On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme

Eyal Kushilevitz* Steve Lu† Rafail Ostrovsky‡

Abstract

With the gaining popularity of remote storage (e.g. in the Cloud), we consider the setting where a small, protected local machine wishes to access data on a large, untrusted remote machine. This setting was introduced in the RAM model in the context of software protection by Goldreich and Ostrovsky. A secure Oblivious RAM simulation allows for a client, with small (e.g., constant size) protected memory, to hide not only the data but also the sequence of locations it accesses (both reads and writes) in the unprotected memory of size n.
Our main results are as follows:

- We analyze several schemes from the literature, observing a repeated design flaw that leaks information on the memory access pattern. For some of these schemes, the leakage is actually non-negligible, while for others it is negligible.
- On the positive side, we present a new secure oblivious RAM scheme, extending a recent scheme by Goodrich and Mitzenmacher. Our scheme uses only $O(1)$ local memory, and its (amortized) overhead is $O(\log^2 n/\log\log n)$, outperforming the previously-best $O(\log^2 n)$ overhead (among schemes where the client only uses $O(1)$ additional local memory).
- We also present a transformation of our scheme above (whose amortized overhead is $O(\log^2 n/\log\log n)$) into a scheme with worst-case overhead of $O(\log^2 n/\log\log n)$.

Keywords: Oblivious RAM, Cuckoo Hashing, Secure Computation.

1 Introduction

Consider the following problem: a small protected CPU wants to run a program that requires access to unprotected RAM, without revealing anything but the running time and the size of memory used by the program. Encryption can be employed to hide the *contents* of the memory cells accessed by the program, but the sequence of read and write locations made to the RAM may leak information as well. The notion of Oblivious RAM was introduced by Goldreich [9] as a means to hide both the contents and the so-called "access pattern"

*Computer Science Department, Technion. Supported by ISF grant 1361/10 and BSF grant 2008411. E-mail: eyalk@cs.technion.ac.il

†E-mail: steve@stealthsoftwareinc.com

‡Department of Computer Science and Department of Mathematics, UCLA. This material is based upon work supported in part by NSF grants 0830803, 09165174, 1065276, 1118126 and 1136174, US-Israel BSF grant 2008411, grants from OKAWA Foundation, IBM, Lockheed-Martin Corporation and the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014-11-1-0392. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. E-mail: rafail@cs.ucla.edu

of memory probes. This can be viewed as casting into the "RAM model" the work on oblivious simulation of Turing machines [21]. The oblivious RAM model also fits the setting of a client that stores its data in a remote-storage server and has various other applications (see, e.g., in [20]).

Previous Work. The initial results on Oblivious RAM were presented by Goldreich [9] and featured two solutions: a "square-root" solution (with low round complexity) and a "recursive square-root" solution. In the "square-root" solution, the time overhead for executing the program increases by a factor of $O(\sqrt{n})$, while in the recursive solution the time overhead increases by $O(2^{\sqrt{\log n \log\log n}})$ factor, where n is the number of memory locations used by the original program. Ostrovsky [16, 17] proposed a different oblivious simulation for which the time requirement is increased only by a factor of $O(\min\left\{\log^3 n, \log^3 t\right\})$, where t is the program running time. Ostrovsky's simulation algorithm is referred to as the *hierarchical algorithm*, because of the nature of the data structure used by the RAM.

The high-level description of Ostrovsky's hierarchical solution can be stated as follows: there is a sequence of buffers whose size grow at a geometric rate, and smaller buffers are reshuffled into larger ones as they fill up. In the original work of [16, 17], the buffers were standard hash-tables with sufficiently large buckets, and it employed a technique known as *oblivious shuffling* for the reshuffles. A slightly different, somewhat simpler reshuffling methods was proposed by Goldreich and Ostrovsky [10],[1] though the asymptotic behavior of reshuffles in [16, 17] and in [10] is the same.

In all of the above solutions, to protect the data *contents*, a private-key encryption and authentication scheme was used and, each time elements were read and written back to the insecure RAM, they would be re-encrypted and authenticated.

The efficiency of Oblivious RAM is measured by three main parameters: the amount of local (client) storage, the amount of remote (server) storage, and the (amortized) overhead of reading or writing an element. Goldreich [9] used sub-linear local storage, while [16, 17, 10] all used a constant amount of local storage. Ostrovsky's hierarchical solution used $O(n \log n)$ remote storage, and offered two different ways to perform oblivious shuffling that led to either $O(\log^4 n)$ access overhead with a small hidden constant, or $O(\log^3 n)$ access overhead with a large hidden constant.

Subsequent works improved upon Ostrovsky's hierarchical solution. Williams and Sion [23] introduced a method to perform oblivious sorting by using $O(\sqrt{n})$ local memory to reduce the access overhead down to $O(\log^2 n)$ (but with super-linear remote storage). Subsequently, Williams et al. [24] made use of Bloom filters to further reduce the overhead to $O(\log n \log\log n)$ and the remote storage to $O(n)$ but, as we shall see, this efficiency improvement allows an adversary to obtain information about the access pattern. Recent works of Pinkas and Reinman [20], Goodrich and Mitzenmacher [12], and Goodrich et al. [13] investigated the usage of cuckoo hashing to further improve parameters. We will show that cuckoo hashing may potentially lead to a severe reduction in security. In particular, we explicitly construct an adversary that breaks the security of the scheme of [20]. However, when used correctly, cuckoo hashing can result in secure schemes as seen in [12, 13].

In [20], a scheme was constructed with constant local storage, $O(n)$ remote storage, and $O(\log^2 n)$ access overhead. In [12] a scheme using cuckoo hashing was given with similar parameters, as well as an alternative scheme that used $O(n^{\varepsilon})$ local storage with $O(\log n)$ overhead, which has comparable parameters to [13]. Recently, Boneh, Mazieres, and Popa [5] extended the notion of oblivious RAM and described a new scheme with low round complexity, analogous to Goldreich's square-root solution [9] of low round complexity. This new scheme introduces a block size B so that the remote storage has size roughly Bn (storing n elements), the local memory size is an adjustable parameter, for which two possible values $B\sqrt{n}$ and $\sqrt{Bn\log(Bn)}$ were suggested, and the access overhead with these memory sizes is $O(\frac{\sqrt{n}}{B}\log(Bn))$ and $O(\sqrt{\frac{n\log(Bn)}{n}})$,

[1] [10] is a journal version that merged results of [9, 16, 17].

respectively. The key feature is that the amortized round complexity is logarithmic or constant. We give a more detailed review of the most relevant schemes in Section 3.

Another interesting question is that of obtaining schemes with *worst-case* (rather than amortized) overhead. A method demonstrating how to perform worst-case overhead of $O(\log^3 n)$ by de-amortizing [10] was introduced by Ostrovsky-Shoup [18]. This question was also recently studied by Goodrich et al. [14] and Shi et al. [22]. Goodrich et al. [14] constructed two schemes, one of which used $O(n^\epsilon)$ client memory to achieve $O(\log n)$ worst-case overhead, the other used constant client memory and showed how to extend the "square-root" solution of [9, 10] to achieve $O(\sqrt{n}\log^2 n)$ worst-case overhead. Shi et al. [22] constructed a scheme with $O(\log^3 n)$ worst-case overhead with constant client memory. We consider the worst-case overhead measure for our construction in Section 6.

The works of Ajtai [1] and Damgård, Meldgaard, and Nielsen [8] provide a solution to oblivious RAM with information-theoretic security in a restricted model where adversary is not allowed to read memory contents. These solutions result in poly-logarithmic access overhead and operate in a slightly different model which only considers hiding the access pattern and not the data itself. To compare these to existing schemes in the "standard" model of [10], we can encrypt the data (with no additional asymptotic overhead) to hide the data as well. Although it is the case that these converted schemes have worse performance than existing schemes in the standard model, the comparison is unfair as these schemes were created as a trade-off between hiding the data and achieving *information theoretic* hiding of the access pattern.

We also mention the related work of Boneh, Kushilevitz, Ostrovsky and Skeith [4] (with amortized improvements in [6]) that describes a "PIR Writing" scheme in a public-key setting. The scheme allows for private storage and retrieval of public-key encrypted data, but the key difference is that while the communication complexity of the scheme is sub-linear (specifically, $O(\sqrt{n \log n})$) and the round complexity is small, the server must perform a *linear* amount of work on the database. "PIR Writing", allows users (without the private key) to write into the database with hiding the read/write pattern, but the model is more taxing than Oblivious RAM schemes that only require a sub-linear amount of computation for the database. PIR Writing is also considered in the work of Ostrovsky-Shoup [18] in the multi-server model.

Our Contribution. In this paper, we point out a flaw regarding the way hash overflows are dealt with in previous schemes. The idea is as follows: after the end of a reshuffle, the server knows that the data stored at a level can be inserted into the hash-table at that level without an overflow. While this simple observation might not seem very useful, it could actually reveal a significant amount of information.

For a concrete example, when using cuckoo hashing, it is known that no three elements a, b, c have the property that "$h_1(a) = h_1(b) = h_1(c)$ and $h_2(a) = h_2(b) = h_2(c)$", otherwise they could not have been inserted into the table. However, during subsequent probes x, y, z to that hash-table, for elements x, y, z which are actually *not* in the table, with some noticeable probability, we may have "$h_1(x) = h_1(y) = h_1(z)$ and $h_2(x) = h_2(y) = h_2(z)$"; in such a case, the server immediately knows that x, y, z cannot all live inside that table. This can be used to distinguish two sequences of memory accesses, one in which only elements living at this hash-table are read, and one which reads elements that do not live at this level. If the table is of constant size, this lead to a distinguisher with constant success probability! We, and independently [12, 7], discovered a flaw in [20] that is due to this issue. In Section 4, we demonstrate this flaw by constructing an actual adversary that can distinguish between two read/write sequences with constant probability.

Believing that there was a deeper cause to this issue, we further investigated how hash overflows affected other Oblivious RAM schemes. We uncover previously undiscovered flaws that we believe to be the root of the problem and present these in Section 4. We summarize the impact of the hash overflow issue on other existing schemes:

- There is a flaw in the proof of Lemma 5.4.1 in [10]. However, in this case, the flaw has only a negligible

impact on the relevant probabilities and so the overall security of the scheme remains intact. Same holds for [16] (Claim 13). A similar flaw appears also in the proof of Theorem 2 of [20]; in this case, however, the impact is significant and the scheme turns out to be insecure.

- The schemes in [17, 23] also rehash their tables if a bucket overflows; the security proofs in these papers note that this only happens with negligible probability.

- The scheme in [24] does not explicitly specify a hash function because it uses a Bloom filter to determine if an element is in a table before querying it. However, if the Bloom filter returns a false positive with an inverse polynomial probability and a standard hash algorithm such as bucket hashing is used, then there exists an adversary with a non-negligible chance of breaking the scheme. (For a careful analysis of this scheme, see [20].)

- One of the schemes in [11] uses constant local memory and $O(\log n)$ sized stashes for cuckoo hashing. The probability of overflow in this case was shown to be negligible, so the security is not impacted. Two other variants (one with constant local memory and one with $O(n^\epsilon)$ local memory) use a constant size stash for cuckoo hashing. The probability of overflow for these hash tables are an inverse polynomial (where the degree is proportional to the size of the stash) and results in a non-negligible advantage for a distinguisher. We emphasize that we refer to [11] just to give another illustration of the hashing issue; indeed, in a newer version of that work [12], constant sized stashes are no longer used and so there are no security problems.

We then ask the question: Is there a secure oblivious RAM that has (amortized) access overhead $o(\log^2 n)$ with constant local memory? The work of [24] offers a solution with access overhead $O(\log n \log\log n)$ using $O(\sqrt{n})$ local storage, but based on the accurate analysis of [20] and our newly discovered flaws regarding hash overflows, the distinguishing advantage of an adversary may be non-negligible. The recent results of [12, 13] show that it is possible to achieve a secure oblivious RAM with $O(\log n)$ access overhead using $O(n^\varepsilon)$ local storage.

We answer the above question in the affirmative by presenting a scheme with $O(\log^2 n/\log\log n)$ access overhead, while maintaining optimal $O(1)$ local storage and optimal $O(n)$ remote storage. To accomplish this, we observe that the access overhead consists of two types: "scan" overhead and "reshuffle" overhead. In several known schemes, there is an imbalance between these two quantities and, by carefully balancing the size of each level, we can asymptotically reduce the overall access overhead. The starting point for our scheme is the secure scheme of [12] that uses $O(1)$ local memory, a stash of size $O(\log n)$ and has $O(\log^2 n)$ access overhead. We apply two modifications to it: (1) we reduce the scan overhead, and (2) we balance the cost of a scan and a reshuffle by increasing the sizes of the levels, but reducing the overall number of levels. Our construction also uses a mechanism that may be seen as a version of the "shared stash" idea from [13] (namely, one stash that is common to all buffers and accumulates a limited number of elements that cannot be stored in the buffers, due to "unlucky" choice of hash functions) except that, in our case, this stash is virtual and its elements are actually re-inserted into the data-structure. Our scheme is described in Section 5 (some of the previous work on which we rely is described in Section 3).

Finally, because the overhead is *amortized*, processing some queries may take much longer than others. Although these times are known in advance and do not reveal any information, it could have a negative impact on practical applications that require each query to be performed quickly. The question of achieving worst-case overhead for Oblivious RAM was recently considered by Goodrich et al. [14] and Shi et al. [22]. We focus on the case where the client has constant memory. In [14], it is shown how to extend the "square-root" solution of [9, 10] so that it achieves $O(\sqrt{n}\log^2 n)$ worst-case overhead. In Section 6, we describe a technique where we increase the memory size at each level by a constant factor and we schedule the operations

in a way that "spreads out" the amortization. As a warm-up, we first show how this technique can be applied to the original "hierarchical" solution [16, 10] to obtain a scheme with $O(\log^3 n)$ worst-case overhead. Then, we describe the necessary additional modifications for our new scheme to achieve an Oblivious RAM scheme with $O(\log^2 n / \log\log n)$ worst-case overhead using constant client memory, without increasing the asymptotic server memory.

2 Preliminaries

RAM Model. The Random Access Machine (RAM) model consists of a CPU and a memory storage of size n. The CPU executes a program that can access the memory by using $\mathsf{Read}(r)$ and $\mathsf{Write}(r, \text{val})$ operations, where $r \in [n]$ is an index to a memory location. The *access pattern* of the program is the sequence of reads and writes it makes to the memory, including both the data and the memory locations.

In an actual implementation of a RAM, the CPU is a protected local device (possibly probabilistic) that internally has a *small* number of registers, performs basic arithmetic operations, and has access to an external device that simulates the memory (e.g., a physical memory device). We say that a simulation is a secure Oblivious RAM if for any two access patterns in ideal RAM, the corresponding access patterns in the simulation are computationally indistinguishable.

Hash Functions. Throughout this paper, as well as in prior works, a hash function is considered to be a random oracle or a pseudo-random function family $\{F_k\}$. The domain of this hash function is typically taken to be $[n]$ along with some dummy values, and the range of these hash functions are indices of hash-tables of size at most $O(n)$. The key property that we need from these hash functions are that they behave like independent random functions, rather than any particular cryptographic feature such as resistance to finding pre-images. These hash functions are used for the hash-tables (standard hashing with buckets or cuckoo hashing, as described next) in the hierarchical solution.

Cuckoo Hashing. Cuckoo hashing is a hash-table data-structure introduced by Pagh and Rodler [19]. The salient property of cuckoo hashing is that an element x can only live in one of two locations. The scheme uses two hash functions h_1 and h_2 and x can only reside in $h_1(x)$ or $h_2(x)$. When an element is inserted, it is always inserted in the first location, kicking out any previous occupant. The kicked out element is then moved to its other location, possibly kicking out another element. This process continues until no element is kicked out, or it runs too long, in which case new hash functions are chosen and the entire table is rehashed. It was shown that, if the number of elements is at most a constant fraction (say 50%) of the table size, then the probability of requiring a rehash is small enough so that an insert takes $O(1)$ amortized time. Subsequent works, such as [2], have shown how to de-amortize cuckoo hashing.

An important variant of cuckoo hashing is cuckoo hashing with a stash. Introduced by Kirsch et al. [15], a stash of size s is added to the cuckoo hash-table to deal with overflows. The work shows that, if a constant size stash is used, then inserting n elements into a cuckoo hash-table of $\Theta(n)$ size will succeed with all but $O(n^{-s})$ probability. Goodrich and Mitzenmacher [12] showed that, if the size of the table is $m = \Omega(\log^7 n)$ and the stash size is $s = O(\log n)$, then the probability of success (in inserting $m/2$ elements into the table) is all but negligible in n.

We note that although, in general, cuckoo hash-tables are dynamic, for oblivious RAM we only use them in a static manner, i.e. all the elements to be inserted into the table are always known ahead of time.

Bloom Filters. A Bloom Filter [3] is a data-structure to determine set membership. It consists of an array of bit flags, all initially set to zeros, along with a set of hash functions $h_1, \ldots, h_n$. When element x is being

inserted, compute $h_1(x), \ldots, h_n(x)$ and set the bit flags corresponding to those locations to 1. When we want to test whether or not some element x' is present, we compute $h_1(x'), \ldots, h_n(x')$ and check if all bit flags are set to 1. This method of testing can possibly lead to false positives; as we shall see in Section 4, this can lead to insecurity in Oblivious RAM schemes that employ Bloom filters.

3 Overview of Previous Schemes

In this section, we give an overview of some of the previous schemes; specifically, [17, 20, 12]. This serves as a starting point for our scheme (in Section 5). For full details, we refer the reader to the original papers.

Starting from [16, 17], most schemes use Ostrovsky's *hierarchical* data-structure that consists of a sequence of buffers $B_k, B_{k+1}, \ldots, B_L$ of geometrically increasing sizes; i.e., B_i is of size 2^i. Typically $k = O(1)$ (i.e., the first buffer is of constant size) and $L = \log n$ (i.e., the last buffer may contain all n elements), where n is the total number of memory locations.[2] Each buffer is maintained as a hash-table of one of two types. The first type is where B_i is a standard hash-table, associated with a (secret) hash-function h_i. In this case, originated from [16, 17], each entry j is actually a bucket of size b containing all the elements mapped by h_i to j. The second type is where B_i is a table of cuckoo-hashing associated with a (secret) pair of hash-functions $(h_{i,1}, h_{i,2})$. In this case, originated from [20, 11], no bucket is needed.

It should be noted that with a certain (small) probability, the chosen hash functions do not allow for legal storage of the elements into the data-structure; in such a case, the schemes either fail (if this event is sufficiently rare) or simply re-choose the problematic hash-functions. (As we shall see below, those buffers are *static* data-structures and so all the elements to be stored in them are known in advance.) In Section 4 below, we show that this failure event actually leaks information that can be utilized by an attacker. A cuckoo hash-table with stash of carefully chosen parameters takes care of these bad events. Finally, it is also possible to use an hybrid scheme, where for buffers $B_k, \ldots, B_{K-1}$ we use standard hashing while for $B_K, \ldots, B_L$ we use cuckoo-hashing (with stash).

Reading an element. When reading an element from the hierarchical data-structure via a $\mathsf{Read}(r)$ operation, we wish to hide the identity of the buffer from which we read (which yields information on the sequence of operations). Hence, we actually read from *each* of the (at most L) buffers. Specifically, we start by reading the top (smallest) buffer B_k in its entirety; then, for each i, if buffer B_i uses standard hashing, we compute $j = h_i(r)$ and read the entire j-th bucket (b elements) of that buffer. (This is how it was done in the original hierarchical solution of [16, 17].) If buffer B_i uses cuckoo hashing, then we read the pair of elements $h_{i,1}(r)$ and $h_{i,2}(r)$; moreover, if a stash is employed, we read the entire stash corresponding to B_i. If element r exists in more than one buffer B_i, then the insertion method (see below) guarantees that the smallest such i for which B_i contains the element r has the most updated value; so, once element r is found, we search upon dummy locations (by using dummy $\circ\, t$ instead of r, where t is a local counter indicating the number of queries performed already) from subsequent buffers. In addition, at the end of this process, we re-insert element r into the top buffer of the data-structure (see below). This (together with the re-shuffling procedure described below) guarantees that, when executing future $\mathsf{Read}(r)$ operations (with the same r), independent locations will be read from each buffer. Finally, we remark that even if element r was not in any buffer before the $\mathsf{Read}(r)$ operation, it will be inserted into the top buffer.

[2] Each entry of these buffers is of the form (id,x), where id $\in [n] \cup \{\text{dummy}[n]\}$ (indicating which of the original n memory locations is stored here, or that this is a "dummy" element whose sole purpose is to help in hiding the access pattern) and x is a value taken from an appropriate domain F and encrypted (under a secret key).

Insertion. Elements are always inserted into the top buffer B_k. This is done both as a result of a "read" operation, as described above, and in response to an update of entry r in memory, when performing a $\mathsf{Write}(r,\text{val})$ operation. Now, every 2^i insertions, buffer B_i is considered "full"[3] and its content is moved into (the larger) buffer B_{i+1}. More precisely, we do the following: after α insertions (one insertion per each Read and Write operation), where α is divisible by 2^i but not by 2^{i+1}, we move all the elements from buffers $B_k, \ldots, B_i$ (there are less than 2^{i+1} such elements) into buffer B_{i+1} (at such time step, B_{i+1} itself is empty). For this, we pick fresh random hash-functions for B_{i+1} (a function h_{i+1} in the case of standard hashing, and functions $h_{i+1,1}$ and $h_{i+1,2}$ in the case of cuckoo hashing). Finally, to "destroy" the correlation between the new locations of the elements (in B_{i+1}) and their old locations, we use oblivious hashing [16, 17]. The re-shuffling method guarantees the following property, which is crucial in the security analysis of all those schemes: if memory location r is accessed more than once (by either a Read or a Write operation) we never look for it in the same buffer B_i with the same hash-function; in other words, in the time until picking a fresh hash-function for B_i, an item r should not be allowed to come back to B_i (and, of course, if r is found in a previously accessed buffer, then in B_i we search for a random element instead).

A "no duplicates" variant. Next, we describe a variant of the above schemes in which the data-structure never contains more than one copy of any element; i.e., for every $r \in [n]$, there is a unique element of the form (r,value) in the data-structure. This idea comes from [20] and is used there as an optional modification for efficiency. In our scheme (in Section 5), we rely on this variant to obtain some simplifications. In this variant, when performing a $\mathsf{Read}(r)$ operation for reading an element r, we proceed as above except that when finding (r,value) in some buffer B_i, we "remove" it from B_i (and keep just the new version of (r,value) that is inserted into the top). This is done by writing into each location that we read in each buffer; in most places, we just write the same pair (id,value) that we just read (re-encrypted, to hide this fact) and only in the (unique) place that we found the element r, we write a "dummy" value instead. When performing a $\mathsf{Write}(r,\text{value})$ operation, we start by performing a read operation with identifier r (where, as above, if we find a pair (r,value') it is removed) except that rather than re-inserting the read value (r,value') into the top buffer, we insert the new value (r,value). The re-shuffling mechanism works as before (in fact, re-shuffling is somewhat simpler, when there are no duplicates, as we do not need to worry about possible conflicts).

4 Flaws in Previous Schemes

In this section, we point out a flaw that occurs in many previous schemes and comes from the re-choosing of hash functions in the case of insertion failure. (We note that the details of this section are not needed for following our own construction in Section 5.) If the event of overflow occurs with non-negligible probability, the act of re-choosing new hash functions can reveal significant information about the data that is stored inside a particular level. The act of choosing new hash functions appears in several previous works, such as [16, 10, 23, 24, 20, 11]. We take a deeper look into the ways information can be leaked due to these hash functions.

To succinctly summarize the problem, we describe a general method for distinguishing access patterns, based on hash overflows. Consider two access sequences Q_0 and Q_1: Both Q_0 and Q_1 start off by inserting enough elements to trigger re-shuffles so that only levels i and $i+1$ are non-empty and contain disjoint, distinct elements. Q_0 continues with reads of distinct elements in level i, and Q_1 continues with reads of distinct elements in level $i+1$.

[3] B_i may not necessarily be full, but to prevent the possibility of distinguishing, for example, the extreme cases one where all the insertions are distinct and the other case where all the insertions are identical, we need to act the same in all cases.

In both sequences, the server can observe the sequence of probes caused by the hash functions. In the former case, because all elements live within level i, the sequence induces a pattern that is consistent with a non-overflowing hash-table. In the latter case, if the hash function is random, the sequence of probes will induce a completely random pattern, which is one that may not be consistent with a non-overflowing hash-table. If this event occurs with inverse polynomial probability, an adversary can distinguish between these two access patterns with a non-negligible advantage. We now take a look at how this affects existing schemes.

Bucket Hashing. Bucket hashing inherently has a chance of failure on insert due to a bucket overflowing, and this probability depends on the size of the buckets in the hash table. The works of [10, 23, 24] rely on bucket hashing as the primary data structure in the hierarchy. In these schemes, it is suggested that in the case of failure, new hash functions are selected and all the elements are re-hashed into a new table.

Cuckoo Hashing. In (ordinary) cuckoo hashing, the probability of failure is also inverse polynomial [19]. The most obvious way an insert can fail is if the new element hashes to the exact same two locations as two previous elements, since 3 elements cannot fit into two locations. In such a case, new hash functions are selected and all the elements are re-hashed into a new table, similar to the case of bucket hashing. Based on this fact, we give explicit examples of distinguishers for the scheme of [20], and we point out that the cuckoo hashing collisions remain problematic even if a small constant-sized stash is used (such as in one of the schemes in [11]).

Distinguisher example. For the sake of consistency, it is assumed that we may only read elements that have previously been inserted. The examples provided will abide by this rule. We let numbers denote elements inserted, letters denote dummy elements, as prescribed by the scheme in [20], and * denote "don't care" elements as prescribed by their scheme. In the following examples, it does not matter how long the query sequences are, and we only state the initial subsequence necessary to make the distinguishing go through. We write $\mathsf{Write}(r_1, \ldots, r_n)$ as shorthand for writing to locations r_1 through r_n with arbitrary values written.

The construction in [20] allows the server to know where the empty slots of the cuckoo hash-table are. As a warm-up example demonstrating how this can leak information, we show that this actually allows the server to distinguish between the two sequences $Q_0 = (\mathsf{Write}(1,2,3,4,5,6), \mathsf{Read}(1))$ and $Q_1 = (\mathsf{Write}(1,2,3,4,5,6), \mathsf{Read}(5))$ with probability $1/4$. After the sequence of initial writes, the buffers will look like this:

Level 1: Empty

Level 2: $5, 6, *, *, A, B, C, D$ stored in a cuckoo hash-table of size 16

Level 3: $1, 2, 3, 4, *, *, *, *, E, F, G, H, I, J, K, L$ stored in a cuckoo table of size 32

In Q_0, the operation $\mathsf{Read}(1)$ will scan the entire level 1, then two random hash locations on level 2, then the places where 1 hashes to on level 3. In Q_1, the operation $\mathsf{Read}(5)$ will read the entire level 1, then the places where 5 hashes to on level 2, then the places where dummy E hashes to on level 3.

Thus, if the two locations that 1 hashes to on level 2 are empty, then the distinguisher outputs Q_0. Since we are using random hash functions, there is a $1/2$ chance each of hitting an empty spot, giving us a $1/4$ distinguishing advantage.

Next, we give an example that addresses our main concern that has to deal with the "collision-free" property of a cuckoo hash-table. The trick to distinguishing is that reading three elements that exist in a level will *never* collide (since they must occupy 3 distinct locations), but accessing three random elements gives a small probability of them all hashing in to the same two addresses in the hash table. In this example,

the server can distinguish between the two sequences $Q_0 = (\mathsf{Write}(1, 2, ..., 12), \mathsf{Read}(1, 2, 3))$ and $Q_1 = (\mathsf{Write}(1, 2, ..., 12), \mathsf{Read}(9, 10, 11))$ with a $1/32^4$ advantage. After the sequence of initial writes, the buffers will look like this:

Level 1: Empty

Level 2: Empty

Level 3: $9, 10, 11, 12, *, *, *, *, A, B, C, D, E, F, G, H$ stored in a table of size 32

Level 4: $1, 2, 3, 4, 5, 6, 7, 8, *, *, *, *, *, *, *, *,$
$I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X$ stored in a table of size 64

In Q_0: The operation $\mathsf{Read}(1)$ will read all of level 1, two random locations in level 2, two random locations in level 3, and the two places it hashes to in level 4. It will be found and written back to level 1. The operation $\mathsf{Read}(2)$ will read all of level 1, two random locations in level 2, two random locations in level 3, and the two places it hashes to in level 4. It will be found and written back to level 1. A reshuffle from level 1 down to level 2 now occurs, the hash functions at level 2 are now re-picked to be new random functions. This does not affect level 3. The operation $\mathsf{Read}(3)$ will read all of level 1, two random locations in level 2, two random locations in level 3, and the two places it hashes to in level 4. It will be found and written back to level 1.

In Q_1: The operation $\mathsf{Read}(9)$ will read all of level 1, two random locations in level 2, the two places it hashes to in level 3, and the two places dummy I hashes to in level 4. It will be found and written back to level 1. The operation $\mathsf{Read}(10)$ will read all of level 1, two random locations in level 2, the two places it hashes to in level 3, and the two places dummy J hashes to in level 4. It will be found and written back to level 1. A reshuffle from level 1 down to level 2 now occurs, the hash functions at level 2 are now re-picked to be new random functions. This does not affect level 3. The operation $\mathsf{Read}(11)$ will read all of level 1, two random locations in level 2, the two places it hashes to in level 3, and the two places dummy K hashes to in level 4. It will be found and written back to level 1.

In Q_1, the two places that the sequence of reads probe in level 3 will never be the exact same two places every time. This is because the hash pattern must induce a valid cuckoo pattern, or "hash graph". However, in Q_0, because the locations are random, there is a $1/32^2$ probability that the two locations probed by the second read will match the two locations probed by the first read, and similarly another $1/32^2$ probability for the last read. This is an example of how the "triple hit" problem stated in the introduction can be seen in practice.

The subtle issue can be found in the simulation proof of [20]. The simulated probe sequence only probes things that are either a real element in level i, or a dummy element in level i. However, in a real execution of the protocol, one might probe something that exists in a lower level (or does not exist at all in the database). In the worst case (as in the example above), none of the 2^i elements probed in between reshuffles of level i exist in level i. This is as if one were to build a brand new cuckoo hash-table with the given hash functions, which is known to fail with probability $O(1/n)$. Thus, there exists a list of instructions such that, with probability $O(1/n)$ in the real execution, the sequence of probes on level i, as observed by the server, induces an invalid cuckoo hash graph. The simulator described in [20] will never generate a sequence of probes that induces an invalid cuckoo hash graph, because their simulated probes are only to elements in the cuckoo hash-table.

Impact on Existing Schemes. We also examined prior schemes, and have found that some suffer from similar flaws as mentioned above. Namely, if a hash table overflows and new hash functions are chosen, this could potentially leak information that would allow an adversary to distinguish between two sequences of reads and writes.

In [10], standard hash-tables are used with buckets of size $O(\log n)$. In Lemma 5.4.1 of [10], the proof states that the only values hashed at level i (between reshuffles) will be ones corresponding to either elements living inside level i or dummy elements. This is not true because if an element x is sitting inside level $i+1$ or deeper, then we still must hash x at level i and access the corresponding bucket. After some number of queries, if the hashes result in a bucket being accessed more times than the total capacity of the bucket, it will be immediately revealed that at least one of these previous queries was not found at level i. It is still true that all the elements being hashed at level i will be unique, so under the condition that these hashes do not overflow a bucket, the accesses made are oblivious. A similar flaw can be found in Theorem 2 of [20] and Claim 13 of [16].

In [16, 17], in the case of overflow, the simulation is aborted. In [10] it is suggested that in the case of an overflow, which happens with negligible chance, a new hash function is chosen. In these three works, aborting or choosing new hash functions results in an information leakage, because the server knows that the data stored in a level is consistent with the hash function. However, the abort or rehash event occurs with only a negligible chance so the security claims remain intact. This is not the case with [20] where, in fact, this flaw leads to a distinguisher.

The concept of choosing a new hash function is repeated in the subsequent work of [23]. The security of this scheme is not impacted as, in fact, this overflow only happens with negligible probability (see Lemma 5.2 below).

In [15], it is shown that a cuckoo hash-table with constant size stash (which was used in [11]) has a probability of overflow of $O(1/m^s)$ where s is the size of the stash and m is the size of a level. This amount can be made to be smaller than any fixed polynomial by increasing the size of s, though it is still not negligible. It is shown in [11] that a cuckoo hash-table with logarithmic size stash will overflow with only negligible probability, assuming that the table is of size $\Omega(\log^7 n)$ and, hence, suitable for use for larger levels. In the updated version [12], constant sized stashes are no longer used, thus resulting in schemes that only have a negligible chance of overflow.

A problem related to hash overflow occurs in the work of [24], where a Bloom filter is used to check whether or not an element is in the hash table before searching for it. This appears to mitigate the problems above, but the issue is that a Bloom filter has some probability for generating false positives. If the Bloom filter generates enough false positives on elements not present in the hash-table, the problem appears again. In order to make the probability negligible, the size of the Bloom filter must be super-logarithmic, thereby increasing the overall storage and overhead

5 Our Scheme

In this section, we present our oblivious RAM scheme, where we both fix the leakage of information and obtain improved efficiency. The high level ideas behind our scheme are as follows: (1) we eliminate the stash by re-inserting its elements back into the data-structure,[4] and (2) we balance between the cost of reads and the cost of writes. Technically, we use as a starting point the constant local memory scheme of [12] whose top buffer B_k is sufficiently large and the first cuckoo-hash level B_{K+1} is also sufficiently large; these choices make the probability of bad events negligible. Next, we provide a detailed description of our scheme.

Our starting point is the hybrid-scheme with no duplicates, as described in Section 3. We choose $k = \log\log n + 1$ and $K = 7\log\log n$, which means that standard hashing is used for the top $O(\log\log n)$ buffers whose sizes are between $2\log n$ and $\log^7 n$; the bucket size for these buffers is set to $b = \log n$ (the choice of parameters is made to deal with attacks of the type described in Section 4). For the larger buffers, we employ cuckoo hashing; we use the version with stash of size $s = \log n$ (as in [12]).

[4] This is similar to the notion of a shared stash, introduced by [13]; see below, for an explanation of the difference.

The first modification we make is that, rather than actually keeping the stash at each level, at the end of the re-shuffling stage (as in [12]), there will be a single stash of size s. The concept of a shared stash was introduced in [13] subsequently used in [12] in the construction of Oblivious RAM using $O(n^\epsilon)$ local storage. In our case, we have constant local storage, so instead of keeping an explicit shared stash, we "virtualize" the stash by re-inserting it into the top buffer. Note that at the end of re-shuffling, the top buffer is empty and we make exactly s insertions, independent of the actual content of the stash.

The second ingredient of our solution, refers to balancing the cuckoo hashing levels. At level $\ell = K + i$, we actually have $t = \log n$ buffers (rather than just one) $B_\ell^1, \ldots, B_\ell^t$ and each of them is of size $t^{i-1} \cdot 2^{K+i}$; in particular, this means that a buffer at each of these levels is larger by a factor of $2t$ than the buffers at the previous level and so, in each re-shuffling, one such buffer can accommodate the elements of all buffers in all previous levels. In order to keep notation consistent, we define $B_\ell^1 = B_\ell$ to be the only buffer for levels between k and K (we may refer to B_ℓ^j as a "sub-buffer" to emphasize that it is one of a set of buffers at the same level ℓ). The operations on the modified data-structure are now implemented as follows. When reading an element r via a $\mathsf{Read}(r)$ operation, we look for it at each cuckoo-hashing level ℓ in all t sub-buffers in reverse order [5] – starting from B_ℓ^t down to B_ℓ^1 (the top levels that do not use cuckoo hashing are read exactly as before). When we perform a re-shuffling into level ℓ for the first time, we do it into sub-buffer B_ℓ^1; in the next time, we will use B_ℓ^2 and so on; only after using all t sub-buffers, the next re-shuffling will be done into level $\ell + 1$. Having t sub-buffers at each levels, allows us to reduce the total number of levels L to $L = O(\log n / \log\log n)$.

A detailed description of our algorithm for performing Read and Write operations appears in Appendix A. We refer the reader to [12] for a detailed description of the (unmodified version) of re-shuffling.

Complexity analysis. We first analyze the *physical size* and the *capacity* of each level, by which we mean the number of elements it may contain before we consider it to be "full".[6] The top level B_k is treated as an array and has size and capacity equal to $2\log n$. The buffers using standard hash with buckets occupy levels $k+1$ through $K-1$. The capacity of each one is twice that of the level before it, and thus for buffer B_{k+i} it is equal to $2^{i+1}\log n$. The size, however is $|B_{k+i}| = 2^{i+1}\log^2 n$ due to the fact that we have buckets of size $O(\log n)$. For the cuckoo hash-tables, the capacity of a single sub-buffer B_K^j, at level K, is $\log^7 n$ and it has size $2\log^7 n$. The total size and capacity at level K is $\log n$ times larger due to the fact that we keep $\log n$ of these buffers. For subsequent levels, we have sub-buffers of size $|B_{K+i}| = 2\log^{8+i} n$ for $i = 1, \ldots, L-K$ and capacities of half that amount. Because we wish to store a total of n elements in the largest level, L, we must have $\log^{8+L-K} n = \Theta(n)$. Solving for L, we have $L = O(\log n / \log\log n)$. Hence, the total number of elements stored remotely is:

$$\sum_{i=0}^{K-k-1} 2^{i+1}\log n + \sum_{i=0}^{L-K} 2\log^{8+i} n = O(\log^7 n) + O(n)$$

which is $O(n)$.

In addition, we locally store only a constant number of elements (typically, 1 or 2) during reading, and $O(1)$ elements also suffice for reshuffling [12].

Next, we analyze the overhead of our scheme. The idea is that each read now costs more (compared to the basic hybrid scheme) but we gain by reducing the cost of re-shuffling (i.e., the sort). Namely, the number of levels is now $O(\log n / \log\log n)$ (since, between cuckoo hashing levels, the size grows by an $O(\log n)$ factor) but at each level we read from each of the t buffers and so the total cost for a read is $O(\log^2 n / \log\log n)$

[5] This is important for obtaining security, in particular in the proof of Lemma 5.1.

[6] The distinction between size and capacity is important when using cuckoo-hashing, whose performance is guaranteed only for buffers whose capacity is not more than, say, 50% of their size.

(dominated by the cuckoo hashing buffers). On the other hand, re-shuffling is done less frequently (since, for each level, we wait until all t sub-buffers are utilized before re-shuffling) and so the amortized cost of re-shuffling is also $O(\log^2 n/\log\log n)$.

In detail, we separate the buffers into three categories: the top buffer B_k, the standard hash buffers $B_{k+1},\ldots,B_{K-1}$, and the cuckoo hash buffers $B_K,\ldots,B_L$. During a read, the entire top buffer is scanned twice, which requires $2\log n$ operations. When scanning the buffers, the standard hash buffers each have buckets of size b, which results in $b(K-k)=O(\log n\log\log n)$ total elements read. Finally, in the cuckoo hash buffers, two elements are read (and then written) from each buffer giving a total of $4(L-K)\log n = O(\log n\cdot\log n/\log\log n)$ elements. Although the cuckoo hash-tables have stashes, the stashes have already been moved to the top level (and possibly reshuffled down), and do not need to be read.

Recall that a reshuffle step can be done with $O(m\log m)$ accesses, where m is the number of elements involved in the shuffle (this includes the re-insertion of the stash whose size is $s=O(\log n)\le O(m)$ to the top buffer, after every reshuffle). Note that, due to the growth of the levels, the number of elements in levels $k,\ldots,i$ is at most twice the number of elements contained in a sub-buffer B_i^j. Also, the act of inserting a single element at the top buffer can be viewed as also inserting "phantom" elements at every level, and these phantom elements are identified during a reshuffle. By this observation, it takes $O(|B_i^j|-\log n)=O(|B_i^j|)$ steps for B_i to become full. Thus, the amortized cost of reshuffling is:

$$\begin{aligned}\sum_{i=k}^{L}\frac{O(|B_i|\log|B_i|)}{O(|B_i|)} &= O(\log\log n)\\ &+ \sum_{i=1}^{K-k-1} O\left(i+2\log\log n\right)\\ &+ \sum_{i=1}^{L-K} O\left((8+i)\log\log n\right).\end{aligned}$$

Since $K-k-1$ is $O(\log\log n)$ and $L-K$ is $O(\log n/\log\log n)$, this gives

$$\begin{aligned}O(\log\log n) &+ O((\log\log n)^2)\\ &+ O((\log n/\log\log n)^2\log\log n)\\ &= O(\log^2 n/\log\log n)\end{aligned}$$

total amortized overhead for reshuffling.

Security. To argue the security of the scheme, we begin with a lemma that is common to previous schemes:

Lemma 5.1. *Except for the top level, no index id will be searched upon twice in any sub-buffer B_i^j before this sub-buffer is reshuffled (i.e. elements are shuffled into or out of it). In other words, all searches to B_i^j between reshuffles are from unique indices (which may collide when hashed).*

Proof. Suppose index id is searched upon (i.e. id is the value to be hashed) for sub-buffer B_i^j. Either id is an actual index or it is a dummy index. In the case of id being a dummy index, the lemma immediately follows from the fact that dummy indices are of the form $dummy\circ t$ where t is a query counter, and subsequent queries will have ever-increasing values of t (either the hierarchy is designed with a query bound $t<T$, or we can add polynomially many new levels at the bottom of the hierarchy). On the other hand, if id is an actual index, whether or not it is found in the RAM is of no consequence: it will be inserted into the top level B_k. All subsequent queries upon id will result in it being found in a "younger" buffer or sub-buffer, causing

dummy locations to be searched upon in the remaining buffers, which includes B_i^j. The only way id can be present in B_i^j or a deeper buffer again is if id is shuffled into B_i^j or the entirety of B_i is full and all $B_i^{j'}$ are shuffled out to a bigger level. Either way, B_i^j will have been reshuffled and a new hash function or functions will be chosen for it. □ □

Next, we show, using [12], that the probability that our hash tables ever overflow is negligible.

Lemma 5.2. *Assuming the hash functions are perfectly random, a reshuffle into sub-buffer B_i^j succeeds with all but negligible probability.*

Proof. First, observe that a reshuffle into sub-buffer B_i^j involves at most m elements being shuffled into a hash table of capacity m. For the lower levels $B_{k+1}, \ldots, B_{K-1}$, there will be m buckets each holding up to $b = \log n$ elements, and for sub-buffers of $B_K, \ldots, B_L$ there will be cuckoo hash-tables of size $2m$ each and a stash of size $\log n$.

In the case of standard hashing, we consider the probability that more than b elements end up in any bucket when we throw m balls into m bins. The probability that there are exactly i elements in any bucket can be bounded as:

$$\begin{aligned}\Pr[\text{bucket has } i \text{ elements}] &= \binom{m}{i}\frac{1}{m^i}\left(1-\frac{1}{m}\right)^{m-i} i \\ &\leq \left(\frac{me}{i}\right)^i \frac{1}{m^i} \\ &= \left(\frac{e}{i}\right)^i.\end{aligned}$$

Summing over all $i = b, \ldots, m$ we get $\Pr[\text{Bin has at least } i \text{ balls}] \leq \left(\frac{e}{b}\right)^b \frac{1}{1-e/b}$. Taking the Union Bound over all m bins, the probability that any bin overflows is bounded by $\left(\frac{e}{b}\right)^b \frac{m}{1-e/b}$ which is negligible in our case, where $b = \log n$ and $2\log n \leq m \leq \log^7 n$.

It is shown in [11, Appendix C] that for cuckoo hash-tables of size m with a stash of size s, as long as $m = \Omega(\log^7 n)$ and $s = O(\log n)$, the probability of overflow is $m^{-\Omega(s)}$ which is negligible in n. These restrictions on m, s hold for all sub-buffers of levels $B_K, \ldots, B_L$. □ □

Theorem 5.3. *Assume one-way functions exist. Then, there exists a PPT simulator S such that, for all $t > 0$ and for all read/write sequences of length t, the distribution of $S(t)$ is computationally indistinguishable from the distribution of executions of our balanced ORAM protocol on any request sequence of length t.*

Proof. We describe a simulator that generates a view of the server that is (computationally) indistinguishable from that of a real execution on any sequence of accesses. First, by the pseudorandomness of the hash functions, we can simulate these using random functions with domain equal to $[n]$ along with all possible dummy values. We can also simulate encrypted data elements by encryptions of zero, by the semantic security of the underlying encryption scheme.

We observe that both a $\mathsf{Read}(r)$ and a $\mathsf{Write}(r,\text{val})$ operations will begin with the same reading step, followed by the same insertion step, causing exactly one new element to be inserted at the top level buffer B_k. Accesses to B_k will always be oblivious, as B_k is always fully scanned. We show how to simulate the view of the server for these operations after we investigate what reshuffling does. We now turn our attention to what happens in sub-buffer B_i^j.

First, observe that when elements are shuffled into B_i^j, brand new hash functions are chosen thereby breaking any possible previous dependencies. In addition, by the oblivious hashing of [12], the server does

not know the locations of the elements in B_i^j, be it in a standard hash-table, cuckoo hash-table, or the stash. Therefore, to simulate the server's view of B_i^j, we only need to consider what happens between two reshuffles.

Let R indicate the set of elements that was inserted into B_i^j (and possibly its stash). Let R' be another arbitrary set of $|R|$ elements, and let h (or h_1, h_2) denote the hash function used for B_i^j during this reshuffle. Recall that because we are treating hash functions as truly random functions, we know that h is sampled uniformly from the set of random functions consistent with R, i.e. inserting the elements of R into the hash-table does not cause an overflow. Let $\mathcal{H}$ denote this distribution, and let $\mathcal{H}'$ denote the uniform distribution of random functions that are consistent with R'. By Lemma 5.2, the statistical distance between $\mathcal{H}$ and $\mathcal{U}$, the uniform distribution of random functions, is negligible, as is the statistical distance between $\mathcal{H}'$ and $\mathcal{U}$.

During a sequence of queries to B_i^j, all elements accessed will be unique by Lemma 5.1. The server only sees a sequence of probes of distinct elements from some sequence of elements S. Let S' be an arbitrary set of $|S|$ distinct elements and consider the following distributions: $\{h(S)|h \leftarrow \mathcal{H}\}$, $\{h(S)|h \leftarrow \mathcal{U}\}$, $\{h(S')|h \leftarrow \mathcal{U}\}$, $\{h(S')|h \leftarrow \mathcal{H}'\}$. The statistical distance between each neighboring pair of

distributions is negligible, and therefore the distance between the first and last is negligible. As the last distribution is for arbitrary $\mathcal{H}'$ and S', we can simulate the server's view by choosing an arbitrary sequence of operations. □ □

6 From Amortized Overhead to Worst-Case Overhead

In our construction above, the client may need to perform a large amount of work during some reshuffle operations (up to $\Omega(n)$ work when handling the very lowest level), although the overall amortized overhead per query is only $O(\log^2 n/\log\log n)$. For certain applications, we may want to guarantee that every query – even those that induce a reshuffle – has a small bounded amount of overhead. Such a guarantee was first introduced in Ostrovsky and Shoup [18] where they described a solution with $O(\log^3 n)$ worst-case overhead using constant client memory. Recently, Goodrich et al. [14] and Shi et al. [22] revisited the scenario of worst-case Oblivious RAM. Goodrich et al. [14] constructed two schemes, one of which used $O(n^\epsilon)$ client memory to achieve $O(\log n)$ worst-case overhead, the other used constant client memory and showed how to extend the "square-root" solution of [9, 10] to achieve $O(\sqrt{n}\log^2 n)$ worst-case overhead. Shi et al. [22] give a construction of an Oblivious RAM with $O(\log^3 n)$ worst-case overhead with constant client memory.

Both the Ostrovsky-Shoup [18] and the Goodrich et al. [14] protocols made use of doubling up the buffer and labeling each as "current" or "previous" and switching between the two. In this section, we describe a method of tripling each buffer and switching between them in order to spread out the amount of work required for the reshuffling. As a warm-up, we show how to convert the scheme of [10] to have worst-case $O(\log^3 n)$ query overhead, and then go on to show how to extend our construction into a scheme that has $O(\log^2 n/\log\log n)$ *worst-case* overhead per query, while increasing the work and the size of the storage required by the server by a constant factor.

6.1 Warm-up Worst-Case Construction.

As a warm-up, we first describe a method, similar to what was originally suggested by Ostrovsky-Shoup [18], to convert the Goldreich-Ostrovsky oblivious RAM scheme [10] so that it will have $O(\log^3 n)$ overhead in the worst-case rather than $O(\log^3 n)$ amortized overhead. Converting our own scheme will proceed in a similar manner, though with additional ingredients (see Section 6.2 below). Recall that the scheme of [10] can simply be viewed as the lower levels of our construction where no cuckoo hashing is occurring and no sub-buffers are being used. That is, each level $i = 1, \ldots, O(\log n)$ is a standard bucketed hash table B_i with 2^i buckets of $O(\log n)$ elements each. Also recall that, in the scheme of [10], they do not wipe out an element when it is

found, but merely copy it to the top buffer. When two identical memory addresses are being shuffled into the same level, the older duplicate is removed. We make the following modifications to the scheme of [10]:

Triple Each Level. Each level i will now have three buffers (hash tables) B_i, B_i' and B_i''. These three buffers are identical in size and structure to the original buffers. They will be marked in a rotating fashion as "active", "inactive", and "output". Searching for an element on level i will involve searching first the active and then the inactive buffer.

Add Working Memory. Each level i will now have permanently allocated working memory (necessary to perform a shuffle into level i). Although the result of the shuffle will eventually be written from the working memory to the output buffer, we define it as a separate entity from the buffers because the working memory may in fact be larger than a buffer. Recall, that in [10], this working memory is of size $5 \log n \cdot 2^i$, for level i, which is only a constant times larger than B_i. Thus, the overall asymptotic storage complexity is not affected by permanent working memory being allocated.

Smearing the Reshuffle. When the active buffer in a level i is full, i.e. when it is the time for it to be shuffled into the next level, it is not immediately shuffled (as this may take more time than we can afford for satisfying the desired worst-case guarantee) but instead it is marked "inactive". The "inactive" buffer on level i should by now be wiped clean and will be labeled "active". Instead of immediately performing the entire shuffling, for every subsequent query, the client performs a fixed number of steps, $\kappa = O(\log^2 n)$, of the oblivious sorting algorithm; by this, the newly-labeled "inactive" buffer on level i, call it N_i, and the "active" buffer in level $i+1$, call it A_{i+1} are slowly (over $2^i/\kappa$ queries) copied into the working memory of level $i+1$, call it W_{i+1}, where they are shuffled together. To clarify, over the course of these $2^i/\kappa$ queries the following steps are performed:

1. The elements from N_i and A_{i+1} are copied into W_{i+1}.
2. The oblivious sort is performed on W_{i+1} while the buffers N_i and A_{i+1} remain available to be searched upon.
3. The oblivious sort finishes by coping the result from W_{i+1} to the "output" buffer at level $i+1$.

Upon completion of the shuffle (after $2^i/\kappa$ queries for level i), the "output" buffer is now marked as the "active" buffer for level $i+1$, the "inactive" buffer from level i is wiped clean and labeled "active", and the "active" buffer from level $i+1$ is wiped clean and labeled "output".

It is important to note that κ, the number of steps of the oblivious sort we perform per client query, is fixed and independent of the data. Also note that the oblivious sorting algorithm occurs entirely within the working memory – the newly inactive buffer at level i and the active buffer at level $i+1$ remain intact and are available to be searched upon. Indeed, the only activity in these two buffers are from client query lookups and data being copied into working memory. Only until the oblivious sorting is finished do we clear the contents of N_i and A_{i+1}.

To see why the modification works, we need to show that shuffles do not overlap each other. The only danger is wiping out an inactive buffer that is not yet finished shuffling. However, we observe that this does not cause a problem because the shuffle at level i involves shuffling $O(2^i \log n)$ elements which requires $O((2^i \log n) \log(2^i \log n))$ steps which is $O(2^i \log^2 n)$ since $i \leq \log n$. Since we perform $\kappa = O(\log^2 n)$ steps per query, the shuffle will be over once 2^i queries are processed at level i, i.e. exactly the rate at which level i fills up.

How much overhead do these modifications incur? On top of the regular query overhead (which takes $O(\log^2 n)$ time), we perform $O(\log^2 n)$ additional steps per level on the shuffle, and there are at most $\log n$ levels that need to be shuffled, thus giving a total of $O(\log^3 n)$ overhead per query. We have thus transformed the $O(\log^3 n)$ amortized overhead of the original scheme into $O(\log^3 n)$ worst-case overhead.

6.2 Worst-Case Construction For Main Scheme.

We now sketch the modifications needed to make our scheme have $O(\log^2 n/\log\log n)$ worst-case overhead. We begin by modifying our scheme so that it does not cascade when shuffling (i.e. when need to shuffle level i, only level i is shuffled into the next level $i+1$, rather than shuffling all levels above i into level $i+1$). In addition, the scheme is modified such that it does not wipe out an element upon finding it, instead duplicates are only removed when they are being shuffled into the same sub-buffer. Note that the latter modification causes an unexpected issue that did not occur in the scheme of [10]. Namely, consider the following scenario: An element is not wiped out and ends up in the stash of a very big level $i \approx L$, a newer copy of the same element is present in a middle level somewhere. Because we re-insert the stash into the hierarchy, the old version of the element will appear at a *higher* level than the newer version, thus causing a conflict. To handle this issue, we add a tag to each element that indicates the lowest level and sub-buffer it has appeared at. As an element is shuffled into level i, sub-buffer j, the tag is updated via the rule $tag' = max(tag, (i,j))$ based on the order in which we search levels and sub-buffers, i.e. $(i,j) \leq (i',j')$ if $i < i'$ or $i = i'$ and $j \geq j'$, which is not the usual lexicographical ordering. When an element is stashed, the tag is *not* updated so as to keep track of the true age of this element. When searching for an element, v, if it is found we keep track of the tag, but continue searching for v up to a point: Let tag be the minimum one. Of all tags of v found so far, before we search a level and sub-buffer, we first check that it is less than tag. If so, then we search for v, otherwise we stop and search the remaining levels and sub-buffers for random elements. The value that is output is the v corresponding to the minimal tag.

To ensure that our main construction will have a worst-case guarantee, we begin by making the modifications as described above to our sub-buffers except that now the client performs fewer steps ($O(\log n)$ for the lower levels $i = K, \ldots, L$) of the shuffling algorithm. However, we now need to deal with another ingredient where our scheme differs from the Goldreich-Ostrovsky [10] scheme: there may be up to $L - K + 1 = O(\log n/\log\log n)$ active stashes sitting at the lower levels that need to be re-inserted to the top; dealing with all these insertions immediately will violate the desired worst-case bound.

To handle this, we add a "stash backlog" at the very top level that can hold up to $O(\log^2 n/\log\log n)$ elements, i.e. all the elements in all the active stashes for each level $i = K, \ldots, L$. Now, when the client wants to execute a query, the client scans the entire stash backlog in addition to the topmost level. After each query is handled, the client removes one element from the stash backlog and inserts it into the main hierarchy. It is important to note that the backlog will never overflow because it will be emptied within $O(\log^2 n/\log\log n)$ queries, whereas it will only be filled when an active stash is being shuffled into (which happens at intervals of at least $O(\log^7 n)$ steps).

To see why this scheme remains secure, we need to justify why Lemma 5.1 applies here as well. Consider any configuration where some value v may appear several times in the hierarchy with different tags. As a thought experiment, we call each sub-buffer v-*searched* if v was searched for in this sub-buffer (between shuffles of that sub-buffer). We must make sure that we never search for v in a v-searched sub-buffer. We also call a sub-buffer v-*safe* if the minimal v tag is smaller than or equal to the index of that sub-buffer (again, the ordering is the search order, not lexicographical). The point is that v will not be searched upon in any v-safe sub-buffer, due to our new search rules. As long as all v-searched buffers are v-safe, then we will never search for v twice in a sub-buffer.

We argue that this holds by induction. Initially, all sub-buffers are labeled with neither. We show that when (1) v is read or written, (2) a copy of v is being shuffled into a new level, or (3) v is stashed with an old tag back to the top, the invariant that all v-searched sub-buffers are also v-safe remains true. In case (1), when v is read or written, it will be re-inserted at the top with a fresh tag which will be the minimal tag. Thus, all levels and sub-buffers in the hierarchy will now be v-safe (and many of them will also now be v-searched). In case (2), if the copy of v being shuffled is not the one with the smallest tag, the minimal tag

remains the same and thus the set of v-safe sub-buffers does not change. If the v being shuffled is the one with the minimal tag, the tag will be updated to the new level and sub-buffer, thus possibly reducing the number of v-safe sub-buffers. However, the only sub-buffers that will be affected are the ones that were on the same level as the v that was shuffled since the updated tag is only one level deeper. Therefore, these sub-buffers must have also been shuffled out of and emptied in the same shuffle that moved v, which means that they are no longer v-searched (if they previously were). Finally, if after a shuffle a copy of v ends up in the stash, as in case (3), then because the tag is not updated, the minimal tag for v must remain the same and therefore the set of v-safe sub-buffers remains unchanged. In all three cases, the v-searched sub-buffers remain v-safe sub-buffers finishing the induction.

We now observe the total worst-case overhead in the Oblivious RAM scheme incurred by these modifications. From our analysis in Section 5, the client takes $O(\log^2 n/\log\log n)$ time to perform a read or write query. The client now also perform up to $O(\log n)$ shuffle steps for each of the $O(\log n/\log\log n)$ levels and read in the stash backlog; both of which take $O(\log^2 n/\log\log n)$ time. Thus overall, the worst-case overhead for our modified scheme is $O(\log^2 n/\log\log n)$.

Finally, we consider the amount of server storage. We triple each sub-buffer in our construction, as well as add a new "stash backlog" of size $O(\log^2 n/\log\log n)$. Furthermore, as in [12], the amount of permanently allocated working memory per level is on the order of the size of a sub-buffer for that level. Therefore, the overall storage complexity in our worst-case Oblivious RAM is only a constant factor worse than our amortized Oblivious RAM.

7 Acknowledgments

We thank the anonymous reviewers of CRYPTO 2011 and SODA 2012 for their helpful comments.

References

[1] Miklós Ajtai. Oblivious RAMs without cryptographic assumptions. *STOC* 2010: 181-190.

[2] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results. *ICALP* 2009: 107-118.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[4] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E. Skeith III. Public Key Encryption That Allows PIR Queries. *CRYPTO* 2007: 50-67.

[5] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote Oblivious Storage: Making Oblivious RAM Practical. *CSAIL Technical Report*: MIT-CSAIL-TR-2011-018, 2011.

[6] Nishanth Chandran, Rafail Ostrovsky, William E. Skeith III. Public-Key Encryption with Efficient Amortized Updates. *SCN* 2010: 17-35.

[7] Bogdan Carbunar and Radu Sion. Write Once Read Many Oblivious RAM. *Technical Report* 2011. `http://www.cs.sunysb.edu/~sion/research/sion2011worm-oram-full-tr.pdf`

[8] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly Secure Oblivious RAM Without Random Oracles. *TCC* 2011: 144-163.

[9] Oded Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. *STOC* 1987: 182-194.

[10] Oded Goldreich, and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43(3): 431-473 (1996).

[11] Michael T. Goodrich, and Michael Mitzenmacher. MapReduce Parallel Cuckoo Hashing and Oblivious RAM Simulations. *CoRR arXiv* abs/1007.1259v1, July 2010. `http://arxiv.org/abs/1007.1259v1`

[12] Michael T. Goodrich, and Michael Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. *CoRR arXiv* abs/1007.1259v2, May 2011. `http://arxiv.org/abs/1007.1259v2`

[13] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation. *CoRR arXiv* abs/1105.4125v1, May 2011. `http://arxiv.org/abs/1105.4125v1`

[14] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia Title: Oblivious RAM Simulation with Efficient Worst-Case Access Overhead. *CoRR arXiv* abs/1107.5093v1, July 2011. `http://arxiv.org/abs/1107.5093v1`

[15] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. In *ESA*, pages 611–622, 2008.

[16] Rafail Ostrovsky. Efficient Computation on Oblivious RAMs. *STOC* 1990: 514-523.

[17] Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. MIT Ph.D. Thesis Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, June 1992.

[18] Rafail Ostrovsky, and Victor Shoup. Private Information Storage (Extended Abstract). *STOC* 1997: 294-303.

[19] Rasmus Pagh, and Flemming Friche Rodler. Cuckoo hashing. In *ESA*, pages 121–133, 2001.

[20] Benny Pinkas, and Tzachy Reinman. Oblivious RAM Revisited. *CRYPTO* 2010: 502-519.

[21] Nicholas Pippenger, and Michael J. Fischer. Relations Among Complexity Measures. *J. ACM* 26(2): 361-381 (1979).

[22] Elaine Shi, Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. *Cryptology ePrint Archive*, Report 2011/407.

[23] Peter Williams, and Radu Sion. Usable PIR. *NDSS* 2008.

[24] Peter Williams, Radu Sion and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. *ACM Conference on Computer and Communications Security* 2008: 139-148.

A Algorithm Description

FOUND$\leftarrow$ $false$
Locally allocate temporary variables $v_r, v_x, id, output$
$id \leftarrow r$
for Each element (v_r, v_x) of B_k **do**
  if $v_r = r$ **then**
    FOUND$\leftarrow$ $true$; $v_r \leftarrow$ dummy $\circ\, T$; $id \leftarrow$ dummy $\circ\, T$; $output \leftarrow v_x$
  end if
  Re-encrypt (v_r, v_x) and store it back in the same location
end for
for $i = k+1, \ldots, K-1$ **do**
  for Each element (v_r, v_x) in bucket $h_i(id)$ in buffer B_i **do**
    if $v_r = r$ **then**
      FOUND$\leftarrow$ $true$; $v_r \leftarrow$ dummy $\circ\, T$; $id \leftarrow$ dummy $\circ\, T$; $output \leftarrow v_x$
    end if
    Re-encrypt (v_r, v_x) and store it back in the same location
  end for
end for
for $i = K, \ldots, L$ **do**
  for $j = t, \ldots, 1$ **do**
    $(v_r, v_x) \leftarrow B_i^j(h_{i,1}^j(id))$
    if $v_r = r$ **then**
      FOUND$\leftarrow$ $true$; $v_r \leftarrow$ dummy $\circ\, T$; $id \leftarrow$ dummy $\circ\, T$; $output \leftarrow v_x$
    end if
    Re-encrypt (v_r, v_x) and store it back in the same location
    $(v_r, v_x) \leftarrow B_i^j(h_{i,2}^j(id))$
    if $v_r = r$ **then**
      FOUND$\leftarrow$ $true$; $v_r \leftarrow$ dummy $\circ\, T$; $id \leftarrow$ dummy $\circ\, T$; $output \leftarrow v_x$
    end if
    Re-encrypt (v_r, v_x) and store it back
  end for
end for
if FOUND **then**
  $(v_r, v_x) \leftarrow (v, output)$
else
  $(v_r, v_x) \leftarrow (v,$ null$)$
end if
if Write operation **then**
  $v_x \leftarrow$ val
end if
Encrypt (v_r, v_x) and insert it into B_k
$T \leftarrow T + 1$
Return $output$