Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation[*]

Michael T. Goodrich [†] Michael Mitzenmacher[‡] Olga Ohrimenko[§] Roberto Tamassia[§]

Abstract

Motivated by cloud computing applications, we study the problem of providing privacy-preserving access to an outsourced honest-but-curious data repository for a group of trusted users. We show how to achieve efficient privacy-preserving data access using a combination of probabilistic encryption, which directly hides data values, and stateless oblivious RAM simulation, which hides the pattern of data accesses. We give a method with $O(\log n)$ amortized access overhead for simulating a RAM algorithm that has a memory of size n, using a scheme that is data-oblivious with very high probability. We assume that the simulation has access to a private workspace of size $O(n^\nu)$, for any given fixed constant $\nu > 0$, but does not maintain state in between data access requests. Our simulation makes use of pseudorandom hash functions and is based on a novel hierarchy of cuckoo hash tables that all share a common stash. The method outperforms all previous techniques for stateless clients in terms of access overhead. We also provide experimental results from a prototype implementation of our scheme, showing its practicality. In addition, we show that one can eliminate the dependence on pseudorandom hash functions in our simulation while having the overhead rise to be $O(\log^2 n)$.

1 Introduction

Companies offering outsourced data storage services are defining a growing industry, with competitors that include Amazon, Google, and Microsoft, which are providing outsourced data repositories for individual or corporate users, with prices that amount to pennies per gigabyte stored.

Clearly, the customers of such cloud computing services have an interest in security and privacy, particularly for proprietary data. As a recognition of this interest, we note that, as of November 2010, the Amazon S3 and Microsoft Azure cloud platform have achieved ISO 27001 certification and Google's cloud computing service has SAS70 certification. In spite of these certifications, the companies that provide outsourced data services nevertheless often have commercial interests in learning information about their customers' data. Thus, the users of such systems should also consider technological solutions for maintaining the privacy of their outsourced data in addition to the assurances that come from certifications and formal audits.

Of course, a key component for users to maintain the privacy of their data is for them to store their data in encrypted form, e.g., using a group key known only to the set of users. Simply encrypting the group's data is not sufficient to achieve privacy, however, since information about the data may be leaked by the pattern in which the users access it. Moreover, in a multi-user scenario the owner of the data repository can infer which customers are collaborating on the same documents. Indeed, such information is leaked even if the contents of the communications between the users and the data repository is encrypted.

1.1 Group Access to Outsourced Data. We are interested in technological solutions to the problem of protecting the privacy of a group's data accesses to an outsourced data storage facility. In this framework, we assume that a trusted group, G, of users shares a secret group key, K, with which they encrypt all their shared data that is stored at a semi-trusted data outsourcer, Bob. Furthermore, we assume that the users access their data according to a public indexing scheme, which Bob knows; hence, we can model Bob's memory, M, as in the standard RAM model (e.g., see [1, 8, 16, 18]).

Each time a user, Alice, in G, accesses Bob's memory, she specifies an index, i, and Bob responds by returning the contents of the ith memory location, $C = M[i]$. Alice then performs the following (atomic) sequence of operations:

1. She decrypts C using K, producing the plaintext value, $P = D_K(C)$, that was stored in encrypted form at index i by Bob.

[*]This research was supported in part by the National Science Foundation under grants 0721491, 0915922, 0953071, 0964473, 1011840, and 1012060, and by the Kanellakis Fellowship at Brown University.

[†]Dept. of Computer Science, University of California, Irvine.
[‡]Dept. of Computer Science, Harvard University.
[§]Dept. of Computer Science, Brown University.

2. She optionally changes the value of P, depending on the computation she is performing, producing the plaintext value, P'.

3. She encrypts P' using a probabilistic encryption scheme based on K, producing ciphertext $C' = E_K(P')$.

4. She returns C' to Bob and directs him to assign $M[i] \leftarrow C'$.

By using a probabilistic encryption scheme, the users in the group G ensure that Bob is computationally unable to determine the plaintext of any memory cell from that cell's contents alone. Also, it is unfeasible for Bob to determine whether two memory cells store encryptions of the same plaintext.

1.2 Stateless Oblivious RAM Simulation. In addition to using probabilistic encryption, the users in the group G also need to hide their data access patterns from Bob, so as to avoid inadvertent information leaks. To facilitate such information hiding, we formulate the privacy objective of the users in G in terms of the *stateless oblivious RAM simulation* problem.

In this framework, we model the group G as a single user, Alice, who has a register holding the key K and a CPU with a private memory (which is used only as a "scratch space"). Alice's interactions with Bob occur in discrete *episodes* in which she reads and writes a set of cells in his memory, using probabilistic encryption, as described above, to hide data contents. Alice's local memory may be used as a private workspace during any episode, but it cannot store any information from one episode to the next. This requirement is meant to model the fact that Alice is representing a group of users who do not communicate outside of their shared access to Bob's memory. That is, each episode could model a consecutive set of accesses from different users in the group G. Moreover, this requirement is what makes this framework "stateless," in that no state can be carried from one episode to the next (other than the state that is maintained by Bob).

To allow Alice, to perform arbitrary computations on the data outsourced to Bob, we assume that Alice is simulating a RAM computation. We also assume the service provider, Bob, is trying to learn as much as possible about the contents of Alice's data from the sequence and location of all of Alice's memory accesses. As mentioned above, however, he cannot see the content of what is read or written (since it is probabilistically encrypted). Moreover, Bob has no access to Alice's private memory. Bob is assumed to be an *honest-but-curious* adversary [11], in that he correctly performs all protocols and does not tamper with data.

We say that Alice's sequence of memory accesses is *data-oblivious* if the distribution of this sequence depends only on n, the size of the memory used by the RAM algorithm she is simulating, the size of her private memory, M, and the length of the access sequence itself. In particular, the distribution of Alice's memory accesses should be independent of the data values in the input. Put another way, this definition means that $\Pr(S \mid M)$, the probability that Bob sees an access sequence, S, conditioned on a specific configuration of his memory, M, satisfies $\Pr(S \mid M) = \Pr(S \mid M')$, for any memory configuration $M' \neq M$ such that $|M'| = |M|$.

Examples of data-oblivious access sequences for an array, A, of size n, in Bob's memory, include the following:

- Scanning A from beginning to end, accessing each item exactly once, for instance, to compute the minimum value in A, which is then stored in $A[1]$.

- Simulating a Boolean circuit, $\mathcal{C}$, with its inputs taken in order from the bits of A.

- Accessing the cells of A according to a random hash function, $h(i)$, as $A[h(1)]$, $A[h(2)]$, $\ldots$, $A[h(n)]$, or random permutation, $\pi(i)$, as $A[\pi(1)]$, $A[\pi(2)]$, $\ldots$, $A[\pi(n)]$.

Examples of computations on A that would *not* be data-oblivious include the following:

- Scanning A from beginning to end, accessing each item exactly once, to compute the index i of the minimum value in A, and then reading $A[i]$ and writing it to $A[1]$.

- Using a standard heap-sort, merge-sort, or quick-sort algorithm to sort A. (None of these well-known algorithms is data-oblivious.)

- Using values in A as indices for a hash table, T, and accessing them as $T[h(A[1])]$, $T[h(A[2])]$, $\ldots$, $T[h(A[n])]$, where h is a random hash function. For example, consider what happens if the values in A are all equal and how unlikely the resulting collision in T would be.

Note that this last example access pattern actually would be data-oblivious if the elements in A were always guaranteed to be distinct, assuming the random hash function, h, satisfies the standard assumptions of the random oracle model (e.g., see [6]).

1.3 Related Prior Results. Data-oblivious sorting is a fundamental problem (e.g., see Knuth [19]), with deterministic schemes giving rise to sorting networks, such as the impractical $O(n \log n)$ AKS network [3, 4, 26, 30] as well as practical, but theoretically-suboptimal, sorting networks [21, 29]. Randomized data-oblivious sorting algorithms running in $O(n \log n)$ time and succeeding with high probability[1] are studied by Leighton and Plaxton [22] and Goodrich [13]. In addition, data-oblivious sorting is finding applications to privacy-preserving secure multi-party computations [33], and it is used in all the known oblivious RAM simulation schemes (including the ones in this paper).

In early work on the topic of oblivious simulation, Pippenger and Fischer [28] show that one can simulate a Turing machine computation of length n with an oblivious Turing machine computation of length $O(n \log n)$, that is, they achieve an amortized $O(\log n)$ time and space overhead for this oblivious simulation.

More recently, Goldreich and Ostrovsky [12] show that a RAM computation using space n can be simulated with an oblivious RAM with an amortized time overhead of $O(\log^3 n)$ per step of the original RAM algorithm and space overhead of $O(\log n)$. Goodrich and Mitzenmacher [14] improve this result by showing that any RAM algorithm, $\mathcal{A}$, can be simulated in a data-oblivious fashion, with very high probability, in an outsourced memory so that each memory access performed by $\mathcal{A}$ has a time overhead of $O(\log^2 n)$, assuming Alice's private memory has size $O(1)$. Their scheme has a space overhead of $O(1)$. Incidentally, Pinkas and Reinman [27] also claim an oblivious RAM simulation result having a time overhead of $O(\log^2 n)$, but there is a flaw in this version of their scheme, as recently shown by Kushilevitz *et al.* [20], who also show that techniques from [14] can be extended to obtain an overhead of $O(\log^2 n/ \log \log n)$.

In addition to these stateless oblivious RAM simulation schemes, Williams and Sion [34] show how to simulate a RAM computation with an oblivious RAM where the data owner, Alice, has a stateful private memory of size $O(\sqrt{n})$, achieving an expected amortized time overhead of $O(\log^2 n)$ using $O(n \log n)$ memory at the data provider. In addition, Williams *et al.* [35] claim a method that uses an $O(\sqrt{n})$-sized private memory and has $O(\log n \log \log n)$ amortized time overhead, but Pinkas and Reinman [27] have raised concerns with the assumptions and analysis of this result.

A few recent approaches consider a stateful RAM simulation, i.e. where Alice maintains a state from one episode to another. A RAM simulation by Goodrich and Mitzenmacher [14] achieves an overhead of $O(\log n)$ and is oblivious with very high probability assuming a private cache of size $O(n^\nu)$, for any given fixed constant $\nu > 0$, which maintains state for Alice. Boneh *et al.* [7] propose a scheme that achieves an amortized overhead of $O(1)$ but using a cache of size $O(\sqrt{n \log n})$, which also maintains state. The scheme in [32] incurs $O(\log^2 n)$ amortized cost but is $O(\sqrt{n})$ in the worst case. However, maintaining a state is essential to the efficiency of all three of these simulation schemes. Thus, these methods are not applicable to the problem of providing privacy-preserving group access to an outsourced data repository.

Returning to stateless oblivious RAM simulation, we note that Ajtai [2] has a recent oblivious RAM simulation result that shows that a polylogarithmic factor overhead in time and space is possible without cryptographic assumptions about the existence of random hash functions, as is done in the previous oblivious RAM simulation cited above. Damgård *et al.* [9] improve this result further, showing that a time overhead of $O(\log^3 n)$ is possible for oblivious RAM simulation without using random functions.

We also note that subsequent to this work, Goodrich *et al.* [15] and Shi *et al.* [31] provide ORAM solutions that target reducing the worst-case access overhead. In [15] the authors deamortize the oblivious RAM construction from this paper and achieve $O(\log n)$ overhead on every access. Shi *et al.* [31] provide a novel construction that uses $O(n \log n)$ server storage and incurs $O(\log^3 n)$ overhead on each access.

In addition to the above-mentioned upper-bound results, Beame and Machmouchi [5] show that if the additional space utilized in the simulation (besides the space for the data itself) is sufficiently sublinear, then the overhead for oblivious RAM simulation has a super-logarithmic lower bound. Such bounds don't apply, of course, to a simulation that uses $O(n)$ additional memory, as is common in the efficient schemes mentioned above.

We provide a summary of the Oblivious RAM simulation schemes and compare with ours in Table 1. Note that the schemes that maintain a state cannot be used to hide a pattern of access by a group of users which is one of the challenges we address in this paper.

1.4 Our Results.

We give an efficient method for simulating any RAM algorithm, $\mathcal{A}$, in a stateless fashion with a time overhead of $O(\log n)$, using an access sequence that is data-oblivious with very high probability, where n is the size of the RAM memory. This result improves a previously best known time overhead of $O(\log^2 n/ \log \log n)$ by Kushilevitz *et al.* [20] for the stateless user scenario. Our methods assume that Al-

[1]In this paper, we take the phrase "with very high probability" to mean that the probability is at least $1 - O(1/n^d)$, for any given fixed constant $d \geq 1$.

Table 1: Comparison of schemes for oblivious RAM simulation.

	User Memory	User State Size	Server Storage	Amortized Access Overhead
Goldreich-Ostrovsky [12]	$O(1)$	-	$O(n \log n)$	$O(\log^3 n)$
Williams-Sion [34]	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n \log n)$	$O(\log^2 n)$
Goodrich-Mitzenmacher [14]	$O(1)$	-	$O(n)$	$O(\log^2 n)$
Kushilevitz *et al.* [20]	$O(1)$	-	$O(n)$	$O(\log^2 n / \log\log n)$
Stefanov *et al.* [32]	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n)$	$O(\log^2 n)$
Goodrich-Mitzenmacher [14]	$O(n^\nu)$	$O(n^\nu)$	$O(n)$	$O(\log n)$
Boneh *et al.* [7]	$O\left(\sqrt{n \log n}\right)$	$O\left(\sqrt{n \log n}\right)$	$O(n)$	$O(1)$
Our result	$\boldsymbol{O(n^\nu)}$	-	$\boldsymbol{O(n)}$	$\boldsymbol{O(\log n)}$
Our result (w/o random oracle)	$O(n^\nu)$	-	$O(n)$	$O(\log^2 n)$

ice has a private memory of size $O(n^\nu)$, for any given fixed constant $\nu > 0$, but she uses this storage only as a private "scratch space" to support computations she performs during each episode (previous methods did not use such a private memory or assumed such a storage could hold a considerable amount of state). Alice is not allowed to maintain state in her private memory from one episode to the next. Thus, this simulation scheme is applicable to the problem of simulating access to a shared data repository by a group of cooperating users that all share a secret key. Moreover, the assumption about the size of Alice's scratch space is motivated by the fact that even handheld devices have a reasonable amount of local memory. For example, if we were to set $\nu = 1/4$, then our simulation would allow a collection of devices having memories with sizes on the order of one megabyte to support privacy-preserving access to an outsourced data repository whose size is on the order of one yottabyte.

Like the previous oblivious RAM simulation schemes mentioned above, our scheme uses a hierarchy of hash tables, together with a small set of pseudorandom hash functions, to obfuscate the access pattern of the algorithm $\mathcal{A}$ (which need not be specified in advance). The main idea of our scheme is to maintain these hash tables as cuckoo hash tables that all share a single stash of size $O(\log n)$. While conceptually simple, this approach requires a new, non-trivial analysis for a set of cuckoo tables sharing a common stash. The idea of a shared stash is novel to this paper but has already been cited in recent work by [14] and [20] for a single-user scenario. In addition, an important technical detail that simplifies our construction is that we make no use of so-called "dummy" elements, whereas the previous schemes used such elements.

In practice, the set of pseudorandom hash functions could be implemented using, e.g., keyed SHA-256 functions [10]. Nevertheless, we also show that our construction can be used to simulate a RAM computation with an overhead of $O(\log^2 n)$ without the use of pseudorandom functions, which may be of some theoretical interest.

Finally, we provide experimental results for a simulation of our scheme, which show the practical effectiveness of our approach. In particular, our experimental prototype simulates the interaction between a user and a data repository allowing us to compare the theoretical overhead of our approach of $O(\log n)$ to what we can expect in practice. We show that amortized overhead of each request does not exceed $2 \log n$ additional accesses to data repository for $n \geq 10^5$ items. Additionally, we give the threshold values at which the shared stash becomes effective for our scheme.

2 Theory Background

For our results, we rely on general methods for data-oblivious simulation of a non-oblivious algorithm on a RAM. As mentioned above, the seminal theoretical framework for such simulations was presented by Goldreich and Ostrovsky [12], who store keys in a hierarchy of hash tables of increasing size, each being twice the size of the previous one. For n items there are $O(\log n)$ levels, each level being a standard hash table with 2^i buckets for some i, and each bucket containing up to $O(\log n)$ keys in order to cope with collisions within the hash table. In this construction the total size of all the tables is $O(n \log n)$. To perform a lookup, the first level is scanned sequentially, and in each of the other levels, a bucket chosen by the hash function for that level acting on the key (or, if the item is found at an earlier level, a random dummy key) is scanned. The item is subsequently re-encrypted and re-inserted into the first level. It is important to note that at all levels a bucket is scanned even if the key is found early, to maintain obliviousness. As levels fill, keys must be shifted down to subsequent levels. The details of the original scheme are rather complex; for further details see the original paper [12].

Recently, a more efficient simulation approach for this problem was outlined by Goodrich and Mitzenmacher [14]. The primary difference in this new line of work is the use of *cuckoo hash tables* in place of the standard hash tables used originally in [12]. We therefore now present some background on *cuckoo hashing.*

As introduced by Pagh and Rodler [25], in standard cuckoo hashing we utilize two tables, each with m cells, with each cell capable of holding a single key. We make use of two hash functions h_1 and h_2 that we assume can be modeled as completely random hash functions. The tables store up to n items, where $m = n(1+\epsilon)$ for some constant $\epsilon > 0$, yielding a load of (just) less than $1/2$; keys can be inserted or deleted over time as long as this restriction is maintained. A key x (which we may also refer to as an "item" or "element") that is stored in the hash tables must be located at either $h_1(x)$ or $h_2(x)$. As there are only two possible locations for a key, lookups take constant time. To insert a new key x, we place x in the cell $h_1(x)$. If the cell had been empty, the operation is complete. Otherwise, key y previously in the cell is moved to $h_2(y)$. This may in turn require another key to be moved, and so on, until a key is placed in an empty cell. We say that a failure occurs if, for an appropriate constant c, after $c \log n$ steps this process has not successfully terminated. Suppose we insert an nth key into the system. It is known that the expected time to insert a new key is bounded above by a constant and the probability that a new key causes a failure is $\Theta(1/n^2)$ (both results depend on ϵ).

There are several natural variations of cuckoo hashing, many of which are described in a survey article by Mitzenmacher [23]. For our purposes, it suffices to understand standard cuckoo hashing, along with the idea of a *stash* [17].

A stash represents additional memory where keys that would cause a failure can be placed in order to avoid the failure; with a stash, a failure occurs only if the stash itself overflows. As shown in [17], the failure probability when inserting the nth key into a cuckoo hash table can be reduced to $O(1/n^{k+2})$ for any constant k by using a stash that can hold k keys. Using this allows us to use cuckoo hash tables for any polynomially bounded number of inserts and deletions using only a constant-sized stash. To search for an item, we must search both the two table locations and the k stash locations. In the context of oblivious simulation, we can search the stash simply by reading each stash location.

As we have stated, however, in order to perform our oblivious simulation, we will make use of a hierarchy of cuckoo tables to hold n items. The smallest of these hash tables may be much smaller than n, which can lead to a potential leakage of information in our setting if we are not careful. For example, if the smallest hash table is of size x, then even using a stash of size k leads to a failure probability of $O(1/x^{k+2})$. If x is for example polylogarithmic in n, then for any constant k, the failure probability is $\Omega(1/n)$, and therefore over the insertion of n items, we would expect failures to occur. Our solution to this problem is to introduce a stash that is shared by all the cuckoo tables; any one of them can utilize a great deal of this stash, but as we show in the following theorem, it is nevertheless highly unlikely that the collective overflows from all the cuckoo tables will also overflow this stash.

Theorem 2.1: *Given a hierarchy of cuckoo hash tables $T_1, \ldots, T_\ell$, where T_i contains $O(2^i \log n)$ elements and $\ell = O(\log n)$, a shared stash of size $O(\log n)$ is enough to avoid overflows with high probability.*

Proof. The probability that the stash for a cuckoo hash table of size x cells (where x is $\Omega(\log^7 n)$) exceeds a total size s is $x^{-\Omega(s)}$ [14]. Further, as long as the hashes for a cuckoo hash table at each level are independent, we can treat the required stash size at each level as independent, since the number of items placed in the stash at a level is then a random variable dependent only on the number of items appearing in that level.

Now consider any point of our construction and let S_i be the number of items at the ith level that need to be put in the stash. It is apparent that S_i has mean less than 1 and tails that can be dominated by a geometrically decreasing random variable. This is sufficient to apply standard Chernoff bounds. Formally, let $X_1, X_2, \ldots, X_\ell$ be independent random variables with mean 1 geometrically decreasing tails, so that $X_i = j$ with probability $1/2^j$ for $j \geq 1$. Then the calculations of [14] imply that the X_i stochastically dominate the S_i, and we can now apply standard Chernoff bounds for these random variables. Specifically, noting that X_i can be interpreted as the number of fair coin flips until the first heads, we can think of the sum of the X_i as being the number of coin flips until the ℓth head, and this dominates the number of items that need to be placed in the stash at any point. Since $\ell = O(\log n)$ then for any constant γ_1 there exists a corresponding constant γ_2 such that the ℓth head occurs by the $(\gamma_2 \log n)$'th flip with probability at least $1 - 1/n^{\gamma_1}$. (See, for example, [24, Chapter 4].) Hence,

$$\Pr\left(\sum_{i=1}^{\ell} S_i > \ell\right) \le 1 - \Pr\left(\sum_{i=1}^{\ell} X_i \le \gamma_2 \log n\right) \le 1/n^{\gamma_1}.$$

Therefore we can handle any polynomial number of insertions with high probability, using a stash of size only $O(\log n)$ that holds items from all levels of our construction.

3 Simulating a RAM Algorithm Obliviously

In this section, we describe and analyze two schemes for stateless oblivious RAM simulation.

3.1 Simulation Using Pseudorandom Functions. We begin with a construction that uses pseudorandom functions and is secure against a polynomially bounded adversary.

Given a RAM algorithm, $\mathcal{A}$, the main goal of our oblivious simulation of $\mathcal{A}$ is to hide the pattern of memory accesses that are made by $\mathcal{A}$. As mentioned in Section 2, we follow the general framework introduced by Goldreich and Ostrovsky [12], which uses a hierarchy of hash tables.

Let n be the number of memory cells of the RAM. We view each such cell as an item consisting of a pair (x, v), where $x \in \{0, \cdots, n-1\}$ is the index and v is the corresponding value. Our data structure stored at the server has three components, illustrated in Figure 1. The first component is a cache of size $O(\log n)$, denoted by Q. The second component is a hierarchy of cuckoo hash tables, $T = (T_1, \ldots, T_L)$, where the size of T_1 is twice the size of Q, each table T_{i+1} is twice the size of table T_i, and T_L is the first table in the sequence of size greater than or equal to n. Thus, L is $O(\log n)$. The third component is a stash, S, shared between all the above cuckoo tables.

RAM items are stored in the data structure in encrypted form. We use a semantically secure probabilistic encryption scheme, which results in a different ciphertext for the same item each time it is re-encrypted. Also, the server is unable to determine whether two ciphertexts correspond to the same item. The stash S is handled in a similar manner whenever we search in it for an item.

We use a family of pseudorandom functions parameterized by a secret value, k_i, for each table, T_i, such that no value k_i is revealed to the server. In particular, k_i is stored in encrypted form for each table T_i, so that each user can read k_i, decrypt it, and then use it to provide the two hash functions, h_1^i and h_2^i, employed by the cuckoo table, T_i, to determine the location of items. In particular, a memory item (x, v) is mapped to locations $h_1^i(x)$ and $h_2^i(x)$ in T_i by the cuckoo scheme (and stored in one of these two locations or in the common stash, S).

The data structure is initialized by storing all the n RAM items into cuckoo table T_L. Each memory access defined by algorithm $\mathcal{A}$ corresponds to an *episode* in our simulation. An episode consists of two phases, an *access phase* and a *rebuild phase.*

Suppose algorithm $\mathcal{A}$ calls for an access to memory item (x, v). The access phase consists of a search for x in the cache, Q, then in the stash, S, and continues with a two-cell cuckoo lookup in each of T_1 to T_L until we find the first item with index x. Once we have found this item, we have achieved the goal of our search, of course. Nevertheless, for the sake of obliviousness, we simulate continuing the search throughout the entire data structure. Namely, we always traverse completely Q and S, and we perform two-cell cuckoo accesses in tables T_1 through T_L. However, after the item is found, we simply access two distinct, independent uniformly chosen random locations in each remaining cuckoo table.

Once we have completed the access phase, which takes $O(\log n)$ time, we then switch to the rebuild phase. We begin by adding or replacing a copy of the found item into cache Q, possibly changing its value in the case of a write operation. To assure obliviousness, we exhaustively scan Q in a sequential manner and re-encrypt and rewrite all its items. Thus, the server cannot distinguish which item was accessed and whether it was modified.

We note briefly that if the item is in the stash, we can obliviously remove it from the stash when placing it into Q, to help make sure the stash does not overflow. One natural approach is to have stash cells have an associated "clean" or "dirty" bit, which is encrypted along with the rest of the item. A clean cell can store an item; a dirty cell is currently being utilized. When an item is found and replaced into Q, we can set the cell to clean in the stash.

After adding enough items, cache Q will eventually overflow. We remedy the overflow by moving all the elements of Q to cuckoo table T_1, including those associated with empty locations. However, in order to maintain obliviousness, we do not wait for an overflow to occur and instead perform the move after a number of accesses equal to the size of Q. The moving down of elements cascades down through the hierarchy of cuckoo tables at a fixed schedule by periodically moving the elements of level $i-1$ into T_i at the earliest time T_{i-1} could have become full. Note that this may include moving some of the elements from stash S, i.e. elements that did not fit in T_{i-1} the last time it was built. Now suppose that we are going to move elements into table T_i

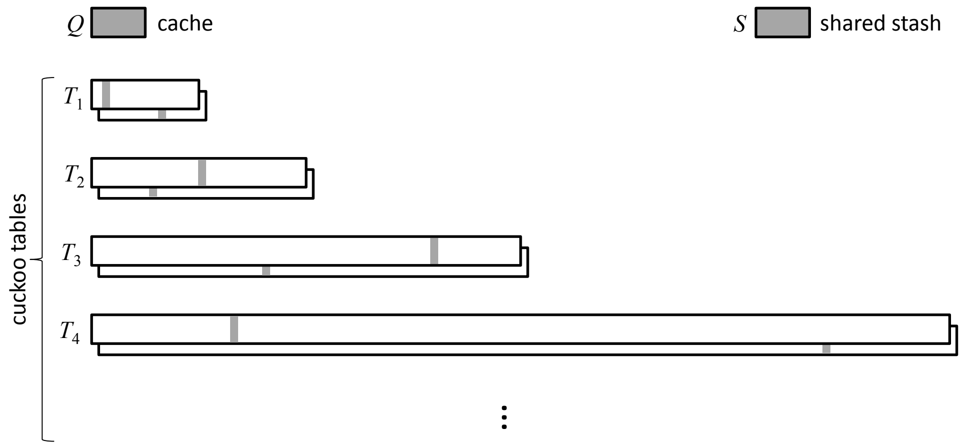


Figure 1: Illustration of the data structure stored at the server for oblivious RAM simulation using pseudorandom functions. In the access phase of the simulation, all the items in the cache, Q and the stash, S, plus two items for each cuckoo table T_i are read by the server. The locations accessed by the server are visualized as gray-filled rectangles.

for the second time, then we instead move the elements into table T_{i+1}. Moreover, we continue applying this rule for $i = 1, 2, \ldots$, until we are copying the elements into a table for the first time or we reach T_L. Thus, the process of copying elements into a cuckoo table occurs at deterministic instances, depending only on the place we currently are at in the access sequence specified by algorithm $\mathcal{A}$.

In order to move m elements from level i into a cuckoo hash table T_{i+1} obliviously, we use an algorithm of [14] to obliviously sort the items using $O(m)$ accesses to the outsourced memory, assuming we have a private workspace of size $O(n^\nu)$, for some constant $\nu > 0$, and $m \geq \log n$, which is always true in our case. This allows us to remove duplicate items and use another algorithm of [14] to obliviously construct a cuckoo table of size m and an associated stash, S', of size $O(\log n)$ in $O(m)$ time, with very high probability, while utilizing the private workspace of size $O(n^\nu)$. Given this construction, we then read S and S' into our private workspace, remove any duplicates and merge them into a single stash S (which will succeed with very high probability, based on Theorem 2.1), and write S back out in a straightforward oblivious fashion. Note that in order to assure obliviousness in subsequent lookups, table T_{i+1} is rebuilt using two new pseudorandom hash functions selected by the client by replacing parameter k_{i+1} with a new secret value.

Theorem 3.1: *Our data-oblivious RAM simulation of memory of size n using pseudorandom functions has an amortized time overhead of $O(\log n)$, with very high probability, using $O(1)$ space overhead and assuming that a client has access to private workspace of size $O(n^\nu)$, $\nu > 0$.*

3.2 Simulation Without Pseudorandom Functions. We can adapt our simulation to avoid the use of random functions by employing an elegant trick due to Damgård *et al.* [9], albeit now further simplified to avoid the use of dummy nodes, which would add an extra level of complication that our scheme doesn't require.

The main idea is to place a complete binary tree, B, on top of all the memory cells used in the algorithm $\mathcal{A}$, and access each memory cell x by performing a binary search from the root of B to the leaf node corresponding to x. That is, we associate each memory cell item used by $\mathcal{A}$ with a leaf of B, define B to have height $\lceil \log n \rceil$, and include information at each internal node v of B so that a search for x can determine in $O(1)$ time whether to proceed with the left child or right child of v. In our case, we store each of the nodes of B in our hierarchy of tables, similar to what is described above, with the shared stash, S, the cache, Q, and the set of cuckoo tables, T_1 to T_L. (See Figure 2.)

The main difference of this scheme with that given above is that in this case we no longer use random hash functions, h_1^i and h_2^i, to determine the locations of each element x in a cuckoo table T_i. Instead, we simply choose two distinct, independent uniformly random locations, i_1 and i_2, in the respective two sides of T_i and associate these with x as a tuple (x, i_1, i_2), which now represents the element x in our table.

Initially, all the nodes of B are stored in this way in T_L, and for each such internal node v, we include

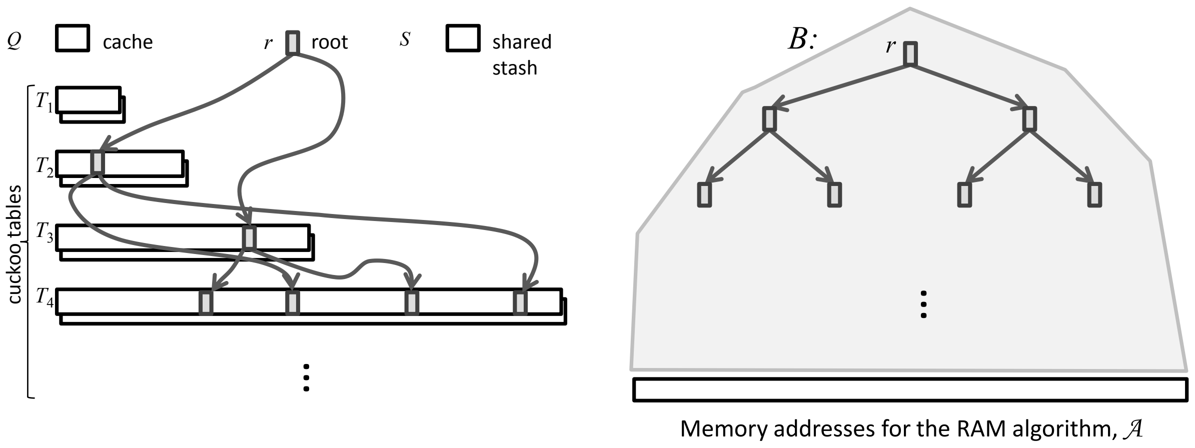


Figure 2: Illustration of the data structure stored at the server for oblivious RAM simulation without using pseudorandom functions. The binary tree is shown conceptually on the right and in terms of the storage locations of its nodes on the left. The storage locations for the nodes of the binary tree, B, are visualized as gray-filled rectangles. In the access phase of the simulation for a non-root node, all the items in the cache, Q and the stash, S, plus two items for each cuckoo table T_i are read by the server.

in v's record pointers to the two random indices (and table index) for v's left child and pointer to the two random indices (and table index) for v's right child. Such pointers can be built obliviously by $O(1)$ calls to oblivious sorting once we have placed all the nodes into T_L. Moreover, we will maintain such pointers throughout our simulation. In addition, we store the root r of B separately, as it is accessed in every step of our simulation.

Let us consider, therefore, how an access now occurs. The critical property, which we maintain inductively, is that, for each node v in B, which, say, is stored in T_j as its earliest (highest) location in our hierarchy, all the ancestors of B are stored in the tables $T_1, \ldots, T_j$, or in r, S, or Q.

Our access for a memory cell x now occurs as a root-to-leaf search in B. We begin by searching in r to identify the two random indices and the table index for each of r's children. Based on the value of x, we need to search next for either the left or right child of r, so let i_1 and i_2 be the two random indices for this node, w, and let j be the index of the highest table T_j storing w (with $j = 0$ if $w \in Q$ and $j = -1$ if $w \in S$). We next search in S and Q for w, and then proceed in T_1 through T_L. Of course, we already know the table where we will find w. So, for each table T_k, $k \neq j$, we simply access two random locations in T_k for the sake of obliviousness. For T_j itself, we look in locations $T_j[i_1]$ and $T_j[i_2]$ to find the cell containing the record for w. If w is not a leaf node, we repeat the above lookup search for the appropriate child of w that will lead us to the node storing x.

Once we have done our lookup for x, and have accessed a root-to-leaf set of nodes, $W = \{w_1, w_2, \ldots, w_{\log n}\}$ in the process, we perform a rebuild phase for W, as in the above construction based on random hash functions, except that we use random locations for all the nodes we move rather than use random functions. Note that by our induction hypothesis, if we move a set of nodes into a table T_i, then all the pointers for these nodes are either in T_i itself (hence, can be identified after $O(1)$ calls to oblivious sorting, which takes $O(n)$ memory accesses by the algorithm of [14]) or at lower levels in the hierarchy (hence, these pointers don't change by our move into T_i). Moreover, all the nodes of W move as a group. Thus, any root-to-leaf path in B must be stored in the tables T_1 to T_L, plus the queue Q and stash S, in a way that satisfies our induction hypothesis.

The lookup for an element x now requires searching for $O(\log n)$ nodes of B in our hierarchy, which costs an amortized overhead of $O(\log n)$ time each. Thus, each lookup costs us an amortized overhead of $O(\log^2 n)$ time. The obliviousness of this simulation follows from an argument similar to that given above for the obliviousness for our method that uses random hash functions. Therefore, we can perform a stateless oblivious RAM simulation without using random hash functions with an amortized time overhead of $O(\log^2 n)$, assuming a private workspace of size $O(n^\nu)$ for some constant $\nu > 0$.

Theorem 3.2: *Our data-oblivious RAM simulation of memory of size n without pseudorandom functions has an amortized time overhead of $O(\log^2 n)$, with very high probability, using $O(1)$ space overhead and assuming that a client has access to private workspace of size $O(n^\nu)$, $\nu > 0$.*

4 Performance

In this section, we discuss the practical performance of our oblivious RAM simulation scheme.

4.1 ORAM Simulator. We have implemented a prototype of our method for oblivious RAM simulation based on pseudorandom functions (Section 3.1) with the goal of estimating the access overhead associated with every request in practice.

Our prototype simulates an interaction between a user and a data repository. The user outsources n data items to the data repository and issues a sequence of item requests. Each data item is of size b and the user is assumed to have a private workspace that can fit $p = O(\sqrt{n})$ items. This space is used only during rebuilds hence our system can be extended to a multi-user case. We recall that we store into the repository a stash S of size $s \log n$, a cache Q of size $\log n$ and a hierarchy of $O(\log n)$ cuckoo hash tables, $T_1, \cdots, T_L$, where L is the smallest i such that $2^{i-1} \log n \geq n$. Each table T_i, $i < L$, consists of two hash tables of size $(1+\epsilon)2^{i-1} \log n$ with hash functions h_1^i and h_2^i. Table T_L contains two hash tables of size $(1+\epsilon)n$, enough to store all n items. We have selected parameters $\epsilon = 0.1$ and $s = 2$ based on our stash experiments below. We use algorithms from [14] to implement oblivious external-memory merge sort and oblivious construction of a cuckoo hash table via map-reduce simulation.

Our prototype is implemented in Java. To generate hash functions we use a variation of a method recommended in [10], where

$$h_1^i(x) = \text{SHA256}(x \mid\mid \text{seed}_i^1) \bmod n,$$

and similarly for h_2^i. The seeds are 64-bit long and are obtained using a SHA256 hash chain starting from an initial seed. The simulation starts with all n items placed in the last cuckoo table T_L. The user then generates a sequence of random requests that ends after the reshuffle of the last table, T_L, is performed. During the simulation, we count the number of read and write accesses by the user to the data repository. In Table 2a, we compare the average number of accesses for several values of n versus $\log n$, the amortized asymptotic overhead of our approach. From the experimental results, we see that for $n \geq 10^5$ average overhead per request is less than $2 \log n$. Since the performance of oblivious sort depends on the size of the user's workspace, p, we experiment with $p = \sqrt{n}$ and $p = 2\sqrt{n}$.

Finally, we compare the amortized overhead of our method with the results reported by Stefanov *et al.* [32] in Table 2b. First, note that our method was targeted to a multi-user scenario and hence clients are not allowed to maintain a state between requests while this is not the case for the *stateful* method in [32]. We observe that our stateless approach achieves practical results that are comparable to those obtained in [32] using the more powerful stateful model. Moreover, since the size of the state impacts the performance of the method of [32], they use blocks of size up to 16MB in their simulation to achieve a smaller overhead.

4.2 Stash Size. In this section we estimate the size of the stash S needed to avoid failures during the rebuild phase. A failure can happen when we move elements from tables $T_1, \ldots, T_{i-1}$ to T_i and the stash overflows, in which case we need to build table T_i again. We show that for a small constant s, a stash of size $s \log n$, is enough to avoid failures.

We ran our simulation for up to $n = 10^6$ items and a varying number of requests. We use a value of ϵ of 0.1 and 0.2. For each experiment, we recorded the lowest size of the shared stash, S, that is needed to avoid a failure. In Figure 3, we show the fraction of trials (out of 2000) that result in a stash overflow. We have that overflows happen more frequently with $\epsilon = 0.1$ than with $\epsilon = 0.2$, as one would expect since smaller tables lead to more collisions. For both values of ϵ we found that a stash of size less than $\log n$ was enough to avoid overflows completely. For the implementation above, we pick $\epsilon = 0.1$ and $s = 2$ since the rebuild of larger tables, i.e. with $\epsilon > 0.1$, is more expensive than reading a stash of size $2 \log n$ on every request.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

[2] M. Ajtai. Oblivious RAMs without cryptographic assumptions. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 181–190. ACM, 2010.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 1–9, 1983.

[4] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

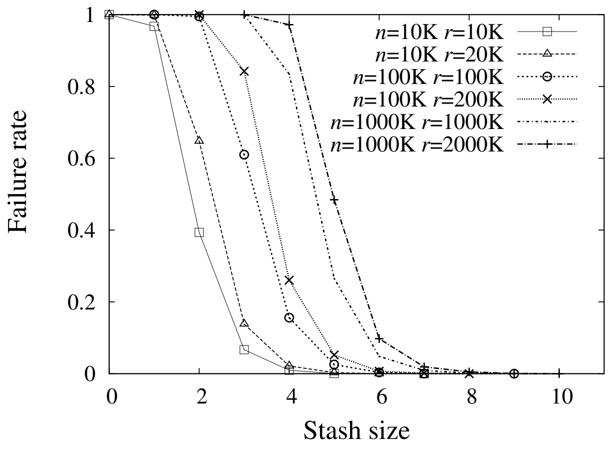
n	$\log n$	Avg request overhead	
		$p = \sqrt{n}$	$p = 2\sqrt{n}$
10,000	14	42	29
100,000	17	31	25
1,000,000	20	28	25
10,000,000	24	28	27

(a)

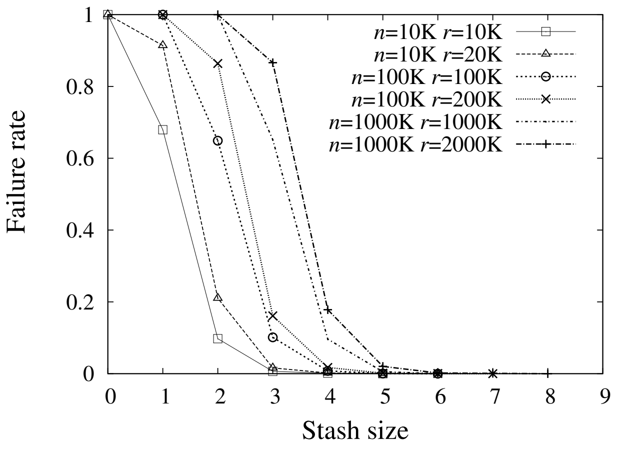
n	Avg request overhead	
	[32]	Our Result
2^{16}	18.4	22
2^{18}	19.9	23
2^{20}	21.5	24
2^{22}	23.2	26

(b)

Table 2: Average number of accesses made by our *stateless* ORAM simulator with n data items and private workspace of size p on a random request sequence that ends as soon as last table T_L is rebuilt. Each data item is of size $b = 10$KB. (a) Influence of the size of the private workspace on the performance. (b) Comparison with results from a *stateful* RAM simulation in [32] using $p = 4\sqrt{n}$.



(a) $\epsilon = 0.1$



(b) $\epsilon = 0.2$

Figure 3: Failure rate in 2000 trials for n items and r requests with (a) $\epsilon = 0.1$ and (b) $\epsilon = 0.2$. The variable K in the charts denotes 1000.

[5] P. Beame and W. Machmouchi. Making RAMs oblivious requires superlogarithmic overhead. Electronic Colloquium on Computational Complexity, Report TR10-104, 2010. `http://eccc.hpi-web.de/report/2010/104/`.

[6] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 62–73, New York, NY, USA, 1993. ACM.

[7] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Technical report, 2011. `http://dspace.mit.edu/handle/1721.1/62006`.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 2nd edition, 2001.

[9] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *8th Theory of Cryptography Conference (TCC)*, pages 144–163, 2011.

[10] Y. Dodis and P. Puniya. Getting the best out of existing hash functions; or what if we are stuck with SHA? In *Applied Cryptography and Network Security (ACNS)*, pages 156–173, 2008.

[11] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 218–229, New York, NY, USA, 1987. ACM.

[12] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[13] M. T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–16. SIAM, 2010.

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

[15] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. *CoRR*, abs/1107.5093, 2011. To appear in Proc. ACM Cloud Computing Security Workshop (CCSW) 2011.

[16] M. T. Goodrich and R. Tamassia. *Algorithm Design:*

Foundations, Analysis, and Internet Examples. John Wiley & Sons, New York, NY, 2002.

[17] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: cuckoo hashing with a stash. *SIAM J. Comput.*, 39:1543–1561, 2009.

[18] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, Inc., Boston, MA, USA, 2005.

[19] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[20] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. Cryptology ePrint Archive, Report 2011/327, 2011. http://eprint.iacr.org/. To appear in SODA 2012.

[21] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan-Kaufmann, San Mateo, CA, 1992.

[22] T. Leighton and C. G. Plaxton. Hypercubic sorting networks. *SIAM J. Comput.*, 27(1):1–47, 1998.

[23] M. Mitzenmacher. Some open questions related to cuckoo hashing. In *Proc. European Symposium on Algorithms (ESA)*, pages 1–10, 2009.

[24] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[25] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 52:122–144, 2004.

[26] M. Paterson. Improved sorting networks with $O(\log N)$ depth. *Algorithmica*, 5(1):75–92, 1990.

[27] B. Pinkas and T. Reinman. Oblivious RAM revisited. In T. Rabin, editor, *Advances in Cryptology (CRYPTO)*, volume 6223 of *Lecture Notes in Computer Science*, pages 502–519. Springer, 2010.

[28] N. Pippenger and M. J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[29] V. R. Pratt. *Shellsort and sorting networks*. PhD thesis, Stanford University, Stanford, CA, USA, 1972.

[30] J. Seiferas. Sorting networks of logarithmic depth, further simplified. *Algorithmica*, 53(3):374–384, 2009.

[31] E. Shi, H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. Cryptology ePrint Archive, Report 2011/407, 2011. http://eprint.iacr.org/. To appear in AsiaCrypt 2011.

[32] E. Stefanov, E. Shi, and D. Song. Towards Practical Oblivious RAM. *CoRR*, abs/1106.3652, June 2011.

[33] G. Wang, T. Luo, M. T. Goodrich, W. Du, and Z. Zhu. Bureaucratic protocols for secure two-party sorting, selection, and permuting. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 226–237, New York, NY, USA, 2010. ACM.

[34] P. Williams and R. Sion. Usable PIR. In *NDSS*. The Internet Society, 2008.

[35] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 139–148, New York, NY, USA, 2008. ACM.