Oblivious RAM Revisited

Benny Pinkas[1,⋆] and Tzachy Reinman[2]

[1] Dept. of Computer Science, University of Haifa, Mount Carmel, Haifa 31905, Israel
benny@pinkas.net
[2] School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem 91904, Israel
reinman@cs.huji.ac.il

Abstract. We reinvestigate the oblivious RAM concept introduced by Goldreich and Ostrovsky, which enables a client, that can store locally only a constant amount of data, to store remotely n data items, and access them while hiding the identities of the items which are being accessed. Oblivious RAM is often cited as a powerful tool, but is also commonly considered to be impractical due to its overhead, which is asymptotically efficient but is quite high. We redesign the oblivious RAM protocol using modern tools, namely Cuckoo hashing and a new oblivious sorting algorithm. The resulting protocol uses only $O(n)$ external memory, and replaces each data request by only $O(\log^2 n)$ requests.

Keywords: Secure two-party computation, oblivious RAM.

1 Introduction

The need to enhance the security of data storage systems and to encrypt the content they store is obvious. Various encryption algorithms are in common use for many years, so content-encryption may be considered, for the most part, as an already-solved issue. Apparently, encryption alone does not suffice. A server, which maintains a data storage system, can gain information about its users' habits and interests, and violate their privacy, even without being able to decrypt the data that they store. The server can monitor the queries made by the clients and perform different traffic analysis tasks. It can learn the usual pattern of accessing the encrypted data, and try to relate it to other information it might have about the clients. For example, if a sequence of queries $q1, q2, q3$ is always followed by a stock-exchange action, a curious server can learn about the content of these queries, even though they are encrypted, and predict the user action when the same (or similar) sequence of queries appears again. Moreover, it is possible to analyze the importance of different areas in the database, e.g., by counting the frequency of the client accessing the same data items. If the server is an adversary with significant but limited power, it can concentrate its resources in trying to decrypt only data

⋆ This research was supported by the European Research Council as part of the ERC project SFEROT, and by the Israel Science Foundation (grant No. 860/06).

T. Rabin (Ed.): CRYPTO 2010, LNCS 6223, pp. 502–519, 2010.

items which are often accessed by the target-user. Another ability of the server is to draw conclusions about relations between queries, and so on.

In order to protect against this kind of privacy violation, one must hide the access patterns of clients of the storage system. This problem is related to the classic result of Pippenger and Fischer on oblivious simulation of Turing machines [18]. In the context of RAM machines, this problem was investigated by Goldreich [8] and Ostrovsky [14] as a software protection problem (the goal there was to hide the pattern of access of a program to memory in order to prevent reverse engineering of the software). The best results of Goldreich and Ostrovsky appear in [9].

Hiding the access pattern, or making it *oblivious*, means that any equal-length sequence of clients' data requests to the server are equivalent from the point of view of the eavesdropper (who might be the server itself). The server must only know the number of queries in the sequence.

The cost of the best protocol of Goldreich-Ostrovsky was efficient asymptotically but clearly unfeasible for any reasonable application: Storing n data items was replaced with storing $O(n \log n)$ items; furthermore, each access to a data item was replaced by $O(\log^3 n)$ data requests to the stored data (this $O(\log^3 n)$ overhead comes with a very large constant factor; it can be replaced with $O(\log^4 n)$ with a reasonable constant).

Due to the overwhelming overhead of the oblivious RAM protocol, it was often cited as a "theoretical" solution which could in principal solve many problems (such as cache attacks, or search on encrypted data; see discussion below), but is clearly impractical. Our goal was to design an improved protocol which will be feasible in practice. We describe in this work a new construction with a considerably improved overhead: it requires the client to store only $O(n)$ items, and replace each data request with $O(\log^2 n)$ accesses to the stored data, where the constants in the "O" notation are small. A detailed comparison with previous schemes appears in Sect. 2.

Other applications of oblivious RAM. We mentioned above that oblivious RAM can be used to hide access patterns to data stored on a remote and untrusted server, or to enable a CPU to operate securely with an untrusted memory. Another application of oblivious RAM is for the symmetric encryption variant of "search on encrypted data", where a client stores data (e.g. mail messages) remotely, and wishes to use the data while protecting its privacy (see, e.g. [19]). Oblivious RAM can also be used for protecting against cache attacks, which are software side-channel attacks run by monitoring the state of the CPU's memory cache. These attacks have been demonstrated to reveal AES keys in real systems [15]. As noted in [15], an oblivious RAM can hide these access patterns, but at a cost which is definitely unacceptable for basic CPU operations.

The basic ideas behind our new construction. We base our solution on the Goldreich-Ostrovsky hierarchical solution, which is described in [9] (and in the full version of our paper). We improve its overhead by using the following primitives instead of the original components of the construction.

– *Cuckoo Hashing.* In the Goldreich-Ostrovsky construction the client maps data items into bins using a random hash function that is kept secret from

the server. The number of items mapped into each bin must be hidden from the server. It is well known that when n items are randomly mapped to n bins then (with high probability) the most populated bin contains $O(\log n)$ items. Therefore in the original construction the client sets each bin to have sufficient room for $O(\log n)$ items, and stores in a bin fake items if less than this number of items are mapped to it. This increases the overall storage required by the construction to $O(n \log n)$.
In comparison, our construction uses Cuckoo hashing [16,17], which is a hashing scheme mapping n items to $2(1+\epsilon)n$ bins with the guarantee that at most a single item is mapped into a bin. Consequently, the construction uses a total of only $O(n)$ server storage.

- *Pseudo-random permutation.* The server observes where items are inserted to the Cuckoo hash table, and might use this information to identify "dummy" items (a discussion of the usage of dummy items is given in Sect. 4). In order to prevent that, the client needs to apply a pseudo-random permutation to the order of the items before inserting them to the hash table.
- *Randomized Shell sort.* The storage system is built of hierarchical levels. Periodically, the items of two adjacent levels are reshuffled. The reshuffling process uses sorting, which is composed of many steps where the client retrieves a pair of encrypted items from the server, decrypts them and compares the results, and stores a re-encrypted version of the sorted pair. The sorting must be oblivious in the sense that the indices of the pair of items that are compared must not leak any information about the results of previous comparisons. The original Goldreich-Ostrovsky construction uses a sorting network for this purpose, but this solution has an overhead of $O(n \log^2 n)$ comparisons, with a very small constant, using Batcher's network [5], or $O(n \log n)$ comparisons, with a constant of about 6100, using the AKS network [2]. We perform sorting using the new randomized Shell sort algorithm of Goodrich [10]. This algorithm is oblivious, sorts with very high probability, and works in $O(n \log n)$ comparisons; where the "O" notation hides only a very small constant.

We stress that even given these improved primitive building blocks, a lot of care had to be taken in order to compose them to a secure, and efficient, oblivious RAM protocol. Additional effort was needed in order to reduce the constant factors of the overhead.

1.1 Basics of Oblivious RAMs

The problem of hiding access patterns is modeled in the following way: The setting includes a client which has a small secure memory, and a server with a large insecure storage. The client can use the server's storage to store and retrieve its data. The client stores internally a secret key of a symmetric encryption scheme, and uses it to encrypt the data before storing it, and decrypt it after retrieving it.

We assume here and throughout the paper that encryption is done with a *semantically secure* probabilistic encryption scheme and therefore two encrypted

copies of the same data look different. The server cannot identify whether these two copies correspond to the same data of the client.

The client has n data items denoted as (v_i, x_i), where $i = 1, \ldots, n$ is an index, v_i is the data identifier or location-index (e.g., a serial number), different for each data item, and x_i is the data payload. It is assumed that all x_i values are of the same length. To simplify the description we assume that the storage service of the server has slots of a size which is equal to the size of an encryption of a data item used by the client. Therefore each slot can be used to store a data item, where the client can ask to store a specific data item in a specific slot location j. All requests to the server are therefore of the form "GET j", which provides the client with the (encrypted) content of slot j, or "PUT `data` j", which stores at slot j the encrypted data provided by the client.

The client has a small amount of secure internal memory. It includes space for $O(1)$ data items, for $O(1)$ secret keys for symmetric key cryptographic functions, and for a constant number of counters which count up to n and therefore are of length $O(\log n)$ bits.

We assume that the server does not tamper and modify the stored data, because this issue can be easily solved by the client authenticating the stored data using a message authentication code (MAC) and a secret key known only to the client. However, the server does learn which location in its storage is being accessed by the client in each operation.

By default, the client cannot hide the fact that it accesses a specific location in the server's storage. The server can examine the contents of its storage and of the requests from the client, but the server obviously cannot learn the contents of the stored data, since it is encrypted. The goal of the client is to hide its access pattern to the stored data. This is expressed in the following definition.

Definition 1. *The input y of the client is a sequence of data items, denoted by $y = ((v_1, x_1), \ldots, (v_n, x_n))$ and a corresponding sequence of operations, denoted by $(op_1, \ldots, op_m)$, where each operation is either a read operation, denoted* $\mathrm{read}(v)$*, which retrieves the data of the item indexed by v, or a write operation, denoted* $\mathrm{write}(v, x)$*, which sets the value of item v to be equal to x.*

The access pattern $\mathcal{A}(y)$ *is the sequence of accesses to the remote storage system, which contain both the indices accessed in the system and the data items read or written. An oblivious RAM system is considered secure if for any two inputs y, y' of the client, of equal length, the access patterns $\mathcal{A}(y)$ and $\mathcal{A}(y')$ are computationally indistinguishable for anyone but the client.*

Hiding the access patterns, or "unifying" them, must have a cost – each access is simulated by more than one access. First, we would like to make the different types of accesses look the same. For example, if we want that *read* and *write* would be indistinguishable, we would have each of them both implement *read* and *write*, i.e., read the value in the accessed location, decrypt it and then rewrite it with an encryption of the same value or a different one. Note that since we use a semantically secure probabilistic encryption, the server cannot identify whether the data was changed before it was written back. We note that this element of making different types of data access look the same, by always using

a read-and-then-write operation, is common for all the following solutions. From here on, we treat all *read*, *write*, or other access-operation, as equal. Adding a write operation to each read operation already multiplies the computational overhead by a factor of two. In addition, we would like to prevent the adversary from distinguishing between accesses to locations $\{v_1, v_2, v_3\}$ and $\{v_2, v_1, v_2\}$, etc. A trivial solution is to read and rewrite the entire set of stored data for each access. Applying this solution is usually infeasible, since it multiplies the computational overhead by a factor equal to the number of stored items, which is normally huge. On the other hand, it is easy to see that this is the best possible deterministic scheme. A probabilistic scheme, where the operation of the client depends on a random bits, can do much better.

2 Related Work

Most oblivious RAM constructions are based on the client having access to a secret (pseudo-)random function, which is implemented using symmetric cryptographic functionalities, such as encryption, and can therefore be constructed assuming the existence of one-way functions. Very recent results of Ajtai [1] and of Damgård et al. [7] construct an oblivious RAM based on no cryptographic assumption (but rather, letting the client use the oblivious RAM itself for storing random coin tosses and accessing them obliviously). The client needs to store remotely (for each of its data items) an equivalent to a poly-logarithmic amount of items, rather than $O(1)$ items in our scheme, and each data request is replaced with a poly-logarithmic number of requests to the server. It is not clear how high is the exponent of this poly-logarithmic overhead. We therefore focus our description of related work on cryptographic oblivious RAM schemes.

The investigation of oblivious RAM techniques was initiated by Goldreich and Ostrovsky [9]. A major tool used in their constructions is a primitive which performs an oblivious sorting of the stored data. That is, it sorts the stored data items according to some index, while hiding from the server all information about the permutation that orders the input set of data items. Specifically, this primitive was implemented in [9] using a sorting network: either the sorting network of Batcher [5] which performs $O(n \log^2 n)$ read operations with a very small constant (approximately 1/2), or the sorting network of AKS [2] which performs only $O(n \log n)$ read operations, but whose complexity has a much larger constant. (The actual overhead of the AKS sorting network is about $6100 n \log n$ comparisons, and therefore it is clear that for any feasible input, the performance of the Batcher network is preferable.)

Goldreich and Ostrovsky presented a basic "square-root" algorithm, whose overhead is $O(\sqrt{n})$ read/write operations for each original access to a data item. They also designed a more complex hierarchical solution, using a data structure composed of levels, where each level is twice the size of the former level, and whose overhead is $O(\log^4 n)$ (using Batcher sorting network). A detailed description of these solutions can be found in [9] or in the full version of our paper.

Williams and Sion [20] modified the hierarchical solution of Goldreich and Ostrovsky, assuming that the client can locally store $O(\sqrt{n})$ data items, rather than $O(1)$ items. This extended local storage enables to run an oblivious merge sort and improve the run time overhead to $O(\log^2 n)$. A solution based on Bloom filters [6] was presented in [21]. In that solution the client stores at the server, for every level, an encrypted Bloom filter and uses it to check whether an item appears in the level. The work in [21] claims to reduce the storage overhead at the server to $O(n)$, and to reduce the number of actual data requests per item requested by the client to only $O(\log n \log\log n)$. That analysis is based on the assumption that the size of the Bloom filter encoding m items is $O(m)$. The overhead is actually larger, since the size of the Bloom filter must also be a function of the number of the hash functions used and of the allowed error probability (which in inevitable when a Bloom filter is used). As a result, the overhead of the Bloom filter based scheme is worse than that of our scheme for any reasonable choice of the number of items n and of the error probability of the filter.

Table 1. A comparison of the different access hiding schemes. (For the scheme of [21], we note that the original analysis is inaccurate. The second line is for an invocation using an optimal number of hash functions, with specific numbers for an error probability of 2^{-64}.)

	computational overhead	client memory (data items)	server storage (data items)
Goldreich-Ostrovsky [9] $\sqrt{n}$	$O(\sqrt{n}\log n)$	$O(1)$	$O(n+\sqrt{n})$
Goldreich-Ostrovsky [9] Batcher	$O(\log^4 n)$	$O(1)$	$O(n\log n)$
Goldreich-Ostrovsky [9] AKS	$O(\log^3 n)$, const ≥ 6100	$O(1)$	$O(n\log n)$
Merge sort [20]	$O(\log^2 n)$	$O(\sqrt{n})$	$O(n\log n)$
Bloom filter [21]			
original analysis (inaccurate)	$O(\log n\log\log n)$	$O(\sqrt{n})$	$O(n)$
optimal # of hash functions	$O(1.44c\log n\log\log n)$ for $c=64$: const > 92	$O(\sqrt{n})$	$O(n)$ $+1.44cn$ bits
This paper	$O(\log^2 n)$	$O(1)$	$O(n)$

Table 1 compares the performance of all schemes described in this section.[1] The performance comparison can be summarized as follows: (1) The constructions of

[1] Note that for the Bloom filter based scheme [21] the first line of the table lists the performance according to the original analysis in [21], which is inaccurate. The second line lists the performance according to a more careful analysis (detailed in the full version of our paper), assuming an allowed error probability of 2^{-c}. The $O(\log n\log\log n)$ overhead in the second line has a constant factor of at least $1.44c$ (greater than 92 for $c = 64$), *in addition* to other constant factors which are similar to those incurred by all schemes. Given this finer analysis, the performance of [21] is worse than the performance of our scheme when $\log n < 1.44c\log\log n$, which is clearly the case for any reasonable choices of n and c. For example, for $n < 2^{80}$ this holds for any error parameter $c \geq 9$.

Goldreich-Ostrovsky and of our work are the only ones using local storage of only $O(1)$ data items; (2) The computational overhead of our construction is better or equal to that of all other constructions (except for the asymptotic overhead of the Bloom filter construction for unreasonably high values of n); (3) The amount of server storage in our construction is better than that of all other constructions (except for the Bloom filer construction, which stores a comparable number of data items and in addition $1.44cn$ bits, which are more than $92n$ bits for $c = 64$).

3 Building Blocks

3.1 Randomized Shell Sort (Oblivious Sorting Algorithm)

Goodrich's recent randomized Shell sort algorithm [10] is an efficient sorting algorithm, using only $O(n \log n)$ comparisons with a relatively small constant factor. Equally important is the fact that this algorithm is also data oblivious. This property means that if we assume that the operation of comparing two items and reordering them according to their value is a black-box (i.e., the result of the comparison is hidden from an external observer, which is the server in our case), then the algorithm performs no operations which depend on the relative order of the elements in the input array. In other words, an external observer who can only observe the items which the algorithm compares, but not the results of the comparisons, sees a list of pairs of items which are compared, where the choice of items for these pairs is independent of the results of previous comparisons.

We note that other sorting algorithms are not known to be both oblivious and efficient. For example, bubble sort is oblivious, but is not efficient; quick sort is efficient (in the average case) but is not oblivious; sorting networks are oblivious, but, as noted in Sect. 1, the only sorting network constructions of size $O(n \log n)$ are not efficient in the practical sense, due to large constants. See [2,10] for details.

We use randomized Shell sort in our scheme in order to reorder items in the server database, according to a new permutation, in a way that prevents the server from tracking the new ordering. The details of the randomized Shell sort construction appear in [10] or in the full version of our paper.

3.2 Cuckoo Hashing

Cuckoo hashing [16,17] is a relatively new hashing algorithm, which in its basic form maps each item to one of two potential entries of a hash table, while ensuring constant lookup and deletion time in the *worst case*, and amortized constant time for insertions.

The basic idea of Cuckoo hashing is to use two hash functions denoted h_0 and h_1 (or multiple hash functions in the general case). The size of a hash table used for storing n items must be slightly larger than $2n$ (to simplify the discussion, we consider the size of the table to be exactly $2n$). When a new item x is inserted to the hash table, it is inserted to location $h_0(x)$. If this location is already occupied

by another item y, then that item is "kicked out" of its current location and is re-located to its other possible location. Namely, if $h_b(y) = h_b(x)$ (for $b \in \{0,1\}$, and initially $b = 0$) then item y is moved to location $h_{1-b}(y)$. If location $h_{1-b}(y)$ is already occupied by another item z (i.e., $h_{1-b}(z) = h_{1-b}(y)$), this item (z) is re-located to location $h_b(z)$, and so on. If this chain of relocations continues for too long, then the table is rehashed using two new hash functions h'_0, h'_1. In this case the insertion time is longer, but analysis shows that this event is rare, and therefore the amortized insertion time is constant. Lookup and deletion are natural – one just has to check the two possible locations of the given item.

Most recent works (e.g., [11,12,13,3,4]) present variants of Cuckoo hashing with guaranteed constant worst-case performance for insertion (this is also referred to as de-amortizing the insertion time of Cuckoo hashing).

4 Our Scheme

We first describe the basic form of our oblivious RAM scheme, which has the desired asymptotic overhead. Appendix A then describes how to improve the constant factors of the overhead of the scheme.

The construction is based on combining a modified version of the hierarchical solution of Goldreich and Ostrovsky with Cuckoo hashing and randomized Shell sort. The server stores the data, which can potentially consist of n items, in a hierarchical data structure of $N = \lceil \log_2 n \rceil + 1$ levels, each of which is twice larger than its previous. Additional levels may be allocated, as necessary (when a new level is allocated, its size is twice the size of the last allocated level).

In the original scheme of Goldreich-Ostrovsky, level i consists of 2^i buckets, where each bucket contains $O(\log n)$ entries. In our scheme level i consists of a table of $4 \cdot 2^i$ single item entries, which will be used to store up to 2^i data items of the client. Storing the items is done in the following way: Along with the 2^i items of the client, up to 2^i "dummy" items might be stored in the level, where the client might access a dummy item in order to hide the fact that it does not need to search for a real item in this level (since the real item was already found in a previous level). All $2 \cdot 2^i$ items of the level are stored in a Cuckoo hashing table of size $4 \cdot 2^i$. (We note that according to this description the first level is used to store only two items. Any actual implementation would probably set the first level to be much larger, say, to contain 128 data items. To simplify the analysis we assume, however, that the first level stores only two items.)

For each level we associate an *epoch*, which is defined for level i as 2^{i-1} requests (the epoch ends when a reshuffle from level $i-1$ to level i occurs). For each level i and its ℓ^{th} epoch, the client randomly chooses two hash functions whose ranges are $\{1 \ldots 2^{i+2}\}$: $h^{i,\ell}_{k,0}$ and $h^{i,\ell}_{k,1}$, where k is a secret key known to the client and used to define these functions. At the end of every epoch each level is re-hashed obliviously, using a new pair of hash functions. The following table summarizes the properties of level i.

	real items	dummy items	size	epoch-length	"moved down"
level i	2^i	2^i	$4 \cdot 2^i$	2^{i-1}	every 2^i requests

Each data request includes both reading and writing to the data structure, such that the server cannot distinguish which operation occurred. In addition, for any request, the accessed item is re-encrypted by the client, using a probabilistic encryption scheme.

Data requests. Initially, the data structure is empty. For each request (of any type) of a location-index (virtual address) v, the following operations are performed.[2]

1. Scan through the entire first level in a sequential order to find the item whose location-index is v. This step includes reading all the items in the first level. If the requested item is found, it is stored in the client's secure memory, and the process continues as usual.
2. Go through all other levels, and for each level $i = 2 \ldots N$, do:
   - If v has not been found yet, examine its two possible locations in the Cuckoo hashing table of the current level (i): $h^{i,\ell}_{k,0}(v)$ and $h^{i,\ell}_{k,1}(v)$. If the requested item is found in one of the two locations, it is stored in the client's secure memory, and the process continues as usual.
   - If v has been found, examine two random locations $h^{i,\ell}_{k,0}(\text{"}dummy'' \circ t)$ and $h^{i,\ell}_{k,1}(\text{"}dummy'' \circ t)$, where t is a counter which is increased with every data request. (These are locations allocated by the Cuckoo hashing for two fresh dummy items which were not searched for before.)
3. Scan again through the entire first level in a sequential order, and write back the updated (and re-encrypted) item of location-index v in the next available location. If v is already in the first level, overwrite it. This step includes reading and writing all items of the first level.

The first level functions as a cache, meaning that for any request, the updated value is written to the first level. Since the capacity of the level is final, after a certain number of requests it becomes full. In order to avoid this, the content of the first level is "moved down" to the second level before the first level becomes potentially exhausted. Now the second level may become full, so the same process is repeated. When the content of the last level has to be "moved down", a new level with twice the number of entries is allocated.[3] "Moving down" the content of level i is done every 2^i requests. This makes sure that no level is overflowed, and that the first level is emptied and has enough available slots at the beginning of each epoch of any of the levels (since the beginning of an epoch of level i is

[2] We assume here, as was implicitly assumed by all previous constructions [9,20,21], that the client does not perform a "read" operation for an item which does not exist in the remote storage.

[3] In fact, if we are willing to disclose an upper bound on the number of items that are stored, there is no need to allocate a new level when the last level has to be "moved down". The system may instead re-order the entire database.

also a beginning of an epoch of all the levels $j < i$). In fact, this process makes sure that at any time level i contains no more than 2^i items, as is stated in Lemma 1 below. Whenever level i is moved down to level $i+1$, the latter level is reshuffled.

When the client moves the content of level i to level $i+1$, it obliviously hashes the content of both levels to level $i+1$. This reshuffling must fulfill the following requirements: (1) If there is a duplicate item (the same location-index, and possibly different data content) in level i and level $i+1$, the newer item (from level i) must be kept, and the older one must be deleted; (2) The resulting buffer, namely level $i+1$ after the reshuffling, must be ordered independently of any of the levels before the reshuffling; (3) Level i must be cleaned, i.e., its content must be deleted. As we continue, we see that all these requirements are fulfilled.

Before describing the reshuffle process, we state Fact 1, which trivially follows from the reshuffling algorithm, and Lemma 1, which is proved in the full version of the paper.

Fact 1. *When a reshuffle from level i to level $i+1$ occurs, all levels $j \leq i-1$ are empty. At the end of the reshuffle, all levels $j \leq i$ are empty.*

Lemma 1. *When a reshuffle from level i to level $i+1$ occurs, each of these two levels contains at most 2^i real items. At the end of the reshuffle, level i is empty and level $i+1$ has at most 2^{i+1} real items.*

4.1 Reshuffling Levels Using Cuckoo Hashing and Randomized Shell Sort

The reshuffle of levels i and $i+1$ into level $i+1$ is a complex process, consisting of the steps enumerated below and based on two basic primitives: (1) *Scanning*, which is reading and (possibly) writing in a sequential order all the items in a given buffer; (2) *Oblivious Sorting* (O-Sort), which is done by randomized Shell sort (see Sect. 3.1). Note that whenever an item is written to a storage (whether it is one of the levels or a temporary buffer), it is re-encrypted. The reshuffle process is also depicted in Fig. 1.

1. Allocate a temporary buffer C, whose size is 2^{i+1}. (Recall that jointly, both levels contain at most 2^{i+1} real items).
2. O-sort each of the levels (level i and level $i+1$): The sorting is according to an order which locates real items before dummy and empty items. At the end of this step, all the real items of level i (at most 2^i) are in its first locations, and all the real items of level $i+1$ (at most 2^i) are in its first locations.

   In the following two steps (3–4), 2^{i+1} items that include all the real items of the two levels are copied into the temporary buffer.
3. Move the first 2^i items from level i to the left side of C. Mark each item as "new". At the end of this step, *all* the real items of level i (and possibly additional items) are in C. (This step is depicted in Step I of Fig. 1.)

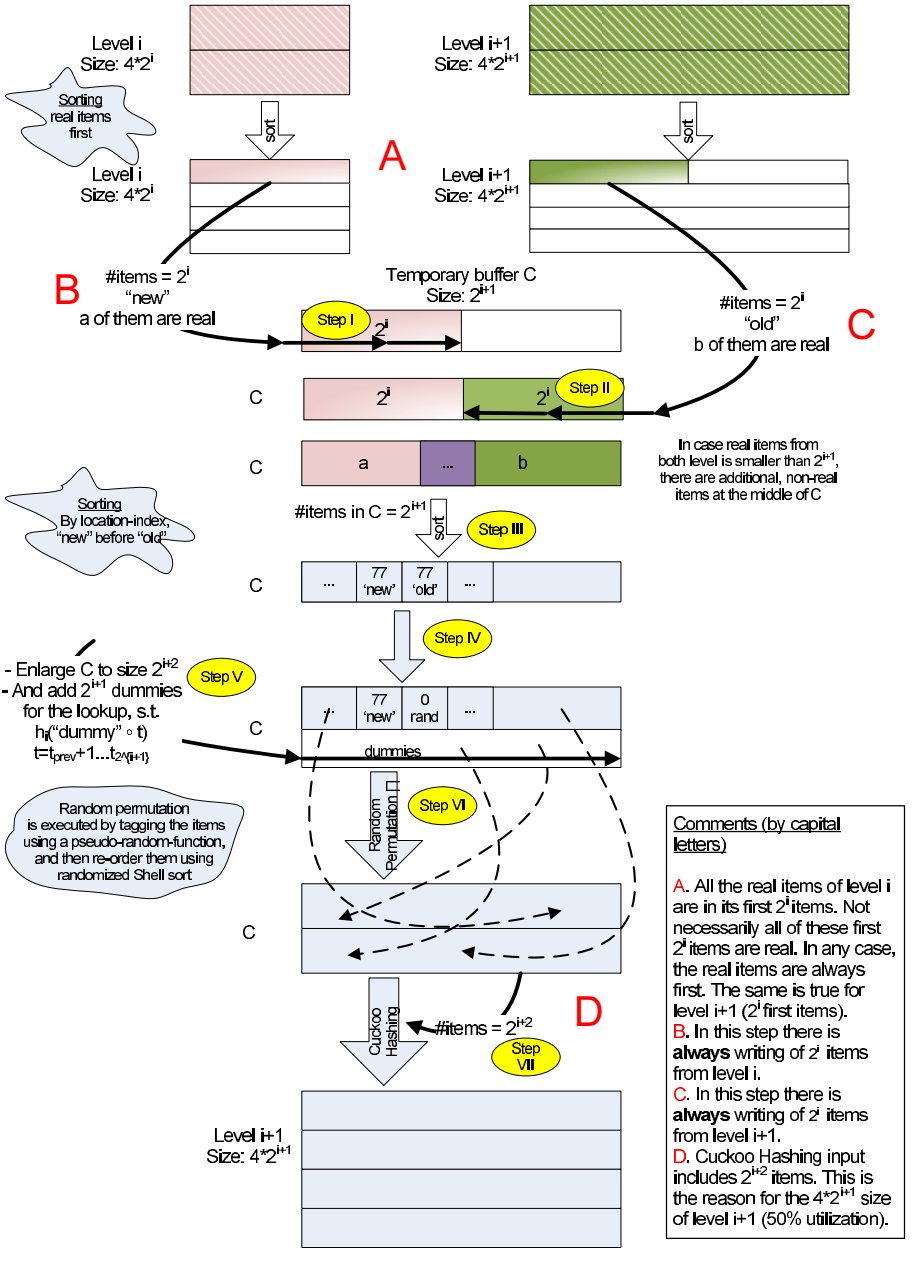

Fig. 1. Reshuffle steps

4. Move the first 2^i items from level $i+1$ to the right side of C. Mark each item as "old". At the end of this step there are 2^{i+1} items in C, that include *all* the real items of both levels and possibly additional items (if both levels together contained less than 2^{i+1} real items). (This step is depicted in Step II of Fig. 1.)

   The goal of the following two steps (5–6) is to erase duplications of items with the same location-indices.
5. O-sort C according to these criterions (ordered): (a) smaller location-indices (virtual addresses) first, (b) items tagged "new" before items tagged "old". (This step is depicted in Step III of Fig. 1).
6. Removing duplicates: Sequentially scan C and erase each old item preceded by a new item with the same location-index (the marks "old" and "new" of the remaining items can be ignored from now on). Replace each erased item and each dummy item with a random item (an item with a special location-index and random content). At the end of this step there are 2^{i+1} real and random items in C, without duplications. We will refer to all these items in the sequel as "real". (This step is depicted in Step IV of Fig. 1.)
7. Create 2^{i+1} dummy items with indices "dummy" $\circ(t+j)$, where t is a counter of the number of requests so far and $j = 1 \ldots 2^{i+1}$.[4] Add these items to C (this requires increasing the size of C from 2^{i+1} to 2^{i+2}). (This step is depicted in Step V of Fig. 1.)
   The client then obliviously reorders the items in a pseudo-random order. This is done by (1) choosing a new keyed pseudo-random function F_k and using it to tag each of the 2^{i+2} items with a new value which is the result of $F_k()$ applied to their location-index, and then (2) obliviously sorting the items by their tags using randomized Shell sort. The new order of the items is independent of their original order. (This step is depicted in Step VI of Fig. 1.)
8. Sequentially scan the buffer and use Cuckoo hashing with two new random hash functions that are kept secret from the server: $h_{0,k}^{i+1,\ell+1}$ and $h_{1,k}^{i+1,\ell+1}$, to map the 2^{i+2} items to the $4 \cdot 2^{i+1}$ entries of level $i+1$ (ℓ is the index of the current epoch). The hash functions are applied to the location-index. At the end of this step there are 2^{i+2} real and dummy items in level $i+1$, located according to the Cuckoo hashing functions. (This step is depicted in Step VII of Fig. 1.)
9. If the Cuckoo hashing fails (due to cycles, see [17]), choose new random secret hash functions $h_{0,k}^{i+1,\ell+1}$ and $h_{1,k}^{i+1,\ell+1}$ and repeat the previous step.

After step 4, all the real content of levels i and $i+1$ is in C (possibly with additional items), Steps 5 and 6 handle possible duplications (of items with the

[4] These dummies are necessary in order to hide whether a data request in level $i+1$ in the next epoch searches for an item which was found in a level prior to level $i+1$. If this event happens in the jth time slot of the epoch, then the client will look for item "$dummy'' \circ (t+j)$, which was inserted to the level in the reshuffle. As a result, every search in this level will be to an item which is stored in the Cuckoo hashing table and which was never searched before.

same location-index in the two original buffers), Step 7 reorders the real and dummy items (pseudo-) randomly, and Steps 8 and 9 insert the items to level $i + 1$, according to the random secret Cuckoo hashing functions.

4.2 Analysis and Implementation

Overhead. The construction uses $\log n$ levels, where level i contains $4 \cdot 2^i = O(2^i)$ items, yielding a server storage of $O(n)$ data items. The overall amortized computational overhead is $O(\log^2 n)$ data requests for each original request of the client: First, observe that accessing an item requires scanning through the first level (which is of constant size), and then accessing two locations in each other level. The reshuffling process uses randomized Shell sort that sorts ℓ elements in $O(\ell \log \ell)$ time, with a reasonable constant factor. Performing the oblivious sorting is the main time-consuming element of the reshuffle process. The size of the sorted array in level i is $O(2^i)$, giving a sorting time of $O(2^i \log 2^i) = O(2^i \cdot i)$ for level i. Level i is sorted every 2^i requests, giving an amortized cost of $\frac{O(2^i \cdot i)}{2^i} = O(i)$. Summing this for all the levels gives $\sum_{i=1}^{\log n} O(i) = O(\log^2 n)$.

Examining the performance more carefully, we note that level i has room for $4 \cdot 2^i$ items (Appendix A shows how to reduce the storage by about 50%). The bulk of the computation overhead comes from the sorting operations. In particular, Step 2 sorts level $i + 1$, which consists of $4 \cdot 2^{i+1}$ items (the other sorting operations are applied to smaller sets of items). However, we show in Appendix A how the sort operations in Step 2 can be changed to sort half as many items. This is estimated to reduce the overhead by 33%. Appendix A discusses an additional optimization which reduces the constant factors of the overhead of the construction by an additional 33%.

Security

Theorem 1. *The oblivious RAM protocol described above is secure according to Definition 1.*

Proof. The security of the construction holds under the assumption of the existence of pseudo-random functions, or assuming that the client has access to random functions (e.g., an internal random number generator which always provides the same output when given the same input). The PRF assumption is probably more reasonable for most applications. A crucial ingredient of the protocol is that the hash functions h_0 and h_1, used to map items during the Cuckoo hashing, are randomly chosen by the client and are kept secret from the server. (When a PRF is used, these functions are defined by some function $F_k()$ where the key k is chosen by the client and is unknown to the server.)

We will show that for any input sequence y of the client, the access pattern $A(y)$ to the storage server is indistinguishable, by a polynomial-time server, from an access pattern A' which can be simulated without any knowledge of y, except for the length of y.

The *contents* of the requests in the access pattern are encrypted with a semantically secure probabilistic encryption scheme, and therefore the server cannot distinguish between the contents of the requests in $A(y)$ and in A'. We therefore only need to show that the locations accessed by the two access patterns in the server's memory are indistinguishable.

Consider the reshuffle operation from level i to level $i+1$. The first steps of the reshuffle perform an oblivious sorting or a serial scan of data items, and are therefore independent of the actual data stored by the client and of the input sequence y. As such they can be easily simulated. Namely, the simulated access pattern A' contains a serial scan in every step of the reshuffle where such a scan in performed (namely, Steps 1, 3, 4 and 6). In addition, whenever the reshuffle performs an oblivious sorting (in Steps 2, 5 and 7), A' performs an oblivious sorting assuming that the values to be sorted are $1, 2, 3, \ldots$.

Let $M = 2^{i+1}$. In Step 7 of the protocol the client obliviously reorders in a pseudo-random order M real values and M dummy values. Step 8 maps these $2M$ values to a Cuckoo hash table of size $4M$, using two hash functions h_0 and h_1 which are chosen at random by the client and are unknown to the server. The client goes over the $2M$ items according to their new order, and attempts to insert each of them to the table according to the Cuckoo hashing algorithm. If x is a certain item in the list, then the client probes locations $h_0(x)$ and $h_1(x)$ in the table and might perform some evictions of items to find a place for x. In this process the server might learn the h_0 and h_1 values of each of the $2M$ items. However, since the server does not know the hash functions used, these values are independent of the actual values of the items. Simulating this process is performed in the following way: Define random functions h_0, h_1, and apply the Cuckoo hashing algorithm to an *arbitrary set* of $2M$ values, say the values $1 \ldots 2M$, using these functions. This process results in exactly the same distribution, as in the real execution, for all the events observed by the server, including the locations probed in the hash table and the occurrences of evictions and cycles (which might cause a repeat of the Cuckoo hashing algorithm as defined by Step 9). Note that our security analysis does not have to analyze the exact probabilities with which evictions and cycles occur, but rather only observe that these probabilities are independent of the data items being hashed.

At the end of the hashing process the server knows, for each of the $2M$ items, the two locations to which this item is mapped by h_0 and h_1, and the exact location in this pair to which this item was eventually mapped. Recall, however, that the $2M$ items were randomly reordered, and that half of them are dummy items. In the epoch that follows, the server can observe which locations are probed in each request of this level. Namely, it might see that the j^{th} request probes locations 10 and 17 to which, say, the first of the $2M$ items is mapped. However, the server does not know whether this is a real or a dummy item. Also, each item hashed into this level is probed at most once during the epoch, since each dummy value is probed at most once (due to the dummy counter being increased), and a real value that is accessed is immediately moved to the top level and is not accessed again in this level during the current epoch. Given these observations, the probes

to the level in this epoch can be simulated in the following way: use the random functions h_0, h_1 that were used in the simulation of the Cuckoo hashing into this level; let $(a_1, \ldots, a_{2M})$ be a random permutation of the numbers $1, \ldots, 2M$; when performing the jth data request from level $i+1$ in the current epoch, probe the locations to which item a_j is mapped by h_0 and h_1.

We have described above how to simulate probes to a specific level during data requests. The entire sequence of probes during data request can therefore be simulated as follows: In Steps 1 and 3 the simulation scans the entire first level. In Step 2 the simulation goes through all levels, starting with the second one, and simulates a pair of probes to each level, as is described in the previous paragraph. □

Implementation. We implemented a basic prototype of our scheme, including the hierarchical data structures, the randomized Shellsort algorithm and the Cuckoo hashing algorithm. This allowed us to simulate the operation of the oblivious RAM construction, and to measure and estimate its performance. We chose Java as an initial platform and compiled using the Sun JDK 1.6.0_16. The testing environment was a standard PC. In our measurements we ignored network delays, and therefore we only provide measurements of the number of operations per data request, rather than of the amount of time each request takes.

We ran experiments on various databases, of sizes between $n = 2^{10}$ and $n = 2^{20}$. For a database of $n = 2^i$ potential items, we ran $k = 2^i - 10$ requests, and counted the number of read/write operations handled by the server. The results appear in Table 2. The constant of the $O(k \log^2 k)$ overhead seems to be about 160. We note that the two improvements described in Appendix A, (minimizing the amount of sorted items – either by not sorting empty items, or by using an advanced Cuckoo hashing algorithm; and reshuffling several levels together)

Table 2. Performance measurements

$\log_2 n$	n	$k = n - 10$ (# of req.)	$k \log^2 k$	#operations	ops per request	const of $O(k \log^2 k)$
10	1024	1014	101113	15445582	15232	152
11	2048	2038	246281	38081523	18685	154
12	4096	4086	588038	91975576	22509	156
13	8192	8182	1382383	218482493	26702	158
14	16384	16374	3208900	511882978	31261	159
15	32768	32758	7370117	1185355399	36185	160
16	65536	65526	16774194	2717439532	41471	162
17	131072	131062	37876427	6175479249	47118	163
18	262144	262134	84930896	13926487414	53127	163
19	524288	524278	189263809	31192732955	59496	164
20	1048576	1048566	419425822	69442426048	66226	165

which have not yet been implemented by us, are estimated to reduce the overhead by about 33% each. Applying both optimizations is likely to reduce the overhead by about 55%, and obtain a constant of about 72 in the "O" notation.

5 Open Questions

The efficiency analysis of our construction, as well as that of all other known constructions of oblivious RAM, is amortized. A data request which is followed by a reshuffle of level i has a larger overhead than a request which requires a reshuffle of a level $j < i$, or one that does not require any reshuffling. A major open goal is, therefore, to reduce the worst case performance of oblivious RAM. Note that the recent result on deamortizing Cuckoo hash [3] does not help here, since it can be applied to the Cuckoo hashing part of the the reshuffling process, but not to the fact that the worst case overhead of reshuffling is high.

Acknowledgements

The authors wish to thank Yuriy Arbitman for informing us of the randomized Shell sort result.

References

1. Ajtai, M.: Oblivious RAMs without cryptographic assumptions. In: STOC 2010 (2010)
2. Ajtai, M., Kolmós, J., Szemerédi, E.: An $O(n \log n)$ sorting network. In: STOC, pp. 1–9 (1983)
3. Arbitman, Y., Naor, M., Segev, G.: De-amortized Cuckoo hashing: Provable worst-case performance and experimental results. In: ICALP (1), pp. 107–118 (2009)
4. Arbitman, Y., Naor, M., Segev, G.: Backyard Cuckoo hashing: Constant worst-case operations with a succinct representation (2010) (manuscript)
5. Batcher, K.: Sorting networks and their applications. In: AFIPS Spring Joint Computing Conference, vol. 32, pp. 307–314 (1968)
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
7. 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/2010/108`
8. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC, pp. 182–194. ACM, New York (1987)
9. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM 43(3), 431–473 (1996)
10. Goodrich, M.T.: Randomized Shellsort: A simple oblivious sorting algorithm. In: Proceedings 21st ACM-SIAM Symposium on Discrete Algorithms, SODA (2010)
11. Kirsch, A., Mitzenmacher, M.: Using a queue to de-amortize Cuckoo hashing in hardware. In: Proceedings of the 45th Annual Allerton Conference on Communication, Control, and Computing, pp. 751–758 (2007)
12. Kirsch, A., Mitzenmacher, M.: Simple summaries for hashing with choices. IEEE/ACM Trans. Netw. 16(1), 218–231 (2008)

13. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: Cuckoo hashing with a stash. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 611–622. Springer, Heidelberg (2008)
14. Ostrovsky, R.: Efficient computation on oblivious RAMs. In: STOC 1990, pp. 514–523. ACM Press, New York (1990)
15. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
16. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 121–133. Springer, Heidelberg (2001)
17. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms 51(2), 122–144 (2004)
18. Pippenger, N., Fischer, M.J.: Relations among complexity measures. J. ACM 26(2), 361–381 (1979)
19. Song, D., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of 2000 IEEE Symposium on Security and Privacy, S&P 2000, pp. 44–55 (2000)
20. Williams, P., Sion, R.: Usable PIR. In: NDSS (2008)
21. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In: ACM Conference on Computer and Communications Security, pp. 139–148 (2008)

A Optimizing the Construction

We present here several optimizations to the basic oblivious RAM construction presented in Sect. 4. The optimizations improve the constant factors of the overhead, but not its asymptotic performance. Still, they are beneficial for any implementation of the construction.

Not storing empty items. Recall that each level i contains up to 2^i real items and 2^i dummy items which must be indistinguishable, from the server's point of view, from the real items. The remaining 2^{i+1} locations in this level are empty, and are needed for the Cuckoo hashing to succeed. The construction can be optimized by not storing in these locations encrypted "empty" data items, but rather using a flag signaling that the entry is empty. Since we can safely assume that a data item is much larger than this flag, this optimization saves about 50% of the storage required by the levels.

As for security, note that this change enables the server to identify empty locations, but it does not enable it to distinguish between real items and dummy items. Namely, this corresponds to revealing to the server the empty locations in a Cuckoo hashing table, but since the hash functions used are kept secret, no information is revealed about the items in the table.

Implication to sorting. Step 2 of the reshuffle algorithm sorts levels i and $i+1$, whose lengths are $4 \cdot 2^i$ and $8 \cdot 2^i$, respectively. These sorting operations are done in order to move the real items to the beginning of these buffers. If empty items are flagged, as suggested above, then there is no need to sort the corresponding entries in the level. Namely, the data to be sorted is half as long as in the basic protocol, and the overhead of sorting is reduced by more than 50%.

Let us therefore estimate how much is saved by this optimization. Note that Steps 5 and 7 sort $2 \cdot 2^i$ and $4 \cdot 2^i$ items, respectively. Assume that the overhead of sorting is linear (this is roughly the case when comparing the overhead of sorting adjacent levels, which are of similar sizes). Before the optimization, the algorithm sorts buffers of sizes $4 \cdot 2^i, 8 \cdot 2^i, 2 \cdot 2^i$ and $4 \cdot 2^i$, which are of total length $18 \cdot 2^i$. After the optimization, it sorts buffers of sizes $2 \cdot 2^i, 4 \cdot 2^i, 2 \cdot 2^i$ and $4 \cdot 2^i$, which are of total length $12 \cdot 2^i$. The overhead of sorting, which is the bulk of the overhead of the entire construction, is therefore reduced by about 33%.

Using an advanced Cuckoo hashing scheme. The basic Cuckoo hashing scheme used in our construction utilizes approximately only 50% of its storage to store real and dummy items, while the remaining storage is empty. The new Backyard Cuckoo hashing [4] algorithm has a much better space utilization – in order to store n items, it requires only $(1 + \epsilon)n$ storage. Using this scheme has therefore the same effect as the optimization suggested above, of not storing empty entries in the hash table: it saves about 50% of the storage required by each level in the hierarchical structure. In addition, the overhead of each sorting operation is reduced by more than 50%, and the overhead of the entire construction is reduced by about 33%.

Reshuffling several levels together. In time t, where $t \bmod 2^i = 0$, and $t \bmod 2^{i+1} \neq 0$, the basic construction performs subsequent reshuffles of levels $1, 2, \ldots, i$. These reshuffles include many redundant steps. (For example, the first reshuffle inserts dummy items into the second level. Then, the reshuffle of the second level begins by (possibly) removing these items. Furthermore, the first reshuffle fills the second level, while the second reshuffle empties it.) Instead, it is possible to reshuffle together in a single step the contents of all these levels into level $i + 1$. According to our estimates this optimization saves an additional 33% of the total overhead.