overhead of $O(\log n)$, with very high probability, for a constant $r > 1$. The space needed at the server in all cases is $O(n)$.*

*Acknowledgments.* We thank Radu Sion and Peter Williams for helpful discussions about oblivious RAM simulation. This research was supported in part by the National Science Foundation under grants 0724806, 0713046, 0847968, 0915922, 0953071, and 0964473. A full version of this paper is available as [8].

## References

1. Ajtai, M.: Oblivious RAMs without cryptographic assumptions. In: Proc. of the 42nd ACM Symp. on Theory of Computing (STOC), pp. 181–190. ACM, New York (2010)
2. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. Cryptology ePrint Archive, Report 2010/108 (2010), `http://eprint.iacr.org/`
3. Drmota, M., Kutzelnigg, R.: A precise analysis of cuckoo hashing (2008) (preprint), `http://www.dmg.tuwien.ac.at/drmota/cuckoohash.pdf`
4. Feldman, J., Muthukrishnan, S., Sidiropoulos, A., Stein, C., Svitkina, Z.: On distributing symmetric streaming computations. In: ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 710–719 (2008)
5. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM 43(3), 431–473 (1996)
6. Goodrich, M., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless oblivious RAM simulation (unpublished manuscript)
7. Goodrich, M.T., Mitzenmacher, M.: MapReduce Parallel Cuckoo Hashing and Oblivious RAM Simulations. ArXiv e-prints, Eprint 1007.1259v1 (July 2010)
8. Goodrich, M.T., Mitzenmacher, M.: Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. ArXiv e-prints, Eprint 1007.1259v2 (April 2011)
9. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proc. ACM-SIAM Sympos. Discrete Algorithms (SODA), pp. 938–948 (2010)
10. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: cuckoo hashing with a stash. SIAM J. Comput. 39, 1543–1561 (2009)
11. Lee, D.-L., Batcher, K.E.: A multiway merge sorting network. IEEE Trans. on Parallel and Distributed Systems 6(2), 211–215 (1995)
12. McDiarmid, C.: Concentration. In: Habib, M., McDiarmid, C., Ramirez-Alfonsin, J., Reed, B. (eds.) Probabilistic Methods for Algorithmic Discrete Mathematics, pp. 195–248. Springer, Berlin (1998)
13. Pagh, R., Rodler, F.: Cuckoo hashing. Journal of Algorithms 52, 122–144 (2004)
14. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 502–519. Springer, Heidelberg (2010)
15. Williams, P., Sion, R.: Usable pir. In: NDSS. The Internet Society, San Diego (2008)
16. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: 15th ACM Conf. on Computer and Communications Security (CCS), pp. 139–148 (2008)