Oblivious RAM Revisited

Benny Pinkas*
Dept. of Computer Science
University of Haifa
Mount Carmel, Haifa 31905, Israel
benny@pinkas.net

Tzachy Reinman
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, which can be used, for example, for search on encrypted data or for preventing cache attacks. However, oblivious RAM it is also commonly considered to be impractical due to its overhead, which is asymptotically efficient but is quite high: each data request is replaced by $O(\log^4 n)$ requests, or by $O(\log^3 n)$ requests where the constant in the "O" notation is a few thousands. In addition, $O(n \log n)$ external memory is required in order to store the n data items. 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 (with a small constant). This analysis is validated by experiments that we ran.

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

*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).

decrypt only data 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 [27]. In the context of RAM machines, this problem was investigated by Goldreich [12] and Ostrovsky [23] 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 [13].

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. Achieving this property implies in particular that the following information would be unknown to the server: (1) the locations of the accessed data items, (2) the order of data requests, and (3) the number of requests to the same location. In addition, different types of access (e.g., get-value, set-value, insert-new-value) must also be indistinguishable.

The cost of the best protocol of Goldreich-Ostrovsky was asymptotically efficient 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 of about 6100 (see Sect. 2); it can be replaced with $O(\log^4 n)$ with a reasonable constant. Note that for $n = 2^{20}$ data items, this $O(\log^4 n)$ factor translates to a factor of at least 160000, without taking into account the constants in the "O" notation.).

Due to the overwhelming overhead of the oblivious RAM protocol, it was often cited as a "theoretical" solution which could in principle 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.

The cost of oblivious RAM It is always preferable to lower the overhead of any computation that one must do. In the case of remote storage the financial incentive for this is pretty straightforward. Consider, for example, Amazon S3 (Amazon Simple Storage Service) [1], the popular cloud storage service. That service charges a monthly fee for each Giga byte stored ($0.15 per month, as of January 2010), as well as a charge for every Gbyte transferred ($0.10-$0.17) and for each operation request, such as PUT or COPY ($.01 per 1000 requests). Suppose that the client has a dataset of $n = 2^{20}$ items. Then, assuming that the constants in our scheme are about the same as in the $O(\log^4 n)$ scheme of Goldreich-Ostrovsky (GO) (which is roughly the case), the client needs to store, and pay, for about $1/\log n = 5\%$ storage as in the GO scheme, and replace each original data request with a fraction of $1/\log^2 n = 0.25\%$ of the requests needed in the GO scheme. Still, it is clear that using oblivious RAM to hide data access patterns substantially increases the price of remote data storage.

Other applications of oblivious RAM It was noted that many security applications are specific instances of the oblivious RAM problem. However, due to the impracticality of the best oblivious RAM constructions, considerable efforts were made to design specific solutions to these applications. 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. [30]).

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. Cache attacks reveal memory access patterns which can be used for cryptanalysis of cryptographic algorithms, by examining data-dependent table lookups. These attacks have been demonstrated to reveal AES keys in real systems [24]. As noted in [24], 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 Sect. 2.1 and Appendix A. 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 [25, 26], 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 [6], or $O(n \log n)$ comparisons, with a constant of about 6100, using the AKS network [3]. We perform sorting using the new randomized Shell sort algorithm of Goodrich [14]. 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 stress that the length of these counters is very small, and a counter can therefore be instantiated by a single 32 or 64 bit word. The data items themselves are typically much longer, in particular in the "remote storage in the cloud" application, where a data item might be a block of a few KBytes at the least. Fortunately, the client needs to store locally only $O(1)$ of these data items.)

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 $((v_1, x_1), \ldots, (v_n, x_n))$ and a corresponding sequence of operations, denoted by $(op_1, \ldots, op_m)$, where each operation is either a read operation, denoted* read(v)*, which retrieves the data of the item indexed by v, or a write operation, denoted* 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. It contains 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 [2] and of Damgård et. al. [10] 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 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. We start our description of related work with a description of the hierarchical solution of Goldreich and Ostrovsky [13], since this is the base scheme from which we start our work. We then describe concisely the performance of other schemes.

The construction of Goldreich-Ostrovsky uses 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. (For example, the server must not learn what was the previous location of the item which was moved to the head of the sorted items). An easy way to achieve this obliviousness is by using a sorting algorithm that accesses items in a predefined order, which is independent of the results of comparisons that the algorithm performs. In particular, they use a sorting network for this purpose, either the sorting network of Batcher [6] which performs $O(n \log^2 n)$ read operations with a very small constant (approximately $1/2$), or the sorting network of AKS [3] 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.)

2.1 The Hierarchical Solution

Structure The hierarchical solution uses a data structure composed of levels, where each level is twice the size of the former level. In each level, items are mapped to locations based on a random secret hash function. For n items, there are $N = O(\log n)$ levels, each of them is a hash-table with 2^i *buckets* ($i = 1 \ldots N$), where each bucket contains $O(\log n)$ items. The total memory size is $O(n \log n)$ items.

Lookup Any access to the data structure is composed of both a read and a write. First, the entire first level is sequentially scanned (i.e., is being read to the secure memory and decrypted); Second, in each of the other levels, one bucket, chosen by a hash function which looks random to anyone but the client, is entirely scanned. The item which was searched for is found in one of these two steps. Then, it is re-encrypted and written into the first level. It must be emphasized that when the requested item is found in a certain level, the process continues as usual in the following levels, except that a random dummy item is looked for, instead of the original requested item.

Reshuffling levels Since any access includes writing to the first level, this level becomes full and the data stored in it should be "moved down" to the next level. Similarly, after 2^i data access requests level i is moved down to level $i+1$. This is done by re-shuffling the data from both levels into the larger (lower) level, using a new random hash function, that is kept secret from the server. The reshuffling algorithm is the most complex component of the protocol and is its performance bottleneck. Reshuffling is composed of multiple scans of the data, obliviously sorting it several times, removing duplicate items, adding dummy items, tagging the items using the new random secret hash function and mapping them to buckets. The amortized computational overhead of reshuffling is $O(\log^3 n)$ per data request (using the AKS network [3], whose constant is about 6100; using Batcher's network [6] results in a total amortized overhead of $O(\log^4 n)$ with more reasonable constants).

A more detailed description of the Goldreich-Ostrovsky Hierarchical Solution can be found in Appendix A. For a full description, including formal proofs, we refer the reader to their original paper [13].

2.2 The "Square Root" Algorithm

The *square root algorithm* was presented by Goldreich and Ostrovsky as a methodic step towards the more efficient and complex hierarchical solution described above. We present this basic solution since its overhead is smaller than that of the hierarchical solution for data sets of reasonable size; in addition, subsequent work of Iliev and Smith [15, 16] was based on this construction.

In the square-root solution the n items are stored in a storage of size $n+2\sqrt{n}$, which includes a "cache" of size $\sqrt{n}$ into which all modified items are written. The number of actual accesses to the data storage is larger by a factor of $O(\sqrt{n}\cdot\log n)$ than in a naive storage. The algorithm is divided to epochs, each composed of $\sqrt{n}$ client's requests. Each request involves $O(\sqrt{n})$ accesses to the data (including a scan of the entire cache), and in addition at the end of each epoch the entire data must be sorted.

A more detailed description of the Goldreich-Ostrovsky square-root algorithm can be found in Appendix B. For a full description, we refer the reader to their original paper [13]. For a short description of the work of Iliev and Smith see Appendix C.

2.3 Using Merge Sort

Williams and Sion [32] present a solution based on the hierarchical solution of Goldreich and Ostrovsky, while assuming that the client can locally store $O(\sqrt{n})$ data items, rather than $O(1)$ as in our scheme and the scheme of [13]. This extended amount of local storage enables to reduce the number of accesses to the server to be only a factor of $O(\log^2 n)$ larger than the number of item requests. This property is achieved using an oblivious merge sort algorithm, which takes advantage of the following fact: when two sorted lists of random items are merged to a single list, it holds with overwhelming probability, for each index i, that among the first i items in the merged list at most $i/2+O(\sqrt{i})$ are from either one of the two lists. Given this

fact the oblivious merge sort algorithm can simply read items at a constant rate from the two lists (which were sorted recursively), and output, also in a constant rate, the sorted merged list. The fact ensures that the merge sort algorithm is oblivious and yet does not run out of the $O(\sqrt{n})$ secure local storage.

A short description of this solution can be found in Appendix D.

2.4 Using Bloom Filters

The construction of Williams et al. in [33] is based on the client storing at the server, for every level, an encrypted Bloom filter [8], and using it, together with the merge sort based construction of [32], in order to check whether an item appears in the level. The client again needs to use local storage of size $O(\sqrt{n})$. The work in [33] 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 $O(m)$. We show in Appendix E that 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 (see Table 1 below).

	computational overhead	client memory (data items)	server storage (data items)
Goldreich-Ostrovsky [13] $\sqrt{n}$	$O(\sqrt{n}\log n)$	$O(1)$	$O(n+\sqrt{n})$
Goldreich-Ostrovsky [13] Batcher	$O(\log^4 n)$	$O(1)$	$O(n\log n)$
Goldreich-Ostrovsky [13] AKS	$O(\log^3 n)$, const ≥ 6100	$O(1)$	$O(n\log n)$
Merge sort [32]	$O(\log^2 n)$	$O(\sqrt{n})$	$O(n\log n)$
Bloom filter [33]			
original analysis (inaccurate)	$O(\log n \log\log n)$	$O(\sqrt{n})$	$O(n)$
arbitrary # of hash functions	$O(\log n \cdot h2^{c/h} \log\log(h2^{c/h}n))$	$O(\sqrt{n})$	$O(nh2^{c/h})$
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: A comparison of the different access hiding schemes. (For the scheme of [33], we note that the original analysis is inaccurate. In the second line, h is the number of hash functions used in the Bloom filter, and the error probability is 2^{-c}. The third line is for an invocation using an optimal number h of hash functions, with specific numbers for an error probability of 2^{-64}; see analysis in Appendix E.)

2.5 A Comparison of All Schemes

Table 1 compares the performance of all access hiding schemes described in this section. Note that for the Bloom filter based scheme [33] the first line of the table lists the performance according to the original analysis in [33], which is inaccurate. The second and third lines list the performance according to our analysis in Appendix E, assuming an allowed error probability of 2^{-c}. The $O(\log n \log\log n)$ overhead in the third 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 [33] 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 $c \geq 9$.

The performance comparison can be summarized as follows:

- The constructions of Goldreich-Ostrovsky and of our work are the only ones using local storage of only $O(1)$ data items.
- 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).
- 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 [14] 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 re-ordering 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 [3, 9, 20, 14] 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 [14] and in Appendix F .

3.2 Cuckoo Hashing

Cuckoo hashing [25, 26] 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

	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

Table 2: Properties of Level i.

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., [17, 18, 19, 4, 5]) 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. Section 5 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[1]. 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 ℓth 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. (Without loss of generality, $h^{i,j}_{k,0}(v)$ can be implemented as a function of the parameters i, j, k, v, where i is the level index, j is the epoch, k is a secret key, and v is the item being searched for.) At the end of every epoch each level is re-hashed obliviously, using a new pair of hash functions. Table 2 summarizes the properties of level i.

[1] To be more accurate, the constant factor 2 may be replaced with any integer number, but for the simplicity of the presentation we choose 2.

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 $h^{i,\ell}_{k,0}(``dummy'' \circ t)$ and $h^{i,\ell}_{k,1}(``dummy'' \circ t)$, 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 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

[2] We assume here, as was implicitly assumed by all previous constructions [13, 32, 33], that the client does not perform a "read" operation for an item which does not exist in the remote storage. If such requests are allowed, then the following policy must be used: A read request for an item whose location-index does not exist in the database must be combined with a write request that writes a dummy value with the same location-index to the database.

[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.

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[4]. As we continue, we see that all these requirements are fulfilled.

Before describing the reshuffle process, we state a fact and a lemma which will be useful in analyzing the reshuffle.

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.*

Proof. The proof follows by induction. The fact trivially holds for the first level. Now, a reshuffle from level i to level $i+1$ occurs at times t where $t \bmod 2^i = 0$, i.e. at times where reshuffling occurs for all levels lower than i. The reshuffling is run just after the reshuffle from level $i-1$ to level i ends. By the step of the induction, at that time all levels $j \leq i-1$ are empty. The reshuffling process itself empties level i. □

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.*

Proof. The lemma is proved by induction on the levels. It trivially holds for the first level. Also, at time 0 all levels are empty. Consider now level $i+1$. The first reshuffle into that level occurs at time 2^i. At that time there are no items in level $i+1$ (since no items have yet been inserted to that level), and there are at most 2^i items in level i (according to the induction step). The reshuffle moves all items from level i to level $i+1$ and therefore when it ends there are at most 2^i items in level $i+1$. Consider now the next reshuffle, at time $2 \cdot 2^i$. Just before the reshuffle occurs there are at most 2^i items in level $i+1$ (since no items were inserted to that level in times $2^i+1, \ldots, 2 \cdot 2^i$), and there are at most 2^i items in level i (by the induction step). The reshuffle moves all items from level i to level $i+1$ and therefore when it ends there are at most 2^{i+1} items in level $i+1$. Immediately afterwards there it a reshuffle from level $i+1$ to level $i+2$ which empties level $i+1$. Levels $1, \ldots, i+1$ are now empty, and are in an identical state to their state in time 0. The proof therefore follows for the following two reshuffles, etc. □

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.

[4] Actually, for levels $i > 1$ this happens automatically, since whenever level i is reshuffled into level $i+1$, level $i-1$ is reshuffled into level i (due to epochs lengths). The first level does not require a deletion at all, items are written to it in a cyclic manner (unless the same item is written to it again, in which case it overwrites its previous value). There is no issue of obliviousness because it is always completely scanned.

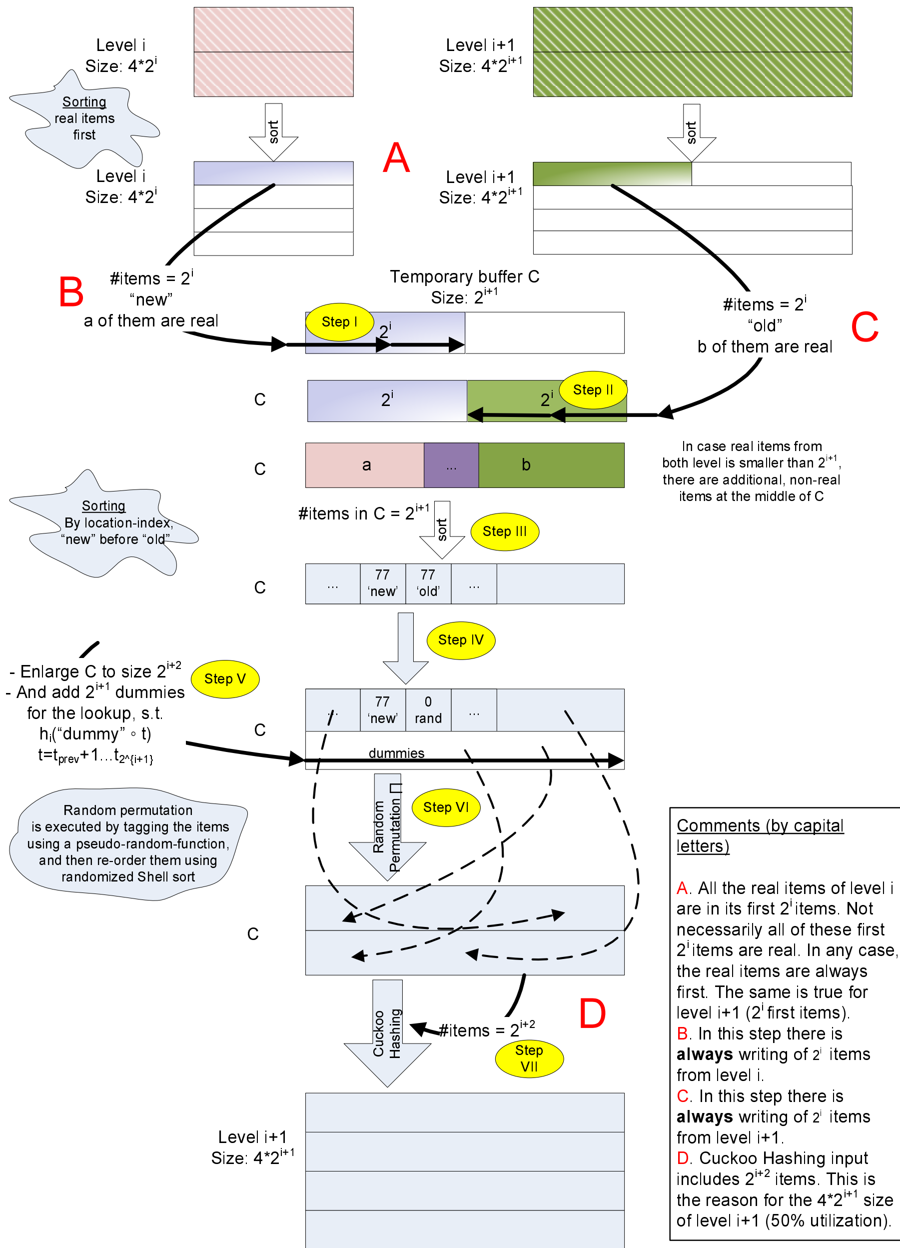

Figure 1: Reshuffle steps.

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.)

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 $j = 1 \ldots 2^{i+1}$.[5] 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 [26]), 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 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.

[5]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.

4.2 Analysis and Implementation

4.2.1 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 (Sect. 5 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 Sect. 5 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%. Section 5 discusses an additional optimization which reduces the constant factors of the overhead of the construction by an additional 33%.

4.2.2 Security.

Theorem 2. *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 $1 \dots 2M$, 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 jth 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 $(a_1, \dots, a_{2M})$ be a random permutation of the numbers $1, \dots, 2M$; 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. □

4.2.3 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, whose properties are: Intel Core 2 Quad CPU, Q9400 @ 2.66GHz, 2.67 Ghz, 3.25 GB of RAM, Physical Address Extension. The operating system was Windows XP, Service Pack 3.

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 3. The constant of the $O(k \log^2 k)$ overhead seems to be about 160. We note that the three improvements described in Sect. 5, (minimizing the amount of sorted

$\log_2 n$	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

Table 3: Performance measurements.

items – either by not sorting empty items, or by using an advanced Cuckoo hashing algorithm; avoiding duplicate items, and reshuffling several levels together) which have not yet been implemented by us, are estimated to reduce the overhead by about $33\%, 16\%$ and 33%, respectively. Applying both optimizations is likely to reduce the overhead by about 63%, and obtain a constant of about 60 in the "O" notation.

5 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, although not its asymptotic performance, and are beneficial for any implementation of the construction.

5.1 Storing Less Items

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 [5] 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%.

5.2 Avoiding Duplication in the Lookup Stage

Steps 5–6 of the reshuffle algorithm perform an oblivious sorting and a linear scan in order to removed duplicate occurrences of items which appear in both level i and level $i + 1$. This duplication occurs when an item is read from level $i + 1$, and its updated version is written to the top level. An alternative way of preventing duplication is not to have duplicate items in the first place, by deleting the "old" copy of the item in level $i + 1$ when it is being read during a data request. We describe next how this optimization can be done.

During a data request, the algorithm searches for the requested item in all levels. In each level the algorithm probes two locations. When the location in which the item is stored is found, it must be written back with a default encrypted invalid value. All other probed locations are written back with their previous values, also re-encrypted. In other words, in each level the client reads two locations and writes back values to these locations. For all locations but the one in which the seeked item is found, the values written back are re-encryptions of the original values. In the location in which the desired item was found, the client writes back a default encrypted invalid value.

This modification of the scheme saves one sorting of the temporary buffer in the reshuffling process, at the cost of additional write operations in the lookup process. I.e., for each lookup we add $2 \log n$ writes, but we save a sorting which costs $O(2^{i+1} \log 2^{i+1})$ operations per a reshuffle of levels i and $i + 1$. The reshuffle of level i occurs every 2^i lookups, and therefore we save there $O(2 \log 2^{i+1}) = O(i)$ operations per lookup. Summing over all levels $i = 1, \ldots, \log n$ we save about $O(\log^2 n)$ operations per lookup.

In Sect. 5.1 we estimated that in the original algorithm sorting costs about $18 \cdot 2^i$ operations per level, whereas after the optimization of that section the overhead of sorting is reduced to $12 \cdot 2^i$. The optimization we presented here removes a sorting of 2^{i+1} items, whose overhead according to that approximation is about $2 \cdot 2^i$. This reduces the overhead of the sorting operations to $10 \cdot 2^i$, which is an addition improvement of 16%.

5.3 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.

5.4 Overall Effect

Combining the different optimizations presented above, the overhead of reshuffling is reduced to about $(10/18) \cdot (2/3) = 37\%$ of its original overhead, a saving of about 65%.

6 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 [4] 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

We wish to thank Yuriy Arbitman for informing us of the randomized Shell sort result, and Ilya Mironov for suggesting to remove duplicate items at the loopup stage.

References

[1] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[2] M. Ajtai. Oblivious RAMs without cryptographic assumptions. STOC 2010, 2010.

[3] M. Ajtai, J. Kolmós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC*, pages 1–9, 1983.

[4] Y. Arbitman, M. Naor, and G. Segev. De-amortized Cuckoo hashing: Provable worst-case performance and experimental results. In *ICALP (1)*, pages 107–118, 2009.

[5] Y. Arbitman, M. Naor, and G. Segev. Backyard Cuckoo hashing: Constant worst-case operations with a succinct representation. Manuscript, 2010.

[6] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 32(1968).

[7] V. E. Benes. Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, 4, 1964.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill and The MIT Press, 1990.

[10] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. Cryptology ePrint Archive, Report 2010/108, 2010. http://eprint.iacr.org/2010/108.

[11] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):293, 2000.

[12] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194. ACM, 1987.

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

[14] M. T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *Proceedings 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[15] A. Iliev and S. W. Smith. Private information storage with logarithm-space secure hardware. In *International Information Security Workshops*, pages 199–214, 2004.

[16] A. Iliev and S. W. Smith. Protecting client privacy with trusted computing at the server. *IEEE Security & Privacy*, 3(2):20–28, 2005.

[17] A. Kirsch and M. Mitzenmacher. Using a queue to de-amortize Cuckoo hashing in hardware. In *Proceedings of the 45th Annual Allerton Conference on Communication, Control, and Computing*, pages 751–758, 2007.

[18] A. Kirsch and M. Mitzenmacher. Simple summaries for hashing with choices. *IEEE/ACM Trans. Netw.*, 16(1):218–231, 2008.

[19] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. In *ESA*, pages 611–622, 2008.

[20] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.

[21] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Computing*, 17(2):373–386, 1988.

[22] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[23] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC '90*, pages 514–523. ACM, 1990.

[24] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[25] R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*, pages 121–133, London, UK, 2001. Springer-Verlag.

[26] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

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

[28] R. Sedgewick. Analysis of Shellsort and related algorithms. In *ESA '96: Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 1–11, London, UK, 1996. Springer-Verlag.

[29] D. L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.

[30] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy, 2000. S&P 2000. Proceedings*, pages 44–55, 2000.

[31] S. Wang, X. Ding, R. H. Deng, and F. Bao. Private information retrieval using trusted hardware. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *ESORICS*, volume 4189 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2006.

[32] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[33] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, 2008.

A The Hierarchical Construction

The hierarchical solution of Goldreich-Ostrovsky uses a data structure composed of levels, where each level is twice the size of the former level. In each level, items are mapped to locations based on a random hash function which is only known to the client. Each item is searched for in every level. Even if an item is found in a certain level the algorithm continues searching for dummy items in lower levels in order to hide from the server the fact that the item was found. When the item is rewritten back to the storage, it is always written to the top level. Therefore, after some data access requests the top level becomes full. The system then performs a reshuffling algorithm which moves items from the first level to the next level, and reorders the items in the next level according to a new hash function. Subsequently, after 2^i data access requests level i is reshuffled with level $i+1$. The reshuffling algorithm is the main bottleneck of the system.

In more detail, the data structure is composed of levels (also called *buffers*), numbered $i = 1 \dots N$, where each of them is a hash-table with 2^i *buckets*, where a bucket contains $m = O(\log n)$ items, [6] n is initialized to be the number of items that are potentially stored in the data storage, and $N = \lceil \log_2 n \rceil + 1$. The total memory size of this structure is $\sum_{i=1}^{\lceil \log_2 n \rceil + 1} 2^i \cdot O(\log n) = O(n \log n)$. For each level we associate an *epoch*, which is defined for level i as 2^i data access requests. Each level is re-hashed (obliviously, using a new hash function), at the end of every epoch.

Initially, the data structure is empty. For each request of a location-index (virtual address) v, the following operations are performed.

[6] The size of a bucket is set to be $O(\log n)$ in order to avoid overflow. This is due to a simple balls and bins result, that for a large enough n, if n balls are randomly placed into k bins, with probability greater than $\frac{n-1}{n}$, the fullest bin has fewer than $3\frac{\ln n}{\ln \ln n}$ balls [22]. So, for the lowest (and largest) level, a bucket should be able to contain $O(\log n)$ items. Asymptotically, decreasing the size of buckets (to i) in upper levels does not save storage.

1. Scan through the entire first level in a sequential order to find an 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, but the process continues as usual. If more than a single item is found, the last item found is stored.
2. Go through all other levels, and for each level $i = 2 \ldots N$, do:
   - If v has not yet been found, scan through the entire bucket $h_{i,j}(v)$ in level i ($h_{i,j}$ is a hash function whose range is $\{1 \ldots 2^i\}$ used by level i in its jth epoch; the bucket size is $O(\log n)$). If the requested item is found, it is stored in the client's secure memory, and the process continues as usual. This step includes sequentially reading all the items in the scanned bucket.
   - If v has been found, scan through the entire bucket whose index is $h_{i,j}(``dummy'' \circ t)$, where t is a counter for the number of requests till now (this gives a randomly chosen dummy value). This step includes sequentially reading all items in the scanned bucket.
3. Scan again through the entire first level in a sequential order, and write back the updated (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 the items in the first level.

Notice that all the operations that are done for each request do not reveal any information on the address accessed. Steps 1 and 3 scan an entire level, and are identical for any access. Step 2 always goes through all the levels (except the first level), regardless of whether the requested item was found already or not, and the bucket is selected using a random hash function applied to a fresh input. This bucket, in turn, is completely scanned. The number of access operations in steps $1-3$ for each request is $O(\log^2 n)$. Steps 1 and 3 access the entire first level, whose size is $O(\log n)$, step 2 goes through $N = O(\log n)$ levels, and on each level it access a whole bucket, whose size is $O(\log n)$.

The first level essentially functions as a cache, meaning that for any request, the updated value is written to the first level. Since the capacity of this level is final, after a certain number of requests, the first level becomes full. In order to avoid this, the content of the first level is "moved down" to the second level. Now the second level may become full, so the same process is repeated. When the content of the last level (level N) has to be "moved down", a new level with twice the number of buckets is allocated.

"Moving down" the content of level i is done at the end of the ith epoch, i.e. every 2^i requests, to ensure that a level is "moved down" before it becomes potentially exhausted. [7] 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 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. Whenever level i is moved down to level $i+1$, the lower level is reshuffled (and this happens at the end of an epoch).

Reshuffling levels When the client moves the content of level i to level $i+1$, it obliviously hashes the content of both levels to level $i+1$ and leaves level i empty. This reshuffling must ensure that if there is a duplicate item (with the same location-index, and possibly different data content) in levels i and $i+1$, the newer item (from the upper level) is kept, and the older item is deleted. The resulting buffer, level $i+1$ after

[7] Actually, the first level becomes potentially exhausted only after $\log n$ requests, so moving it down could have been done after about $\log n$ requests, rather than 2 requests. The threshold of 2^{i-1} is due to the last level, which might become full after $2^{N-1} = \log n$. Asymptotically, applying the threshold of the last level to all the levels does not cost.

the reshuffling, must be ordered in a way which is independent of any of the levels before the reshuffling; Also, the content of level i must be deleted[8].

Level i can hold 2^i items, and level $i+1$ can hold 2^{i+1} items. However, it holds that jointly, the buckets in both levels contain at most 2^{i+1} items (otherwise, a reshuffle would have executed earlier). So the goal is to have these up to 2^{i+1} items (each a pair of location-index and a data) relocated in level $i+1$, according to a new random hash function h, whose range is $\{1 \dots 2^{i+1}\}$ (the number of buckets in level $i+1$). With high probability (see Footnote 6), h maps no more than $i+1$ items to the same bucket. Since $i+1 \leqslant \log n$, no bucket is overflowed (in the rare case on which h does map more than $i+1$ items to the same bucket, the algorithm re-chooses a new random hash function (to replace h)).

The reshuffling process consists of 12 steps enumerated below, and is based on two basic primitives: (1) *Scanning*, which is reading in a sequential order all the items in a given buffer and possibly updating them; (2) *Oblivious Sorting* (O-Sort), which is tagging elements using a PRNG and sorting by these tags, yielding a random permutation.

1. Move the items from the two buffers to a temporary buffer C whose size is $(2^i + 2^{i+1}) \cdot \log n$; Mark each item as "new" (if it is copied from the upper level, i) or "old" (if it is copied from level $i+1$).

2. *Moving duplicate items near each other:* O-sort C according to the following criterions (ordered): (a) non-empty items come before empty ones, (b) smaller location-indices (virtual addresses) come first, (c) new items come before old.

3. *Erasing duplicates:* Scan C and erase each old item preceded by a new item with the same location-index.

4. Add dummies (items with location-index 0 and random content), such that C contains exactly 2^{i+2} non-empty items (the remaining $(2^i + 2^{i+1}) \cdot \log n - 2^{i+2}$ items are empty).

5. *Mapping to buckets;* Scan C and use the new hash function h to tag the 2^{i+2} non-empty items with the indices of the 2^{i+1} buckets of level $i+1$ (non-dummies are hashed using their location-index as a parameter, dummies are hashed using their location-index and their random content).

6. O-sort C according to these criterions (ordered): (a) non-empty items before empty ones, (b) lower tags first.

7. *Checking that no more than* $\log n$ *items were mapped to a bucket:* Scan C and check that no tag was given to more than $m = O(\log n)$ items. if there is such a tag (h is overflowing), go to step 5).

8. Scan C and tag $2^{i+1} \cdot m$ of the empty items, where each tag is a bucket index and it is given to exactly $m = O(\log n)$ items.

9. O-sort C according to these criterions (ordered): (a) tagged items before non-tagged ones, (b) lower tags first, (c) non-empty items before empty ones (among items with the same tag).

10. Scan C and erase tags of empty items.

[8] Actually, for level $i > 1$ this happens automatically, since whenever level i is reshuffled into level $i+1$, level $i-1$ is reshuffled into level i (due to epochs lengths). The first level does not require a deletion at all, items are written to it in a cyclic manner (unless the same item is written to it again, in which case it overwrites its previous value). There is no issue of obliviousness because it is always completely scanned.

11. *Ensuring that exactly m items are mapped to each bucket:* O-sort C according to these criterions (ordered): (a) tagged items before non-tagged ones, (b) lower tags first.

12. Scan C and clean it – erase the tagging and empty items that do not contain location-indices. Move the $2^{i+1} \cdot \log n$ prefix of C to level $i+1$.

After step 1, all the content of levels i and $i+1$ is in the temporary buffer C; steps 2 and 3 handle the possible duplication of items with the same location-index in the two original buffers; steps $4-10$ end with a tags-sorted array, where each tag is applied to exactly m items; step 11 deletes "spaces", such that the $2^{i+1} \cdot \log n$ tagged items are located as the prefix of C, which is equivalent to 2^{i+1} buckets of size $\log n$, and this is copied (with some cleaning work) to level $i+1$ in step 12.

Overhead: The main overhead of the reshuffle algorithm comes from running the oblivious sorting. Sorting is performed four times during the reshuffle of layer i, where the largest array sorted in this level is of length $(2^{i+2}+2^{i+1}) \cdot \log n$ (in Step 9). Sorting this array using the AKS network takes $O(2^i i \log n)$ comparisons. The reshuffle happens every 2^i requests, so the amortized cost per request is $O(i \log n)$ comparisons. Summing this over levels $i = 1, \ldots, \log n$ gives a total amortized overhead of $O(\log^3 n)$. Replacing the AKS network with Batcher's network which uses $O(n \log^2 n)$ comparisons results in a total amortized overhead of $O(\log^4 n)$. (Instead of using bucket of size $O(\log n)$ is every level it is possible to use buckets of size $O(\log i)$ in level i. This reduces the constant factors of the actual overhead but not its asymptotic performance.)

B The "Square Root" Construction

At the beginning of an epoch, the permuted area is being permuted, that is, a permutation P over the integers $[1, n+\sqrt{n}]$ is uniformly selected and the content of the permuted memory is obliviously relocated accordingly. Next, for each client's requests, to location-index i, the following is performed.

1. Scan through the entire shelter in a sequential order to find an item whose location-index is i. This step includes reading all the items in the shelter. If the requested item is found, it is stored in the client's secure memory, and the process continues as usual.

2. If the item is not found in the shelter, access the actual location $P(i)$ in the permuted area. The requested item is there, and is now being stored in the client's secure memory.
   If the item is found in the shelter, access the next dummy item (in the location $P(n+j)$ where j is the number of the step in this epoch).

3. Scan again through the entire shelter in a sequential order, and write back the updated (re-encrypted) item of location-index i, in the next available location. If i is already in the shelter, overwrite it. This step includes reading and writing all the items in the shelter.

It is easy to see, that steps 1 and 3 are trivially oblivious, in the sense that they are identical for each request. Step 2 are oblivious due to the randomness of the permutation P.

At the end of an epoch, the values in the shelter are obliviously written to the permuted area. This can be done by obliviously sorting the entire storage ($n + 2\sqrt{n}$ items) by the location-indices (dummy's index is infinite), and in case of duplication, leave the newer value, that is from the shelter, and modify the older's location-index to infinite. This practically replaces the old values that are in the permuted area with the new

values that are in the shelter, and resets the shelter. This process with the permuting of the permuted area (at the beginning of each epoch), is called *reshuffling*.

The number of access operations in steps $1-3$ for each request is $O(\sqrt{n})$ (mainly due to steps 1 and 3). The number of access for the sorting during the moving of data at the end of an epoch is $O(n\log^2 n)$, divided by $\sqrt{n}$ (number of requests per epoch) gives a total overhead factor of $O(\sqrt{n}\cdot\log^2 n)$.

C Implementation of the "Square Root" Solution, using a Trusted Computing at the Server

Iliev and Smith [15, 16] designed and implemented a secure storage system based on a variation of the "Square Root" algorithm (this algorithm was used because of its superior performance for reasonable size data sets). Their system used a secure co-processor (SCOP) as the client "representative" at the server machine. This makes the server more expensive but requires less communication. In addition, they improve the reshuffling procedure – they use Benes Permutation Network [7] instead of Batcher Sorting Network [6] (except the initial shuffle that is over all the database – which is done using a Batcher network)[9].

Other modifications they have introduced into the "Square Root" solution are the following:

- For efficiency, instead of scanning all the "shelter" for each request, the algorithm scans only the "active" items in the shelter, meaning that for the first request, no item is scanned in the shelter, for the second request – one item, and so on, such that the first requests are faster than the later.
- In the reshuffling procedure, they use a Luby-Rackoff cipher [21] as a pseudorandom permutation, instead of using a truly random permutation. This allows the SCOP to compute inverse permutations and composition of permutations, such that the SCOP should store only the last one or two permutations, and not the whole sequence of permutations.
- In the reshuffling procedure, they distinguish between items that have been "touched" since the previous reshuffling, and items that have not been "touched", and they require to shuffle only the "touched" items. Re-ordering (according to the new permutation) the untouched items is done in a non-oblivious way, and only the re-ordering of the touched items and the merging of the two groups of items must be done obliviously.

Criticism It is not clear how the distinction between "touched" and "untouched" remains secret. In a first glance, it seems that an adversary (in particular, the server) is able to follow this distinction and conclude which items were touched since the last reshuffling. Furthermore, even if this can be hidden, the server can count the number of touched vs. untouched items and conclude the amount of repeated requests since the last reshuffling. The last drawback is also a result of scanning only the "active" part of the shelter – the size of this part implies the amount of repeated requests.

D Merge Sort Based Solution

Williams and Sion [32] base their solution on the hierarchical solution of Goldreich and Ostrovsky. In order to improve the computational overhead to $O(\log^2 n)$, they require a $O(\sqrt{n})$ secure client storage, and replace the reshuffling stage with their version of reshuffling, using *Oblivious Merge Sort*.

[9]A different design with $O(n/k)$ cost, where k is the amount of secure storage available to the client, was introduced in [31].

Their reshuffling algorithm composed of three stages. First, the real items are copied (re-encrypted) to a temporary buffer at the server storage (the dummy ones are removed). Second, these items are re-located according to a new hash function, using an oblivious merge sort. Third, the items are copied back to their level, with additional dummy items. The first stage is done by scanning the level, reading only the real items to a client local queue, and when it is about half full, the client start to write items from the queue to the buffer (while continuing scanning). The queue size is $O(\sqrt{n})$ and the authors prove that with high probability it never overflows. The second stage is based on a similar idea, and is done recursively (recursion depth is $O(\log n)$, using client $O(\sqrt{n})$-sized queues). Its correctness depends on the assumption that the items are initially located uniformly (a property of the permutation). At the final stage, a similar idea (to the one in the first stage) is used to copy back to the level the items from the temporary buffer with additional dummies (again, using a client $O(\sqrt{n})$ queue).

E Analysis of the Overhead of the Bloom Filter Based Solution

Let us denote the number of items stored in a certain level as m, and the number of hash functions used in the Bloom filter to encode these items as h. Let us also denote the number of bits in the Bloom filter as M. The value of h is set in Section 4.7 of [33] to be $h = 5$, and it also required there that error probability of the Bloom filter be 2^{-64}. We analyze first the performance based on a setting with $h = 5$ hash functions, and then analyze performance in a setting with an optimal number of hash functions.

E.1 An Analysis for the Case of $h = 5$ Hash Functions

Denote the error probability of the filter as 2^{-c}. The error can be expressed as

$$(1 - (1 - 1/M)^{hm})^h \approx (1 - e^{-h\frac{m}{M}})^h = 2^{-c}.$$

Therefore a filter with h hash functions, and with an error probability of 2^{-c}, must store the following number of bits per item item:

$$\frac{M}{m} = \frac{-h}{\ln(1 - 2^{-c/h})} \approx h \cdot 2^{c/h}.$$

In other words, a level with m items requires a filter with $M = h2^{c/h}m$ bits. For the parameters of [33], $h = 5, c = 64$, this means that the filter must have about $M = 35500 \cdot m$ bits.

The process of shuffling a level of m items includes (in Step 5 of [33]) sorting the Bloom filter by doing $O(M \log\log M)$ accesses to the remote storage. In [33] it is assumed that this overhead is essentially $O(m)$ but, in fact, plugging in the previous estimates, it is $O(h2^{c/h}m \log\log(h2^{c/h}m))$. This is for a level which has m items. Therefore the amortized overhead of this step alone per accessing an item in the storage (which has $\log n$ levels, of size 2^i for $i = 1, \ldots \log n$) is $\sum_{i=1}^{\log n} O(h2^{c/h} \cdot 2^i \cdot \log\log(h2^{c/h}2^i))/2^i \approx O(h2^{c/h} \log\log(h2^{c/h}n) \log n)$.

Assume, as suggested in [33], that $h = 5$, $c = 64$, then Step 5 alone incurs an overhead of $35500 \cdot \log n \cdot \log(15 + \log n)$ accesses per item request. For $n = 2^{20}$ this overhead is about $3.5 \cdot 10^6$. Assuming a lower error parameter of $c = 32$, and $n = 2^{20}$, the overhead per request is about 40000. Note that these estimates are without taking into account other constants in the "O" notation.

Let us compare the overhead of [33] with the $O(\log^2 n)$ overhead of our scheme, while ignoring the constants in the "O" notation (which are rather similar in both cases). The protocol of [33] is better only when

$h2^{c/h} \log\log(h2^{c/h}n) \log n < \log^2 n$. For the case $h = 5$, $n = 2^{20}$, this requires that the security parameter satisfies $c < 5$, namely that the error probability is greater than $1/32$, which is clearly unreasonable.

E.2 An Analysis Based on an Optimal Number of Hash Functions

The work in [33] suggests using $h = 5$ hash functions. It is well known that for given values of m (number of items) and M (size of filter), the optimal number of hash functions is $h = 0.7M/m$ (see, e.g., [11]). It is also known that in that case, in order to achieve an error probability of at most 2^{-c}, the size of the filter must be at least $M = c \cdot m/\ln 2 \approx 1.44c \cdot m$. Therefore, $h = c$.

Given these observations, and assuming an allowed error probability of 2^{-64}, the number of hash functions, and therefore the number of queries to the Bloom filter at each request and in every level is 64 (which is quite high compared to the two queries to the Cuckoo hashing table in out scheme), and the size of the filter at the last level must be at least $92n$ bits. The overhead of Step 5 for a level with a Bloom filter of size M is $O(M \log\log M)$. It translates to a total overhead of $O(M \cdot \log n \cdot \log\log M)$ for n queries, where M is the size of the largest Bloom filter used. Since this filter is of size $M = 92n$, the total *amortized* overhead per query is $O(\log n \log\log n)$ with a constant of at least 92 in the "O" notation. Had we used an error parameter of 2^{-32}, the constant in "O" notation would have been at least 46. These constants are on top of any constants incurred by applying the algorithm several times per level, and in each level. (These constants are similar to those in our solution.) We can therefore conclude that for any reasonable value of n, this $O(\log n \log\log n)$ overhead is worse than the $O(\log^2 n)$ overhead of our scheme, since it will hold that $92 \log\log n > \log n$).

F Details of the Randomized Shell Sort Algorithm

Shell sort [29] is a sorting algorithm, called after its inventor, Donald Shell. The basic building block of Shell sort, is sorting subsets of the given array, usually using an insertion sort. The subsets are according to a given sequence of offsets $\{o_1, o_2, \ldots\}$ such that at the first pass, the subsets are $\{0, o_1, 2o_1, \ldots\}$, $\{1, 1 + o_1, 1 + 2o_1, \ldots\}$, and so on, at the second pass the subsets are $\{0, o_2, 2o_2, \ldots\}$, $\{1, 1 + o_2, 1 + 2o_2, \ldots\}$, and so on. The last offset must be 1, meaning sorting the whole array. The efficiency of this algorithm depends on the choosing of the sequence of offsets, and is due to the tendency of some sorting algorithms (in particular insertion sort) to be more efficient if the input is "almost sorted". A common sequence of offsets for sorting an array of length n is $\{n/2, n/4, \ldots\ n/n = 1\}$. The analysis of the average-case is an open problem and there is no proof that it is $O(n\ logn)$ (optimal) [28].

In his recent work, Goodrich [14] suggests a simple, randomized, data-oblivious version of the Shell sort algorithm that always runs in $O(n\ logn)$ time and sorts any input correctly with very high probability. The innovation in this new approach is that in the inner sorting, it refers to *regions* and compares randomly selected pairs of elements of the two regions being compared. In order to make the presentation of the andomized Shell sort more clear, we first describe again the traditional Shell sort, in a different way.

Let the sequence of offsets used to sort an array A of length n be $\{n/2, n/4, \ldots\ 1\}$. For each offset $o_i \in O$ we define *regions* $\{r^i_j\}$ of length o_i. For example, for $o_1 = n/2$, region r^1_1 is $A[0 \ldots n/2 - 1]$, and r^1_2 is $A[n/2 \ldots n - 1]$; for $o_2 = n/4$, region r^2_1 is $A[0 \ldots n/4 - 1]$, r^2_2 is $A[n/4 \ldots n/2 - 1]$, etc.

In its first pass, Shell sort sorts the following subarrays: $\{r^1_1[0] = A[0], r^1_2[0] = A[n/2]\}$, $\{r^1_1[1] = A[1], r^1_2[1] = A[n/2 + 1]\}$, and so on. In the second pass, the subarrays are: $\{r^2_1[0] = A[0], r^2_2[0] = A[n/4], r^2_3[0] = A[n/2], r^2_4[0] = A[3n/4]\}$, $\{r^2_1[1] = A[1], r^2_2[1] = A[n/4+1], r^2_3[1] = A[n/2+1], r^2_4[1] = A[3n/4 + 1]\}$, ..., $\{r^2_1[j] = A[j], r^2_2[j] = A[n/4 + j], r^2_3[j] = A[n/2 + j], r^2_4[j] = A[3n/4 + j]\}$, for

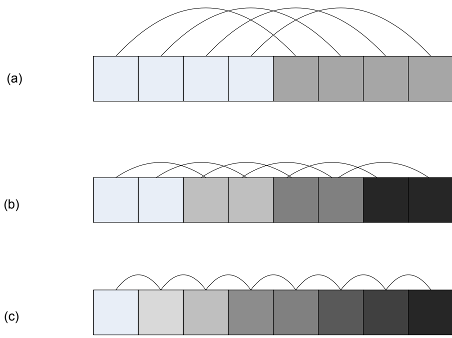

Figure 2: Traditional Shell sort (a-c).

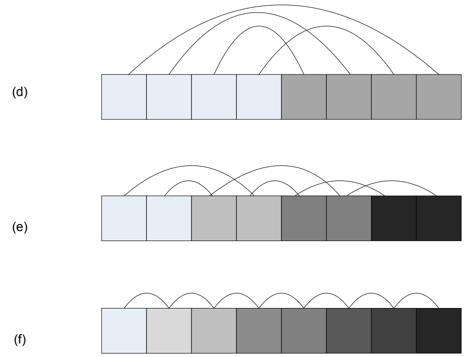

Figure 3: Randomized Shell sort (d-f).

$j = 0 \dots n/4 - 1$, and so on ($\log n - 1$ passes). In the last pass, it sorts the entire array (which is already almost sorted). See Fig. 2 (a-c).

It is easy to notice that each subarray that is sorted in Shell sort contains one representative from each region. In each pass, the kth subarray contains the kth element of each region. Here comes the first change in randomized Shell sort. Instead of taking the kth element of each region, it **randomly** selects one representative from each region. See Fig. 3 (d-f) for illustration. Naturally, the last pass is identical in both variants – the regions' size is 1, and the entire array is sorted.

In addition to randomizing the selection of a region representative, randomized Shell sort has additional unique attributes. We now define randomized Shell sort more carefully.

For offset o_i, we compare pairs of the regions $\{r^i_j\}$, first using a shaker pass and then an extended brick pass (these together are the *shaker-brick* pass, which replaces the insertion sort pass in the traditional Shell sort):

- **Shaker Pass.** An increasing sequence of adjacent-region comparisons, followed by a decreasing sequence of adjacent-region comparisons, i.e., compare the following pairs:
  $\{r^i_1, r^i_2\}, \{r^i_2, r^i_3\}, \dots, \{r^i_{n/o_i-2}, r^i_{n/o_i-1}\}$ and than compare these pairs in the inverse order
  $\{r^i_{n/o_i-1}, r^i_{n/o_i-2}\}, \dots, \{r^i_2, r^i_1\}$
- **Extended Brick Pass** An increasing sequences of: regions 3 offsets apart, then of 2 offsets apart, and finally odd-even adjacent and even-odd adjacent, i.e., compare the following pairs:
  $\{r^i_1, r^i_4\}, \{r^i_2, r^i_5\}, \dots, \{r^i_{n/o_i-4}, r^i_{n/o_i-1}\}$, than $\{r^i_1, r^i_3\}, \{r^i_2, r^i_4\}, \dots, \{r^i_{n/o_i-3}, r^i_{n/o_i-1}\}$,
  than $\{r^i_1, r^i_2\}, \{r^i_3, r^i_4\}, \dots, \{r^i_{n/o_i-3}, r^i_{n/o_i-2}\}$, and finally $\{r^i_2, r^i_3\}, \{r^i_4, r^i_5\}, \dots, \{r^i_{n/o_i-2}, r^i_{n/o_i-1}\}$

In the traditional Shell sort, when two regions are compared, the elements of these regions are participating in the comparison, paired by their indices, i.e., the jth element of one region is compared with the jth element of the other, for $j = 1 \dots l$ where l is the regions length. In contrast, in randomized Shell sort, in each pass, pairing is done randomly, and it is repeated a constant number of times, denoted $C \geq 1$.

As mentioned earlier, randomized Shell sort sorts any input correctly with very high probability. According to experiments in [14], the error-probability of the algorithm is less than 0.01% for $C = 1$. Choosing carefully the parameters (e.g., enlarging C) can decrease this probability. In our use of randomized Shell sort we can sort the smaller-sized levels using bubble sort (the cost of this is negligible due to the sizes of the sorted levels), and use randomized Shell sort only for sorting the larger levels.

An external observer cannot distinguish between two runs of the Shell sort algorithm (both the original one, and the randomized variant). The basic primitive used is comparing two elements, which can be done obliviously with $O(1)$ secure storage. The advantage of the randomized Shell sort algorithm is its performance. While Goodrich [14] proves that this algorithm runs in $O(n \log n)$, the same problem for the traditional Shell sort is still open [28].