Oblivious RAM with *Worst-Case* Logarithmic Overhead[*]

Gilad Asharov[†] Ilan Komargodski[‡] Wei-Kai Lin[§] Elaine Shi[¶]

Abstract

We present the first Oblivious RAM (ORAM) construction that for N memory blocks supports accesses with *worst-case* $O(\log N)$ overhead for any block size $\Omega(\log N)$ while requiring a client memory of only a constant number of memory blocks. We rely on the existence of one-way functions and guarantee computational security. Our result closes a long line of research on fundamental feasibility results for ORAM constructions as logarithmic overhead is necessary.

The previous best logarithmic overhead construction only guarantees it in an *amortized* sense, i.e., logarithmic overhead is achieved only for long enough access sequences, where some of the individual accesses incur $\Theta(N)$ overhead. The previously best ORAM in terms of *worst-case* overhead achieves $O(\log^2 N/\log\log N)$ overhead.

Technically, we design a novel de-amortization framework for modern ORAM constructions that use the "shuffled inputs" assumption. Our framework significantly departs from all previous de-amortization frameworks, originating from Ostrovsky and Shoup (STOC '97), that seem to be fundamentally too weak to be applied on modern ORAM constructions.

[*]A preliminary version of this paper appeared in IACR-CRYPTO 2021.

[†]Bar-Ilan University. Email: `Gilad.Asharov@biu.ac.il`.

[‡]Hebrew University and NTT Research. Email: `ilank@cs.huji.ac.il`.

[§]Cornell University. Email: `wklin@cs.cornell.edu`.

[¶]CMU. Email: `runting@gmail.com`.

Contents

1 Introduction

Imagine a client that wishes to offload a database containing sensitive information to an untrusted server and later access the database and retrieve parts of it. By now, it is well-known that merely encrypting the entries of the database before uploading them to the server does not guarantee privacy (e.g., [6,22,23,42]). Indeed, the access patterns themselves may reveal non-trivial information about the underlying data or program being executed on the data. To mitigate these kinds of attacks, we would like to be able not only to encrypt the underlying data but also to "scramble" the observed access patterns so that they look unrelated to the data. The algorithmic tool that achieves this goal is called an Oblivious RAM (ORAM).

An ORAM, introduced in the seminal work of Goldreich and Ostrovsky [17,18], is a (probabilistic) RAM machine whose memory accesses do not reveal anything about the input—including both program and data—on which it is executed. An ORAM construction accomplishes this by permuting data blocks stored on the server and periodic reshuffling them around. Since their introduction more than 30 years ago, ORAMs have also become a central tool in designing various cryptographic systems, including cloud computing design, secure processor design, multi-party computation protocols, and more [4,13,14,16,27–30,32,34,35,39–41].

To be useful, ORAMs have to be "efficient". Whether an ORAM is efficient or not is typically measured by its (asymptotic) *overhead in bandwidth*: that is, how many data items must be accessed in the oblivious simulation as compared to the original non-oblivious implementation. There has been a tremendous effort in designing the most efficient ORAM construction possible [2,7,18,19,25, 31,33,36,38]. The current record is the OptORAMa scheme by Asharov et al. [2] (building on Patel et al. [31]) who obtained an ORAM with amortized logarithmic overhead. Namely, their ORAM can simulate a RAM of size N so that over the span of T accesses, for large enough T, the total number of accesses would be $O(T \cdot \log N)$. The beautiful lower bound of Larsen and Nielsen [26] (see also [24]) shows that this is essentially the best possible: that is, every ORAM construction must spend *on average* $\Omega(\log N)$ physical accesses per one logical operation.[1]

Worst-case overhead. Much of the recent progress on ORAM constructions focuses on reducing its amortized cost [2,31], whereas the worst-case overhead of an operation was ignored. Specifically, while achieving logarithmic amortized overhead, these constructions have $\Omega(N)$ worst-case overhead, due to the occasional reshuffling operations. This worst-case behavior renders these schemes much less useful in many applications since every now and then an access will "block" until $\Omega(N)$ physical accesses are complete which is clearly unacceptable.

The first to address this problem were Ostrovsky and Shoup [30] who showed how to spread the reshuffling operations over time, and achieve a worst-case $O(\log^3 N)$ overhead version of the original ORAM of Goldreich and Ostrovsky [18]. Related techniques were later applied to other ORAM schemes [7,20,25]. In spite of the recent great progress in ORAM constructions, the best known construction in terms of worst-case overhead is from almost a decade ago due to Kushilevitz, Lu, and Ostrovsky [25] who achieved $O(\log^2 N/\log\log N)$ worst-case overhead (and their scheme was further clarified in the subsequent work of Chan et al. [7]). Crucially, the techniques of Ostrovsky and Shoup do not apply to the recent ORAM constructions that are based on "randomness reusing" of [2,31], as we elaborate below in Section 2.

[1]The lower bounds of [24,26] only apply to "online" ORAMs which support operations that come in an online fashion, one by one. These lower bounds even apply to computationally secure constructions. There is a logarithmic lower bound for "offline" ORAMs which see the whole set of operations ahead of time due to Goldreich and Ostrovsky [18], but it only applies to statistically secure constructions in the balls-and-bins model (see Boyle and Naor [5]).

Thus, the current state of affairs leaves open the following fundamental question (also raised in [2]):

Is there a worst-case *logarithmic overhead ORAM? That is, is there an ORAM construction that can simulate* every *logical operation with $O(\log N)$ physical accesses?*

1.1 Our Contributions

Optimal worst-case overhead ORAM. We propose a new ORAM construction that achieve logarithmic worst-case overhead (in the memory size), while consuming $O(1)$ client-side storage, and $O(N)$ server-side storage. Here, N denoted the memory size. Obliviousness of our construction relies on the existence of one-way functions.

Theorem 1.1. *There is a computationally-secure ORAM with $O(\log N)$* worst-case *overhead assuming that one-way functions exist.*

The construction that achieves Theorem 1.1 is in the most commonly studied model for computationally secure ORAMs and makes the same set of assumptions. We assume a standard word-RAM where each memory word has at least $w = \log N$ bits, i.e., large enough to store its own logical address. We assume that word-level addition and Boolean operations can be done in unit cost. We assume that the CPU has a constant number of (word size) private registers. We additionally assume that a single evaluation of a PRF resulting in at least word-size number of pseudorandom bits, can be done in unit cost.

In terms of techniques, we depart from all previous "deamortized" ORAM constructions. While all previous works[2] almost directly use the approach of Ostrovsky and Shoup [30], it seems like this approach is fundamentally too weak to be applied on modern ORAM constructions that achieve amortized logarithmic overhead [2, 31]. To this end, we build a new set of novel algorithmic tools from ground up and show how to amend the construction of Asharov et al. [2] so that it could be deamortized.

Linear-time oblivious deduplication. A noteworthy building block that we develop is an algorithm for efficient oblivious *deduplication*. Consider two sets of n elements A and B, where the goal is to obliviously compute $A \cup B$, that is, merge them into one larger set while removing duplicate elements. (Note that we assume that the elements within A are distinct, and ditto for B. Also, we assume that if there is a duplication, we keep the copy coming from A for concreteness.) Oblivious deduplication is a central building block in many previous worst-case efficient ORAM constructions. Before this work, the only known solution was to apply a generic oblivious sort on the concatenation of A and B, followed by a linear scan which "deletes" the duplicates (after the sort, duplicates are adjacent in memory). Using the best known oblivious sort (e.g., AKS [1]), this approach would incur $O(n \cdot \log n)$ time. Eventually the extra log factor propagates into the overhead of the ORAM, resulting in $O(\log^2 N/\log\log N)$ worst-case overhead (at best).

We show that the extra $\log n$ overhead can be avoided by designing a *linear time* deduplication algorithm which is oblivious if the input arrays are randomly shuffled. That is, we show that if A and B are shuffled with independent secret permutations, then there is a way to compute $A \cup B$ in time $O(n)$ while maintaining obliviousness. Linear overhead is clearly the best possible (since just reading the input takes linear time), matching the state of the art without obliviousness.

[2]Here we ignore tree-based constructions [33, 36, 38] since it is not known how to use them to get even amortized logarithmic overhead.

Theorem 1.2. *There is a linear time (probabilistic) algorithm that gets as input two sets A and B and outputs $A \cup B$. The algorithm is further computationally oblivious if A and B are independently secretly shuffled and if one-way functions exist.*

Conclusions and Open Problems

We see our work as closing a long line of research on theoretical feasibility results for ORAM constructions. Our *worst-case logarithmic overhead ORAM* result, at least asymptotically, is optimal in terms of computational overhead. Unfortunately, the concrete constant, inherited from [2], underlying our construction is rather large. Using better generic building blocks (say the oblivious compaction algorithm of Dittmer and Ostrovsky [11]) we can get a much better constant but we believe that it is still too large for deployment. Whether there is a construction with (worst-case) logarithmic asymptotic overhead and a *small* constant is a major open problem.

Another exciting open problem is to bring down the cost of *statistically secure* ORAMs closer to $O(\log N)$ or prove that it is impossible. Specifically, the best statistically secure ORAMs has overhead $O\left(\log^2 N/\log\log N\right)$ [9] and it relies on the tree-based paradigm due to Shi et al. [33] which was later improved by [9, 10, 14, 36, 38]. Interestingly, tree based ORAMs have so far been more concretely efficient than hierarchical ones and so this question is also somewhat related to the previous one.

Lastly, we mention a recent work of Asharov et al. [3] who gives an optimal Oblivious *PRAM* (OPRAM). An OPRAM is an extension of ORAM to the parallel setting where several processors make concurrent accesses to a shared memory. Their main result is that (assuming one-way functions) any PRAM with memory capacity N can be obliviously simulated in space $O(N)$, incurring only *amortized* $O(\log N)$ overhead in work and (worst-case) $O(\log N)$ overhead in depth. We believe that our techniques can be further extended to obtain a *worst-case* logarithmic work and depth overhead OPRAM, but this is left for future work.

2 Technical Overview

2.1 Background: Underlying ORAM Without Deamortization

The hierarchical paradigm and oblivious hash tables. We rely on an underlying (amortized) ORAM scheme which follows the hierarchical paradigm established by Goldreich and Ostrovsky [17, 18]. An ORAM scheme in the hierarchical paradigm can be viewed as a technique to reduce the task of constructing ORAM to the task of constructing an oblivious hash table. Specifically, a hierarchical ORAM typically consists of $\log_2 N + 1$ levels numbered $0, 1, 2, \ldots$. Each level i is an *oblivious hash table* that can contain at most 2^i elements. An oblivious hash table is a data structure that supports the following operations:

- Build takes an input array containing (key, value) pairs and creates the data structure (we also say a pair is an element, a block, or an item);
- Lookup receives a key k, and returns the value corresponding to the key k contained in the data structure, or returns $\perp$ if not found or if the key looked up is dummy (denoted $\perp$).
- Extract is called when the data structure is destructed, and returns a list of unvisited items in the data structure.

Almost all known ORAM schemes in the hierarchical paradigm guarantee the following *non-recurrent* invariant: for each oblivious hash table in the hierarchy, the same real (i.e., non-dummy) key must be looked up *at most once* during the life-cycle of the data structure. Therefore, the data structure only needs to provide obliviousness if this non-recurrent assumption is respected. Finally, an oblivious hash table often has an access budget in the sense that it can only support up to an a-priori fixed number of lookup requests. Typically this budget is at least the size of the array input into Build.

Achieving amortized logarithmic overhead. The original oblivious hash table implementation suggested by Goldreich and Ostrovsky [17, 18] is slow and takes $O(n \log n)$ time to build for an input array of size n. This would result in a non-optimal ORAM scheme. Instead, we adopt the efficient oblivious hash table suggested by Asharov et al. [2] (which is built upon Patel et al. [31]). Asharov et al. showed an oblivious hash table with $O(n)$ build time and $O(1)$ lookup overhead, except with the following input assumptions, output requirement, and caveat:

- *Randomly shuffled input assumption and output requirement:* the input array of Build must be randomly shuffled; moreover, the Extract function outputs the unvisited blocks in a randomly shuffled order. In either case, the randomness is hidden from the adversary.
- *Size assumption:* to obtain negligible in λ failure probability, the construction only works for hash tables that are at least $\mathsf{poly}\log(\lambda)$ in size.
- *Stash:* while the main hash table data structure can indeed be looked up in constant time, the hash table construction actually comes with a stash, and sometimes the element to be looked up actually resides in the stash. Each stash has expected constant size but with noticeable probability, the size can be as large as $O(\log \lambda)$.

We briefly overview how past works [2,31] deal with the above imperfectness to make the ORAM scheme work. The requirement of the first bullet is handled by utilizing the input assumption: when an oblivious hash table is destructed in the ORAM, the unvisited elements in the hash table appear in a random order, and the random permutation is hidden from the adversary (this is sometimes referred to as the "residual randomness" technique). The size assumption can be dealt with by using another designated, slower, data-structure for smaller levels in the ORAM that are less than $\mathsf{poly}\log(\lambda)$ in size (since there are few of those and they are small, it will not affect the overall asymptotic overhead). Finally, the stash issue can be dealt with by merging the stashes of all oblivious hash tables into a single one, and accessing the merged stash once and for all for each ORAM request — one can prove that the merged stash is at most poly-logarithmic in size except with negligible probability, and therefore can be stored in a designated data structure to allow fast lookup (i.e., taking strictly logarithmic time).

Simplifying assumptions. For ease of understanding, let us first ignore the size assumption and the stash issue mentioned above— these introduce additional technicalities for constructing an optimal, deamortized ORAM as we shall mention shortly. For the time being, we pretend that we can indeed have an oblivious hash table for randomly shuffled inputs that can be built in linear time, regardless of the size, and moreover, assuming that lookup need not deal with the stash technicality.

Underlying ORAM scheme without deamortization. With these simplifying assumptions, we can construct an ORAM scheme as follows—our description below matches the rebuild description of Asharov et al. [3] in their optimal OPRAM scheme, and is a variant of Goldreich and Ostrovsky's original hierarchical construction.

Assume that the total memory size N is a power of 2. Imagine that there are $\log_2 N + 1$ levels, where the i-th level is an oblivious hash table (for randomly shuffled inputs) of capacity 2^i. We use $\mathsf{T}_0, \ldots, \mathsf{T}_L$ to denote the $L+1$ levels where $L = \log_2 N$.

In the steady state of the ORAM (i.e., ignoring the initial time steps when the levels are not yet populated), every level except level 0 is either *half full* (HF) or *full* (F). A level $i > 0$ is *half full* iff it contains up to 2^{i-1} real blocks. A level i is *full* iff it may contain up to 2^i real blocks[3]. Level 0 is either empty or full, and as we shall see, it is guaranteed to be empty at the end of every ORAM request. Whenever a new ORAM request arrives asking for the block at logical address addr, we do the following:

- *Fetch phase.* From $i = 0$ to L, we look up each oblivious hash table for the logical address addr; once the block is found, for all subsequent levels, we instead look for a dummy block.
- *Maintain phase.* The block at addr just fetched is updated if necessary, and then it is entered into the smallest level (i.e., level 0), which makes the smallest level full. At this moment, let ℓ be either the smallest level that is half full or $\ell = L$ if all levels are full. Regardless of which case, all levels $0, 1, \ldots, \ell - 1$ are full. We now perform the following rebuild procedure:

  1. For each level $i = 0$ to $\ell - 2$ in parallel:
     let $\mathsf{T}'_{i+1} := \mathsf{Build}(\mathsf{Intersperse}(\mathsf{T}_i.\mathsf{Extract}(), D_i))$ — where D_i is an array of size 2^i containing only dummy elements, and Intersperse merges two randomly shuffled arrays into a randomly shuffled array.
  2. Let $\mathsf{T}'_\ell := \mathsf{Build}(\mathsf{T}_{\ell-1}.\mathsf{Extract}() \cup \mathsf{T}_\ell.\mathsf{Extract}())$ while we also remove dummy elements when unifying the two arrays;
  3. Replace $\mathsf{T}_1, \ldots, \mathsf{T}_\ell$ with the new hash tables $\mathsf{T}'_1, \ldots, \mathsf{T}'_\ell$ and let T_0 be emptied.

  After this rebuild procedure, $\mathsf{T}_0, \ldots, \mathsf{T}_{\ell-1}$ are all half full (in fact T_0 is empty), and T_ℓ becomes full. In the above Steps 1 and 2, one can also imagine that each level $1, \ldots, \ell - 1$ is "*rebuilding itself down*" into the next level (and we will use this terminology later). For ease of understanding, the maintain phase is depicted in Figure 1.

Fact 2.1. *In the above construction, a level $i \geq 1$ switches state (either from half full to full, or vice versa) every 2^{i-1} requests. Similarly, a level i wants to rebuild itself down every 2^i requests — this also coincides with when level $i+1$ refreshes.*

Assuming that the oblivious hash table supports Build in linear-time (for randomly shuffled inputs), supports Lookup in constant time, and moreover, its Extract function outputs unvisited blocks in a random order, then the above ORAM scheme is secure and achieves $O(\log N)$ *amortized* overhead.

2.2 Why Existing Deamortization Techniques Fail

Clearly the scheme mentioned in Section 2.1 cannot give a worst-case efficient ORAM. For instance, when rebuilding the largest level (which happens every N accesses), just reading it requires $O(N)$ work. We describe existing deamortization techniques and explain why they are incompatible with the new generation of amortized optimal ORAM constructions.

[3]The actual number of real blocks may be smaller if the requests keep asking for the same block or a small set of blocks. The maximum load is achieved when the ORAM requests cycle through addresses $1, 2, \ldots, N$ in a round-robin fashion.

Ostrovsky and Shoup [30] proposed a deamortization technique that, roughly speaking, "spreads" the rebuild procedures over many accesses rather than performing them atomically. The challenge is how to support accesses to the level while it is being rebuilt.

To do this, first, we modify the access process so that it does *not* delete an element once it is found in some level (while it is still reinserted into the smallest level). With this change, it is immediate that except for the smallest level, the only time the contents of a level changes, is during the rebuild procedure and in the latter we always just "pull" content from its previous (i.e., smaller) level. That is, it takes 2^i accesses until the content of the ith table is modified, and it will be filled with the content of table $i-1$ which is also fixed and known. Thus, the rebuild process for a level can be done slowly and in advance across many accesses as follows: each level has a table called CurrentActive and has another table that is UnderConstruction, which is being slowly built from CurrentActive and the previous level CurrentActive. When the construction of the level is complete, we just update the pointer of CurrentActive to the new rebuilt table, and start a new UnderConstruction version, thus doing the rebuild in the deamortized sense, by spreading the cost uniformly.

There is one very important technical detail with the above approach: since we never actually delete elements, the same key may (and will) appear multiple times in the structure, perhaps with different values. The important invariant is that the *newest* version of the element is always the one that resides in the smaller level. At some point, these copies will meet in the same level during some rebuild process and then the older copy will be suppressed and discarded. The later task is known as oblivious **deduplication** and it is usually implemented using oblivious sort.

Multiple variants and adaptations of the above deamortization technique were used in previous ORAM constructions [7, 20, 25]. However, as we shall argue next, there are inherent obstacles one runs into while trying to apply it on the more recent (amortized) logarithmic overhead ORAM constructions [2, 31].

Challenge I: Access-while-rebuild breaks security. Recall that in the deamortization technique of Ostrovsky and Shoup, we start rebuilding the table into the next level while we also access it at the same time. However, when applied on the ORAM of [2, 31], this completely breaks the input assumption (and therefore security) and is not compatible with the "residual randomness" technique: A lookup that is performed while building reveals the position of an element in the new table—the security of the latter inherently relies on the permutation being completely secret.

To elaborate further, in Ostrovsky and Shoup we know in advance the content of the levels and we start the rebuild process ahead of time, while still allowing accesses to the same levels. In the context of Ostrovsky and Shoup, this is *secure* as we re-randomize the levels (by obliviously sorting/shuffling it) during the rebuild process. However, in our case we cannot re-randomize the levels as this is too expensive, and we follow the residual randomness technique. Moreover, we cannot know in advance which elements will be looked up in the previous level, i.e., the residual randomness will be consumed and thus for security we are not allowed to reuse randomness.

Challenge II: Deduplication takes quasi-linear time. As mentioned, multiple copies of the same key will appear during the lifetime of the ORAM (some might be with different values) and so a deduplication mechanism is needed. More precisely, the task is to compute $A \cup B$ given two sets A and B of size n. (Assume we prefer the copy in A over B for concreteness.) The best known algorithm uses oblivious sort and takes quasi-linear time. However, if we want to have a logarithmic overhead ORAM construction we must implement it in *linear* time. Note that, even ignoring obliviousness, it is not immediately obvious how to do this in linear time. The most natural approach is to use sorting, but better algorithms can be achieved using hashing tools.

Challenge III: Cannot "lock" shared memory. As mentioned, each level in [2,31] *effectively* supports lookup in constant time, but this is due to the "shared stash" technique (previously used in [9, 19, 21, 25]). Specifically, in isolation, each level requires $O(1)$ accesses to a main table and an additional scan of a $O(\log N)$-size stash. The shared stash trick utilizes the fact that there are many levels and an access will translate to multiple lookups for the same key in many levels—this allows us to merge all of the stashes into a global one and perform lookup only once for all levels. Therefore, the number of accesses per level is $O(1)$ *amortized.* This makes the ORAM construction not completely black-box in a hash table and therefore less compatible with Ostrovsky-Shoup [30]. Concretely, while we rebuild a table, we *cannot* "lock" the global stash as we need to allow accesses to it (addition and removal of elements) while handling other ORAM accesses.

2.3 Our Deamortization Approach

We now intuitively describe how to deamortize the ORAM scheme mentioned in Section 2.1. To address Challenge I mentioned in Section 2.2, whenever some level is involved in a rebuild, i.e., its destructor $\mathsf{Extract}$ function is being called, this level can no longer support lookups. Yet the ORAM must continue to serve requests. We propose a new pipelining approach that works with this new constraint that comes with the new amortized logarithmic-overhead ORAMs.

Our key idea is to maintain two copies of each level, henceforth called the A-copy and the B-copy respectively, each uses its own *independent* randomness. At a high level, whenever some level in the A-copy is involved in a rebuild and being destructed, the *corresponding* level in the B-copy fills in the lookups; and whenever some level in the B-copy is involved in a rebuilt and being destructed, the *next level* in the A-copy can fill in. This guarantees that we never lookup and rebuild from a table at the same time – while rebuilding all lookups are performed at a different copy that uses independent randomness. That is, we essentially split the queries between A and B, so that we can reuse randomness in each hierarchy by itself. Whenever an element is found in one copy, we update its content and put it in the top of the two hierarchies of A and B.

We next elaborate our idea in the following schedule of deamortization. Henceforth, we use A_i and B_i to denote the i-th level in the A-copy and B-copy, respectively. In our earlier non-deamortized scheme, recall that a level $i \geq 1$ is reconstructed every 2^{i-1} requests. We may imagine that there is some counter ctr that increments upon every request, and thus a level $i \geq 1$ is refreshed whenever the $\mathsf{ctr} = k \cdot 2^{i-1}$ for some integer k. For simplicity, in this overview we ignore the treatment of the first level and the last level, and just look at intermediate levels. To achieve deamortization, the idea is to rely on careful pipelining to maintain the following schedule:

- Level A_i is refreshed (i.e., completes reconstruction) at time step $\mathsf{ctr}_1 = k \cdot 2^i + 2^{i-2}$ where k is an integer. Level A_i is now "half full" and just pulled new content from level $i-1$.
- Level B_i is refreshed at time $\mathsf{ctr}_2 = k \cdot 2^i + 2 \cdot 2^{i-2}$, i.e., just a little later than the corresponding A_i finished reconstruction.
- At time $\mathsf{ctr}_2 = k \cdot 2^i + 2 \cdot 2^{i-2}$, level A_i starts pulling more content from A_{i-1}. This will take 2^{i-2} time, which will finish at $\mathsf{ctr}_3 = k \cdot 2^i + 3 \cdot 2^{i-2}$.
  During this rebuild period $[\mathsf{ctr}_2, \mathsf{ctr}_3]$, A_i is dysfunctional and cannot support accesses; fortunately, B_i contains identical contents as A_i (but rerandomized differently for obliviousness), and therefore B_i can fill in for the lookups.
- At time ctr_3, A_i finishes, and B_i can catch up and pull information from level B_{i-1}. It will start rebuilding and will finish at time $\mathsf{ctr}_4 := \mathsf{ctr}_3 + 2^{i-2}$. At this time range B_i is dysfunctional,

however, its content is also in A_i, which also hold in addition some fresher content from A_{i-1}.

- After B_i finishes and becomes full, A_i wants to push all its content down into level A_{i+1} while also pulling new content from level A_{i-1}. However, the rebuild of level A_{i+1} takes twice as much time. Thus, level A_i will have new content already at time $\mathsf{ctr}_5 := \mathsf{ctr}_4 + 2^{i-2}$, while level A_{i+1} will have the old content of A_i only at time $\mathsf{ctr}_6 := \mathsf{ctr}_5 + 2^{i-2}$. Luckily, the content is not lost; The content exists all this time at the B_i copy, which is activated and functional in $[\mathsf{ctr}_4, \mathsf{ctr}_6]$.
  At time ctr_5, we essentially finished a cycle, i.e., A_i is half full and just finished refreshing, and we are essentially back to the first item.

Recall that rebuilding A_i (or B_i, resp.) from A_{i-1} (or B_{i-1}, resp.) takes work $\Theta(2^i)$. This amount of work is spread across $\mathsf{ctr}_2 - \mathsf{ctr}_1$ (or $\mathsf{ctr}_3 - \mathsf{ctr}_2$, resp.) time steps. One can verify that indeed, $\mathsf{ctr}_2 - \mathsf{ctr}_1 = \Theta(2^i)$ and thus in each time step, a constant amount of work is performed associated with the rebuilding of A_i (or B_i, resp.). At most $O(\log N)$ many levels are being rebuilt simultaneously, and thus the total amount of work per time step associated with rebuilds is $O(\log N)$. Observe also that the time in between two adjacent rebuilds of A_i is only 2^{i-1}, and the hash table A_i can support up to 2^i requests, so we will not run out of the access budget. A similar observation holds for B_i.

In our final scheme, we will actually have different designated place for the table A_i when it is half-full, and a different placed for A_i when it is full, denoted as $\mathsf{A}_i^{\mathsf{HF}}$ and $\mathsf{A}_i^{\mathsf{F}}$, respectively (likewise for $\mathsf{B}_i^{\mathsf{HF}}$ and $\mathsf{B}_i^{\mathsf{F}}$). Thus, we have 4 tables at the same level, but (the newest version of) each element has exactly 1 copy in each of A_i and B_i. Moreover, only one copy of $(\mathsf{A}_i^{\mathsf{HF}}, \mathsf{A}_i^{\mathsf{F}})$ is valid at any given time, likewise for B_i. We can view $\mathsf{A}_i^{\mathsf{HF}}$ and $\mathsf{A}_i^{\mathsf{F}}$ as two possible states of A_i, and then we have just two copies in each level, or as independent hash tables, and then we have four tables. The rebuild schedule is depicted in Figure 3.

Why deduplication is necessary in the deamortized scheme. The rebuild procedure of the deamortized scheme is otherwise the same as the underlying non-deamortized scheme except that a deduplication pre-processing step is necessary whenever we are building a half full level A_i and a full level A_{i-1} to create a new level A_i (and the same for the B-copy). This is because in the deamortized scheme, the rebuilding process is happening while the ORAM is still serving queries. For example, A_i may be rebuilding itself down into A_{i+1}, and the corresponding B_i is taking over as A_i in serving queries. During the rebuild, a memory block at address addr^* may get rebuilt from A_i, into A_{i+1}, but it can also be requested during the same rebuild interim, causing a separate copy of addr^* to be entered into the smallest level. Duplicates will be suppressed when two duplicate copies of the same addr^* "meet" in a future rebuild, while lookup guarantees that the freshest copy (which resides in the level that is smallest compared to any other copy) will be found.

We next describe how to accomplish oblivious deduplication in linear time.

2.4 Linear-Time Oblivious Deduplication

To address Challenge II from Section 2.2, another contribution of our paper is a linear-time oblivious deduplication algorithm that takes advantage of the fact that the input arrays to be deduplicated are randomly shuffled. To see the main idea, let us first describe a *non-oblivious* algorithm that runs in linear time. (This is already not completely trivial.) While the most natural implementation is to sort the concatenated array and then perform a linear scan while removing duplicates (which reside in adjacent positions after the sort), the cost of this implementation is dominated by the sort. Using the best sorting algorithms in the RAM model, one can achieve $o(n \cdot \log n)$ cost [15,37] but we still do not know of linear time sorting algorithms.

Our idea to get a linear time algorithm is to use hashing tools, and more specifically a Cuckoo hash. Cuckoo hash is a hash table that supports lookup by just 2 accesses to a main table and another scan of say $O(\log \lambda)$ size stash. (The stash needs to be of this size to guarantee the probability of failure is negligible in λ.) Consider hashing X_1 and X_2 independently into Cuckoo hash tables T_1, T_2 each having a long enough stash so that the hashing succeeds with all but negligible probability. For $i \in \{1,2\}$, denote $T_i = (T_i^M, T_i^S)$ where T_i^M is the main table of T_i and T_i^S is the stash of T_i. This costs just $O(n)$. Now, we can perform the deduplication as follows:

1. For each element in the stash of T_1, i.e., T_1^S, we perform a full lookup in T_2. That is, for each of the $O(\log \lambda)$ elements in T_1^S, we touch $O(1)$ elements in T_2^M and then scan T_2^S. Doing so, we remove the elements from T_2 that are also in T_1^S. This costs overall $O(\log^2 \lambda)$.
2. So far, we took care of elements that appear in T_2 and in T_1^S and we need to remove duplicates which appear in T_2 and T_1^M. For this, we scan every remaining element in T_2 and for each we touch the two locations in T_1^M to check if it is there. If so, we delete it from T_2. Crucially, now we do not need to visit the stash T_1^S. When we look for the elements in $T_2 \setminus T_1^S$, we can go directly to the main table and spend $O(1)$ work per lookup, as we know that the element is not in the stash T_1^S!

Eventually, we concatenate the elements of T_1 together with what is left in T_2 after performing the above process. Overall, the cost of this algorithm is $O(n + \log^2 \lambda) = O(n)$ whenever n is large enough compared to λ.

This algorithm is clearly non-oblivious and "naively" replacing the Cuckoo hash with an oblivious version thereof does not meet our goal since known constructions require $\omega(n)$ time for building (since building an oblivious Cuckoo hash uses oblivious sort). This is where the "shuffled inputs" assumptions comes into play: we do not necessarily need the full power of Cuckoo hashing since we are guaranteed that the input lists are shuffled. Therefore, instead of using Cuckoo hash, we use *in a white-box manner* the linear-time oblivious hash table for shuffled inputs from [2] (see also Section 3.4). This hash table has linear build time and is secure only when the input array is randomly shuffled; otherwise, it behaves conceptually in a similar manner to "standard" oblivious Cuckoo hash: lookup is performed by a scan of a stash and $O(1)$ accesses to a "main table". We therefore manage to use it for our purpose, deduplication.

Dealing with the shared stash. So far, we assumed an idealized oblivious hash table (for randomly shuffled inputs) without stashes. As mentioned earlier, known instantiations of the oblivious hash table with the desired efficiency requirements have an extra stash that must be visited during a Lookup operation, and the constant-time lookup is only possible with a "shared stash" trick, i.e., by merging all logarithmically many stashes into a globally shared one.

Accommodating the shared stashes creates extra technicalities, as we mentioned in Challenge III in Section 2.2. To deal with them, the main idea is to support more fine-grained access to the shared memory area. That is, while in [2], every extract requires to atomically scan the shared memory to retrieve all elements that "belong" to some given level, in our construction we avoid such linear scans by using more efficient data structures. Specifically, we use a version of an oblivious dictionary which supports lookup of elements w.r.t. various auxiliary keys such as the level they came from or the logical address.

To elaborate, recall that each level is associated with a stash of $O(\log \lambda)$ elements, and thus the shared stash consists of $O(\log N \cdot \log \lambda)$ elements. All these elements are store in an oblivious dictionary accompanied with the auxiliary keys. Specifically, an element is inserted to the dictionary when a level is newly built, is queried by a logical address when lookups are performed during the fetch phase, and is popped from the dictionary when a level is being extracted. Recall that each operation of the oblivious dictionary takes only $\mathsf{poly}\log(\log N + \log \lambda)$ time, e.g., by instantiating a perfect ORAM [8], and that each level is at least $\mathsf{poly}(\log N + \log \lambda)$ in size. The efficiency follows as inserting or popping elements take only a $o(1)$ fraction of time during build or extract a level, and only $O(1)$ queries are performed during a fetch. The security follows since the dictionary is perfectly oblivious.

Organization. The remaining of the paper is organized as follows. In Section 3 we provide the preliminaries, which includes definition of obliviousness, some basic building blocks we use in our construction, our 2-key dictionary and revisit and overview the oblivious hash table construction of [2]. In Section 4 we provide our deduplication algorithm, and in Section 5 we provide our deamortized construction. The case of combining the stashes is deferred to Appendix C.

3 Preliminaries

The security parameter is denoted λ and it is given as input to algorithms in unary (i.e., as 1^λ). A function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}^+$ is *negligible* if for every constant $c > 0$ there exists an integer N_c such that $\mathsf{negl}(\lambda) < \lambda^{-c}$ for all $\lambda > N_c$. Two sequences of random variables $X = \{X_\lambda\}_{\lambda\in\mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda\in\mathbb{N}}$ are *computationally indistinguishable* if for any probabilistic polynomial time algorithm $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that $\left|\Pr\left[\mathcal{A}(1^\lambda, X_\lambda) = 1\right] - \Pr\left[\mathcal{A}(1^\lambda, Y_\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$. For $n \in \mathbb{N}$, denote $[n] = \{1, \ldots, n\}$.

Random-access machines (RAM). A RAM (or *a RAM program*) is an interactive Turing machine that consists of a memory and a CPU. The program maps some input to an output where its computation is performed by the CPU and using the interaction with the memory. The memory is denoted as $\mathsf{mem}[N, w]$ and is indexed by the logical address space $[N] = \{1, 2, \ldots, N\}$. We denote by w to denote the bit-length of each block. The CPU has an internal state that consists of $O(1)$ words. The memory supports read/write instructions $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w \cup \{\perp\}$:

- If $\mathsf{op} = \mathsf{read}$ then $\mathsf{data} = \perp$ and the returned value is the content of the block located in the logical address addr in the memory.
- If $\mathsf{op} = \mathsf{write}$ then the memory data in logical address addr is updated to data.

We use the standard setting that $w = \Theta(\log N)$ (so an address can be stored in a word). We follow the standard convention that the CPU performs one word-level operation per unit time, i.e., such that additions or subtraction (arithmetic operations), bitwise operations such as AND, OR, NOT or shift, memory accesses or evaluation of pseudorandom function.

3.1 Oblivious Machines

Intuitively, we say that a machine M is *oblivious* if there exists a simulator that can simulate its access pattern without knowing the input. Specifically, there exists a simulator Sim such that, for all inputs x, all memory accesses in the computation $M(x)$ can be simulated by Sim where Sim just receives the length of x (i.e., $|x|$) and not the input itself.

We say that a RAM program M_f oblivious simulates a (deterministic) RAM program f if for every input x it holds that $M_f(x) = f(x)$, and that M_f is oblivious. For randomized functionalities, we require that the joint distribution of the output of M_f and the access pattern of the simulator is indistinguishable from the joint distribution of the output of f and its access pattern in the computation. See discussion in [2]. We are now ready for the definition of oblivious simulation:

Definition 3.1 (Oblivious simulation)**.** *Let $f, M_f : \{0,1\}^* \to \{0,1\}^*$ be two RAM machines. We say that M_f* obliviously simulates *f if there exists a probabilistic polynomial time simulator* Sim *such that for every input $x \in \{0,1\}^*$, the following holds:*

$$\left\{(\mathsf{out}, \mathsf{Addr}) \;:\; (\mathsf{out}, \mathsf{Addrs}) \leftarrow M_f(1^\lambda, x)\right\}_\lambda \approx \left\{\left(f(x), \mathsf{Sim}(1^\lambda, 1^{|x|})\right)\right\}_\lambda$$

depending on whether $\approx$ refers to computational, statistical, or perfectly indistinguishable we say that M_f is computationally, statistically, or perfectly oblivious, respectively.

Reactive random-access machines. We consider functionalities that are reactive, i.e., proceed in stages, where the functionality preserves an internal state between stages. Such a reactive functionality can be described as a sequence of RAM machines, where each machine also receives as an input a state, updates it, and the output is the input state for the next machine. We use it to capture building blocks such as oblivious hash tables (see Section 3.4).

A reactive machine $\mathcal{F}$ receives commands of the form $(\mathsf{command}_i, \mathsf{inp}_i)$ and produces an output out_i while maintaining some (secret) internal state. Our definition considers an adversary $\mathcal{A}$ (a distinguisher) that participates in either a real execution or an ideal one, and with each command receives the access pattern (resp. simulated access pattern) and the output of the algorithm (resp. output of the functionality). The adversary $\mathcal{A}$ can then adaptively choose the next command to execute.

Definition 3.2 (Oblivious simulation of a reactive functionality)**.** *We say that a reactive machine $M_\mathcal{F}$ is an oblivious implementation of the reactive functionality $\mathcal{F}$ if there exists a PPT simulator* Sim *such that for any non-uniform PPT (stateful) adversary $\mathcal{A}$, the view of the adversary $\mathcal{A}$ in the following two experiments $\mathsf{Expt}^{\mathsf{real}, M_\mathcal{F}}(1^\lambda)$ and $\mathsf{Expt}^{\mathsf{ideal},\mathcal{F}}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)$ is computationally indistinguishable:*

```
$\mathsf{Expt}^{\mathsf{real}, M_\mathcal{F}}_{\mathcal{A}}(1^\lambda)$:
Let $(\mathsf{cmd}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}(1^\lambda)$
Loop while $\mathsf{cmd}_i \neq \bot$:
    $\mathsf{out}_i, \mathsf{Addr}_i \leftarrow M_\mathcal{F}(1^\lambda, \mathsf{cmd}_i, \mathsf{inp}_i)$

    $(\mathsf{cmd}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_i, \mathsf{Addr}_i)$
```

```
$\mathsf{Expt}^{\mathsf{ideal},\mathcal{F}}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)$:
Let $(\mathsf{cmd}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}(1^\lambda)$
Loop while $\mathsf{cmd}_i \neq \bot$:
    $\mathsf{out}_i \leftarrow \mathcal{F}(\mathsf{cmd}_i, \mathsf{inp}_i)$.
    $\mathsf{Addr}_i \leftarrow \mathsf{Sim}(1^\lambda, \mathsf{cmd}_i)$.
    $(\mathsf{cmd}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_i, \mathsf{Addr}_i)$
```

We define statistical or perfect simulation analogously, requiring the two experiments to be either statistically close or identically distributed.

ORAM simulation overhead. We consider the standard ORAM functionality, implementing a logical memory. In this functionality, the user gets to choose the next command (i.e., either read or write) as well as the address and data according to the access pattern it has observed so far. During its life span, the functionality holds (as an internal state) N memory blocks, each of size w. Denote the internal state $\mathbf{X}[1, \ldots, N]$. Initially, $\mathbf{X}[\mathsf{addr}] = 0$ for every $\mathsf{addr} \in [N]$. The functionality is as follows:

- $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$: where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w$.
  1. If $\mathsf{op} = \mathsf{read}$, set $\mathsf{data}^* := \mathbf{X}[\mathsf{addr}]$.
  2. If $\mathsf{op} = \mathsf{write}$, set $\mathbf{X}[\mathsf{addr}] := \mathsf{data}$ and $\mathsf{data}^* := \mathsf{data}$.
  3. Output data^*.

Typically, the metric of interest for ORAM construction is known as *computation overhead* and it is is defined as the (multiplicative) blowup in runtime of the compiled program. We distinguish between the worst-case and the amortized variants:

- **Amortized computation overhead**: We say that the amortized computation overhead of the ORAM is $g\colon \mathbb{N} \to \mathbb{N}$ if for every sequence of operations $((\mathsf{op}_1, \mathsf{addr}_1, \mathsf{data}_1), \dots, (\mathsf{op}_q, \mathsf{addr}_q, \mathsf{data}_q))$ at most $g(N) \cdot q$ computation steps are taken during the execution of the ORAM.

- **Worst-case computation overhead**: We say that the amortized computation overhead of the ORAM is $g\colon \mathbb{N} \to \mathbb{N}$ if every sequence of operations $((\mathsf{op}_1, \mathsf{addr}_1, \mathsf{data}_1), \dots, (\mathsf{op}_q, \mathsf{addr}_q, \mathsf{data}_q))$, handling each operation in the sequence consumes $g(N)$ computation steps.

It is immediate that worst-case overhead $g(N)$ directly implies amortized overhead of $g(N)$ but the converse is not necessarily true.

3.2 Basic Building Blocks

We briefly describe few functionalities that we use in our construction, and refer to [2] for their implementation:

1. $\mathsf{Intersperse}(X, Y)$ is an algorithm that takes two arrays, each is randomly shuffled, returns a randomly shuffled array, and runs in linear time (i.e., $O(|X|+|Y|)$). It obliviously implements the ideal functionality $\mathcal{F}_{\mathsf{Shuffle}}$–which takes an array and randomly shuffled it.
2. $\mathsf{IntersperseRD}(X)$ takes an array that contains real and dummy elements, and is assumed that all real elements are shuffled among themselves, but there is no guarantee about the locations of the dummies in the array (e.g., they can all reside at the end of the array). The algorithm returns a randomly shuffled array, runs in linear time, and obliviously implements the ideal functionality $\mathcal{F}_{\mathsf{Shuffle}}$.
3. Compaction: Given an array in which some of the elements are distinguished, it moves all distinguished elements to the beginning of the array and runs in linear time.

3.3 Perfectly Oblivious 2-Key Dictionary

In this section we use known results to get a new oblivious dictionary-like data structure. Our structure, termed two-key dictionary, needs to support three operations: Insert, PopKey, and $\mathsf{PopTime}$. Each element has a key k and a label t for "time". The Insert operation takes the key k along with timestamp t and a value v and adds (k, t, v) to the dictionary, the k appears at most once in the dictionary (so that it overwrites if there is a previous tuple (k, t', v') for the same k). One can pop an element with key k by using $\mathsf{PopKey}(k)$—the operation returns and removes the element with the key k. Analogously, $\mathsf{PopTime}(t_1, t_2)$ takes as input a time period $[t_1, t_2]$ and returns an element that is labeled with a timestamp t in the given period. This (reactive) functionality appears as Functionality 3.3.

Functionality 3.3: $\mathcal{F}_{\mathsf{2KeyDict}}$ - Dictionary Functionality

- Initialization of the state: let M be an empty list indexed by $k \in [K]$ for the given key space K, where all $M[k]$ are initialized as $\perp$.
- $\mathcal{F}_{\mathsf{2KeyDict}}.\mathsf{Insert}(k, t, v)$:
  - **Input:** a key k, time $t \in \mathbb{N}$, and a value v, where k might be $\perp$, i.e., a dummy insertion.
  - **The procedure:**
    1. If $k \neq \perp$, set $M[k] := (t, v)$.
  - **Output:** The Insert operation has no input.
- $\mathcal{F}_{\mathsf{2KeyDict}}.\mathsf{PopKey}(k)$:
  - **Input:** a key k (that might be $\perp$, i.e., dummy).
  - **The procedure:**
    1. Set $(t^*, v^*) := M[k]$ and then set $M[k] := \perp$.
  - **Output:** The valude v^*.
- $\mathcal{F}_{\mathsf{2KeyDict}}.\mathsf{PopTime}(t_1, t_2)$:
  - **Input:** time $t_1, t_2 \in \mathbb{N}$ such that $t_1 < t_2$.
  - **The procedure:**
    1. Let k be the smallest index such that $M[k] = (t^*, \cdot)$ for some $t^* \in [t_1, t_2]$. If no such k exists, set $v^* := \perp$. Otherwise, set $(t^*, v^*) := M[k]$ and then set $M[k] := \perp$.
  - **Output:** The value v^*.

Theorem 3.4. *Assume the tuple of (k, t, v) (i.e., key, time, and value) can be stored in a constant number of memory words. Assume further that the two-key dictionary needs to support at most n elements. There exists a perfectly oblivious implementation of functionality $\mathcal{F}_{\mathsf{2KeyDict}}$ such that each operation* Insert*,* PopKey*, and* $\mathsf{PopTime}$ *takes $O(\log^4 n)$ time in the worst-case.*

Proof. The first step is to obtain a perfectly oblivious ORAM that has $O(\log^3 n)$ worst-case overhead when simulating a memory of size n. Such an ORAM can be obtained by applying the deamortization technique of Ostrovsky and Shoup [30] on the (amortized) $O(\log^3 n)$ overhead perfect ORAM construction of Chan et al. [8]. It directly works (although never formally stated to the best of our knowledge) since the construction of Chan et al. is based exactly on the original hierarchical framework of Goldreich and Ostrovsky [17, 18] which was deamortized in Ostrovsky and Shoup [30].

Given this ORAM, it is straightforward to prove the theorem by just compiling a non-oblivious implementation of $\mathcal{F}_{\mathsf{2KeyDict}}$. For the latter, we instantiate two balanced binary search trees (e.g., red-black tree), where the first tree orders elements according to the key k, and the second tree orders elements by the given time t. This implementation has logarithmic cost for each operation. Therefore, compiling it using the above worst-case perfect ORAM, we have a perfectly oblivious implementation of $\mathcal{F}_{\mathsf{2KeyDict}}$ taking $O(\log^4 n)$ time in the worst case. $\square$

3.4 Oblivious Hash Table for Shuffled Inputs

Here we recall what an oblivious hash table is and the shuffled inputs assumption (taken from [2, Section 4.4]). An oblivious hash table is a (reactive) functionality that supports three operations: $\mathsf{Build}, \mathsf{Lookup}$ and $\mathsf{Extract}$ that are defines as follows. The Build operation gets as input an array of items, Lookup is used to search for an item and then delete it, and finally $\mathsf{Extract}$ returns the "remaining" elements in the table. Obliviousness means, as usual, that the access patterns throughout the life time of the system should be unrelated to the elements in the array nor the values being searched for. We will achieve this guarantee *assuming the input to* Build *is random shuffled.*

Functionality 3.5: $\mathcal{F}_{\mathsf{HT}}$ - Hash Table Functionality for Non-Recurrent Lookups

$\mathcal{F}_{\mathsf{HT}}.\mathsf{Build}(\mathbf{I})$: The input

- **Input:** an input array $\mathbf{I} = (a_1, \ldots, a_n)$ containing n elements, where each a_i is either dummy or a (key, value) pair denoted $(k_i, v_i) \in \{0,1\}^D \times \{0,1\}^D$ for some $D \in \mathbb{N}$. We assume that both the key and the value can be stored in $O(1)$ memory words, i.e., $D = O(w)$ where w denotes the word size.
- **The procedure:**
  1. Initialize the internal state state to $\mathbf{I}, \mathbf{P}$ where $\mathbf{P} = \emptyset$. $\mathbf{P}$ will store the keys that were already queried.
  2. **Output:** The Build operation has no input.

$\mathcal{F}_{\mathsf{HT}}.\mathsf{Lookup}(k)$:

- **Input:** a key $k \in \{0,1\}^D \cup \{\perp\}$.
- **The procedure:**
  1. Parse the internal state as $\mathsf{state} = (\mathbf{I}, \mathbf{P})$.
  2. If $k \in \mathbf{P}$ (i.e., k is a recurrent lookup) then halt and output fail. (Security is only guaranteed if no such recurrent lookup is performed, so in the construction we do not need to check this explicitly.)
  3. If $k = \perp$ or $k \notin \mathbf{I}$ then set $v^* = \perp$.
  4. Otherwise, set $v^* = v$ where v is the value that corresponds to the key k in $\mathbf{I}$.
  5. Update $\mathbf{P} = \mathbf{P} \cup \{(k, v)\}$.
- **Output:** The element v^*.

$\mathcal{F}_{\mathsf{HT}}.\mathsf{Extract}()$:

- **The procedure:**
  1. Parse the internal state $\mathsf{state} = (\mathbf{I}, \mathbf{P})$.
  2. Define an array $\mathbf{I}' = (a'_1, \ldots, a'_n)$ as follows. For $i \in [n]$ set $a'_i = a_i$ if $a_i = (k, v) \notin \mathbf{P}$. Otherwise, set $a'_i = \mathsf{dummy}$.
  3. Shuffle $\mathbf{I}'$ uniformly at random.
- **Output:** The array $\mathbf{I}'$.

The work of Asharov et al. [2, Corollary 8.9] shows a construction of a hash table, denoted as CombHT, with the following properties:

Theorem 3.6. *Assume that one-way functions exist. Then, for any $c \in \mathbb{N}$, there exists a construction, denoted as* CombHT*, that (computationally) obliviously implements the $\mathcal{F}_{\mathsf{HT}}$ functionality (Functionality 3.5) with the following properties:*

1. *The input array is $\log^{9+c} \lambda \leq n \leq 2^{\lambda}$;*
2. *The input assumption is that the input array is randomly shuffled;*
3. Build *and* Extract *each take $O(n)$ time.* Build *outputs a main table and a stash of size $O(\log \lambda)$;*
4. Lookup *takes $O(1)$ time in addition to linearly scanning a stash of size $O(\log \lambda)$.*

The high-level idea of this construction is given in Appendix A for completeness, and we refer to [2, Section 8.4] for full details.

4 Oblivious Deduplication in Linear Time

Consider two arrays X_1 of size n and X_2 of size $2n$, where it is guaranteed that at least half of the elements in X_2 are dummies,[4] and the keys in each array are unique. In what follows we describe an algorithm for merging the (real) contents of the arrays while removing duplicates, preferring the ones in X_1. That is, viewing the input arrays as sets, we compute $X_1 \cup X_2$ while preferring duplicate elements from X_1. We start with the abstract functionality $\mathcal{F}_{\mathsf{Dedup}}$ and then give our implementation.

The functionality $\mathcal{F}_{\mathsf{Dedup}}$. The exact functionality is described next. It can be viewed as a non-efficient non-oblivious implementation. The input consists of an array X_1 of size n and an array X_2 of size $2n$. The array X_2 contains at most n real elements. Every key appears at most once in each input list (a key may appear once in each of the arrays with different associate values). The functionality does the following:

1. Initialize an array Y of size $2n$.
2. Copy to Y all real elements in both arrays X_1 and X_2. If the two arrays contain the same key (with possibly different associated value), then remove the copy from X_2 and prefer the one in X_1. Pad Y with dummies to be of size $2n$.
3. Uniformly shuffle Y and return it.

The main theorem of this section is stated next.

Theorem 4.1. *There is an algorithm that implements the functionality $\mathcal{F}_{\mathsf{Dedup}}$ in time $O(n)$ for $n \geq \log^{11} \lambda$ (and with negligible error probability). The algorithm is computationally oblivious if one-way functions exist and if the input arrays are independently randomly shuffled.*

Recall the $O(n)$ time non-oblivious algorithm from Section 2—it is clearly non-oblivious and "naively" replacing the Cuckoo hash with an oblivious version thereof does not meet our goal since known constructions require $\omega(n)$ time for building. However, we do not necessarily need the full power of Cuckoo hashing since we are guaranteed that the input lists are shuffled. Therefore, instead of using Cuckoo hash, we use the hash table CombHT from Section 3.4 which has linear build time and otherwise behaves conceptually in a similar manner to "standard" oblivious Cuckoo hash: lookup is performed by a scan of a stash and $O(1)$ accesses to a "main table". The idea therefore is conceptually in the same spirit, but we rearrange and then compose the procedures of CombHT (in a non-black-box way) to guarantee obliviousness. Namely in Step 3, for each duplicate that reside in the first hash table, we mark the element by its counterpart in the second hash table; then in Step 5, we are able to emulate identically the lookup procedures on the second table (even we perform no access on its stash). We refer the reader to the construction of CombHT Appendix A for a comprehensive construction.

The algorithm $\mathsf{Dedup}(X_1, X_2)$ works as follows:

1. Perform $\mathsf{T}_1 := \mathsf{HT.Build}(X_1)$ and $\mathsf{T}_2 := \mathsf{HT.Build}(X_2)$.
2. Denote $\mathsf{T}_1 = (\mathsf{sk}_1, \mathsf{OBins}_1, \mathsf{CombS}_{1,\mathsf{T}}, \mathsf{CombS}_{1,\mathsf{S}}, \mathsf{OF}_{1,\mathsf{T}}, \mathsf{OF}_{1,\mathsf{S}})$ and $\mathsf{T}_2 = (\mathsf{sk}_2, \mathsf{OBins}_2, \mathsf{CombS}_{2,\mathsf{T}}, \mathsf{CombS}_{2,\mathsf{S}}, \mathsf{OF}_{2,\mathsf{T}}, \mathsf{OF}_{2,\mathsf{S}})$.
3. Initialize an empty array L. Linearly scan $\mathsf{OF}_{2,\mathsf{S}}$ and $\mathsf{Comb}_{2,\mathsf{S}}$, and for each element (k, v), perform the following:

[4]One could easily modify our algorithm to work more generally for a list X_2 of size m which has at least n dummies and result with an array of size m. We chose to be concrete for simplicity.

   (a) Perform a real lookup $(k', v') := \mathsf{T}_1.\mathsf{Lookup}(k)$. If k is found in T_1 then:
      i. Mark the element (k', v') at T_1 as "`CombS`" if k comes from $\mathsf{CombS}_{2,\mathsf{S}}$, or "`OF`" if k comes from $\mathsf{OF}_{2,\mathsf{S}}$.
      ii. Mark the element (k, v) as "`accessed`" in T_2.
   (b) Write (k', v') to the next slot in L (including the mark when k is found).

   In the end of this loop, obliviously shuffle L.
4. Perform $S'_1 := \mathsf{T}_1.\mathsf{Extract}()$. Then, perform $S_1 := \mathsf{Intersperse}(S'_1 \| L)$.
5. Linearly scan S_1, and for each element (k, v), perform the following:
   (a) If k is marked as "`OF`", then perform a real lookup $(k', v') := \mathsf{T}_2.\mathsf{Lookup}(k)$ while not scanning the stashes $(\mathsf{OF}_{2,\mathsf{S}}, \mathsf{CombS}_{2,\mathsf{S}})$ and proceeding as if k is found in $\mathsf{OF}_{2,\mathsf{S}}$.
   (b) If k is marked as "`CombS`", then perform a real lookup $(k', v') := \mathsf{T}_2.\mathsf{Lookup}(k)$ while not scanning the stashes $(\mathsf{OF}_{2,\mathsf{S}}, \mathsf{CombS}_{2,\mathsf{S}})$ and proceeding as if k is found in $\mathsf{CombS}_{2,\mathsf{S}}$.
   (c) If k is not marked, then perform a real lookup $(k', v') := \mathsf{T}_2.\mathsf{Lookup}(k)$ while not scanning the stashes $(\mathsf{OF}_{2,\mathsf{S}}, \mathsf{CombS}_{2,\mathsf{S}})$ and proceeding as if k is not found in the stashes.
6. Perform $S_2 := \mathsf{T}_2.\mathsf{Extract}()$ (recall that "`accessed`" elements in T_2 are not extracted).
7. Run $Z := \mathsf{Intersperse}(S_1 \| S_2)$. Run tight compaction on Z to move all dummy elements to the end. Truncate the array to be of size $2n$. Run $\mathsf{IntersperseRD}$ to randomly shuffle Z.
8. Output Z.

The full proof of Theorem 4.1 appears in Appendix B. Here, we briefly argue that the efficiency is as required. In step 3, we scan the stashes of T_2 and then for each element perform a $O(\log \lambda)$-time lookup, and then we shuffle L. Since the stashes are of size $O(\log \lambda)$, the running time of this step is $O(\log^2 \lambda) \leq O(n)$. Step 5 consist of a linear scan of a list and then an $O(1)$ lookup on each item. Steps 4 and 6 consume $O(n)$ time. Step 7 consumes $O(n)$ time, as well. Overall, the overhead is linear in n, as needed.

5 The ORAM Construction with Worst Case Complexity

In this section we present our deamortized construction.

The combined stash technique. As we saw in Section 3.4, our hash table supports lookup in $O(1)$ time in addition to a lookup in a stash of size $O(\log \lambda)$. To allow faster lookups, constructions use "the combined stash" technique (see [2, 7, 21]). According to this technique, all stashes of all levels are combined into one global stash. Then, utilizing the fact that we look for the same element in all levels (or dummy lookup once the element is found), we have to search for the element only once in the global stash (instead of searching for it in $O(\log N)$ different stashes), and then spend just $O(1)$ lookup time per level.

As we mentioned in the introduction, the fact that there is a shared memory to all levels introduces some complications in the final construction. We therefore present our deamortized construction in two steps:

1. In Section 5.1 we look at a somewhat idealized construction in which the main building block is a hash table that takes $O(1)$ time per lookup, while it takes linear time for $\mathsf{Extract}$ and Build (on shuffled inputs). That is, "there is no stash". We emphasize that we do not know how to realize such an oblivious hash table. Nevertheless, we describe this construction for aiding understanding and capturing the main ideas behind our deamortized construction.

2. In Appendix C we proceed to our final construction, in which the hash table is implemented as in Section 3.4, that is, $O(1)$ lookup time in addition to a scan of a stash of size $O(\log\lambda)$. To achieve effectively $O(1)$ time per lookup, we have also to use the combining stash technique as we described above.

5.1 Assuming Hash Table with $O(1)$ Lookup Time

In this section, we assume the existence of a construction of a hash table, denoted as HT, that achieves the following:

Assumption 5.1. *Assume that for any $c \in \mathbb{N}$ there exists a construction, denoted as HT that obliviously implements the $\mathcal{F}_{\mathsf{HT}}$ functionality (Functionality 3.5) with the following properties:*

1. *The input array is $\log^{9+c}\lambda \le n \le 2^{\lambda}$;*
2. *The input assumption is that the input array is randomly shuffled;*
3. Build *and* $\mathsf{Extract}$ *each take $O(n)$ time.*
4. Lookup *takes $O(1)$ time.*

This is equivalent to Theorem 3.6 where the construction has no stash and Lookup takes worst-case $O(1)$ time. We proceed to the construction. We first start with the underlying primitives and the memory organization, and proceed to the specification of the construction.

Structure: Let $\ell = \lceil 11\log\log\lambda\rceil$ and $L = \lceil\log N\rceil$.

1. Each level $i \in \{\ell+1,\ldots,L\}$ consists of four instances of HT as in Assumption 5.1, each of capacity 2^i. We denote the levels as $(\mathsf{A}^{\mathsf{HF}}_{\ell+1},\ldots,\mathsf{A}^{\mathsf{HF}}_{L})$, $(\mathsf{A}^{\mathsf{F}}_{\ell+1},\ldots,\mathsf{A}^{\mathsf{F}}_{L})$, $(\mathsf{B}^{\mathsf{HF}}_{\ell+1},\ldots,\mathsf{B}^{\mathsf{HF}}_{L})$ and $(\mathsf{B}^{\mathsf{F}}_{\ell+1},\ldots,\mathsf{B}^{\mathsf{F}}_{L})$.
2. Two perfect dictionaries (see Section 3.3), denotes as $\mathsf{A}_\ell, \mathsf{B}_\ell$, each of capacity $2^{\ell+1} + O(\log N \cdot \log\lambda)$. Each dictionary holds elements of the form $(\mathsf{addr}, \mathsf{data})$ where $\mathsf{addr} \in [N]$, $\mathsf{data} \in \{0,1\}^w$.
3. Pointers $(\mathsf{A}_\ell,\ldots,\mathsf{A}_L)$, $(\mathsf{B}_\ell,\ldots,\mathsf{B}_L)$ where each A_i points to either $\{\mathsf{A}^{\mathsf{HF}}_i, \mathsf{A}^{\mathsf{F}}_i, \mathtt{Null}\}$ and each B_i to $\{\mathsf{B}^{\mathsf{HF}}_i, \mathsf{B}^{\mathsf{F}}_i, \mathtt{Null}\}$, where $\mathtt{Null}$ is a null pointer.
4. A global counter ctr, initialized as 0.

Construction 5.2: Oblivious RAM $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$

- **Input:** $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w$.
- **Secret state:** As above.
- **Initialization:** ctr is initialized to 0 as above, and all other data structures are initialized as empty.
- **The algorithm:**

Lookup:

```
1. Initialize $\mathsf{found} = \mathsf{false}$, $\mathsf{data}^* = \perp$.
2. Perform $\mathsf{fetched} := \mathsf{A}_\ell.\mathsf{PopKey}(\mathsf{addr})$.
3. If $\mathsf{fetched} \neq \perp$: then $\mathsf{B}_\ell.\mathsf{PopKey}(\perp)$.
   Otherwise, $\mathsf{fetched} := \mathsf{B}_\ell.\mathsf{PopKey}(\mathsf{addr})$.
4. If $\mathsf{fetched} \neq \perp$: set $\mathsf{found} = \mathsf{true}$.
5. For each $i \in \{\ell+1,\ldots,L\}$ in increasing order, do (if $\mathsf{A}_i$ (or $\mathsf{B}_i$ resp.) is $\mathtt{Null}$, then let
   the result of $\mathsf{Lookup}$ be $\perp$):
    (a) If $\mathsf{found} = \mathsf{false}$:
        i. Set $\mathsf{fetched} := \mathsf{A}_i.\mathsf{Lookup}(\mathsf{addr})$.
        ii. If $\mathsf{fetched} \neq \perp$ then set $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
    (b) Else, perform $\mathsf{A}_i.\mathsf{Lookup}(\perp)$.
    (c) If $\mathsf{found} = \mathsf{false}$:
        i. Set $\mathsf{fetched} := \mathsf{B}_i.\mathsf{Lookup}(\mathsf{addr})$.
        ii. If $\mathsf{fetched} \neq \perp$ then set $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
    (d) Else, perform $\mathsf{B}_i.\mathsf{Lookup}(\perp)$.
Write back:
 6. If $\mathsf{found} = \mathsf{false}$, i.e., this is the first time $\mathsf{addr}$ is being accessed, set $\mathsf{data}^* = 0$.
 7. Let $(k, v) := (\mathsf{addr}, \mathsf{data}^*)$ if this is a $\mathsf{read}$ operation; else let $(k, v) := (\mathsf{addr}, \mathsf{data})$.
 8. Insert $(k, v)$ into $\mathsf{A}_\ell$ and $\mathsf{B}_\ell$ using $\mathsf{Insert}(k, \mathsf{ctr} \bmod 2^{\ell+1}, v)$.
Rebuild:
 9. Increment $\mathsf{ctr}$ by 1.
10. For $i \in \{\ell + 1, \ldots, L\}$:
    (a) If $\mathsf{ctr} \equiv 2^{i-2} \bmod 2^i$ then continue to 1-out-of-4 case:
```

If $\mathsf{ctr} \equiv$	$0 \bmod 2^i$	$2^{i-2} \bmod 2^i$	$2 \cdot 2^{i-2} \bmod 2^i$	$3 \cdot 2^{i-2} \bmod 2^i$
Set $\mathsf{A}_i :=$	`Null`	$\mathsf{A}_i^{\mathsf{HF}}$	`Null`	$\mathsf{A}_i^{\mathsf{F}}$
Set $\mathsf{B}_i :=$	$\mathsf{B}_i^{\mathsf{F}}$	$\mathsf{B}_i^{\mathsf{F}}$	$\mathsf{B}_i^{\mathsf{HF}}$	`Null`
Start	$\mathsf{RebuildHF}(\mathsf{A}_i^{\mathsf{HF}})$	$\mathsf{RebuildHF}(\mathsf{B}_i^{\mathsf{HF}})$	$\mathsf{RebuildF}(\mathsf{A}_i^{\mathsf{F}})$	$\mathsf{RebuildF}(\mathsf{B}_i^{\mathsf{F}})$

```
    By starting a task we mean to add the relevant task into the list Tasks. The procedures
    RebuildHF and RebuildF are defined below.
11. In a round robin fashion, for each task $\mathsf{t} \in \mathsf{Tasks}$, execute $\mathsf{t.eachEpoch}$ steps.
12. Return $v$.
```

Before proceeding, we refer the reader to depictions of the rebuilding scheduling in Figures 2 and 3. In Step 10a, the schedule of the rebuild tasks is asymmetric ($\mathsf{A}_i^{\mathsf{HF}}$ and $\mathsf{A}_i^{\mathsf{F}}$ always start earlier than the B_i counterparts). This leads to the asymmetry between the setting of pointers A_i and B_i in Step 10a. Due to the asymmetry in schedule, there is a period such that both pointers A_i and B_i are available and storing distinct sets of elements (i.e., from $\mathsf{ctr} \equiv 2^{i-2} \bmod 2^i$ to $\mathsf{ctr} \equiv 2 \cdot 2^{i-2} \bmod 2^i$). Hence, Step 5 has to fetch addr in both A_i and B_i as their contents are distinct (our schedule is determinisitic, but fetching addr in both tables is necessary as we do not know which table stores addr).

The procedure RebuildF. In this procedure, we build the table $\mathsf{A}_i^{\mathsf{F}}$ from the two tables $\mathsf{A}_{i-1}^{\mathsf{F}}$ and $\mathsf{A}_i^{\mathsf{HF}}$ (similarly, $\mathsf{B}_i^{\mathsf{F}}$ from $\mathsf{B}_{i-1}^{\mathsf{F}}$ and $\mathsf{B}_i^{\mathsf{HF}}$). This is done by extracting the two tables, running Dedup (see Section 4) on the two tables, and then building the hash table. All those operations take linear work, and therefore we can spend $O(1)$ time per Access to the ORAM and finish the task in linear time. This is formalized in the $\mathsf{eachEpoch}$ variable.

In case the level to be rebuilt is $\ell + 1$, we extract all elements from the dictionary of level ℓ. This takes $O(\mathsf{poly} \log \log N)$ per element. This will also be the $\mathsf{eachEpoch}$ value. That is, we spend $O(\mathsf{poly} \log \log N)$ work for the rebuilding of level $\ell + 1$ with each Access to the ORAM.

$\mathsf{RebuildF}(\mathsf{C}_i^{\mathsf{F}})$:

- **Input:** The task has input $\mathsf{C}_i^{\mathsf{F}} \in \{\mathsf{A}_i^{\mathsf{F}}, \mathsf{B}_i^{\mathsf{F}}\}$ for some index $i \in \{\ell + 1, \ldots, L\}$.

- $\mathsf{eachEpoch}$: The total time allocated to this task is 2^{i-2}.
  1. If $i = \ell + 1$: Let $W \in O(2^i \cdot \mathsf{poly}(\log\log N))$ bound the total work of this procedure. Set $\mathsf{eachEpoch} = W/2^{i-2} = \mathsf{poly}\log\log N$.[5]
  2. If $i > \ell + 1$: The total work is $W \in O(2^i)$. Set $\mathsf{eachEpoch} = W/2^{i-2} \in O(1)$.
- **The task:**
  1. If $i = \ell + 1$, run $\mathsf{C}_{i-1}.\mathsf{PopTime}(0, 2^\ell - 1)$ repeatedly for 2^ℓ times. That is, we extract all elements with $\mathsf{ctr} \bmod 2^{\ell+1} \in [0, 2^\ell - 1]$, i.e., all elements that were added to the dictionary while building $\mathsf{A}_i^{\mathsf{HF}}$ and $\mathsf{B}_i^{\mathsf{HF}}$. Let X be the list of popped elements, and then obliviously shuffle X. Run $Y := \mathsf{C}_i^{\mathsf{HF}}.\mathsf{Extract}()$.
  2. Else $i > \ell + 1$, run $X := \mathsf{C}_{i-1}^{\mathsf{F}}.\mathsf{Extract}()$ and $Y := \mathsf{C}_i^{\mathsf{HF}}.\mathsf{Extract}()$.
  3. Run $Z := \mathsf{Dedup}(X, Y)$.
  4. Run $\mathsf{C}_i^{\mathsf{F}} := \mathsf{HT}.\mathsf{Build}(Z)$.

The procedure RebuildHF. In this procedure, we rebuild table $\mathsf{A}_i^{\mathsf{HF}}$ from the contents of the table $\mathsf{A}_{i-1}^{\mathsf{F}}$ (or $\mathsf{B}_i^{\mathsf{HF}}$ from $\mathsf{B}_{i-1}^{\mathsf{F}}$). This is performed by adding dummy elements and building the next level. For $i > \ell + 1$ this requires linear work, and therefore we can spend $O(1)$ time per Access to the ORAM and finish the task in linear time. Likewise the case of $\mathsf{RebuildF}$, the level $\ell + 1$ requires some more work but we have to finish also in linear time, so we spend more work with each access to the ORAM.

For the case of $i = L$ we do not simply build $\mathsf{A}_L^{\mathsf{HF}}$ from $\mathsf{A}_{L-1}^{\mathsf{F}}$. Instead, we also have to merge the contents on $\mathsf{A}_L^{\mathsf{F}}$ and $\mathsf{A}_{L-1}^{\mathsf{F}}$ into $\mathsf{A}_L^{\mathsf{HF}}$. This is performed similarly to $\mathsf{RebuildF}$: We first extract the two levels, run deduplication, and build level L.

RebuildHF($\mathsf{C}_i^{\mathsf{HF}}$):

- **Input:** The task gets as input a table $\mathsf{C}_i^{\mathsf{HF}} \in \{\mathsf{A}_i^{\mathsf{HF}}, \mathsf{B}_i^{\mathsf{HF}}\}$ for some index $i \in \{\ell + 1, \ldots, L\}$.
- $\mathsf{eachEpoch}$: The total time allocated to this task is 2^{i-2}.
  1. If $i = \ell + 1$: Let $W \in O(2^i \cdot \mathsf{poly}(\log\log N))$ bound the total work of this procedure. Set $\mathsf{eachEpoch} = W/2^{i-2} = \mathsf{poly}\log\log N$.
  2. If $i > \ell + 1$: The total work is $W \in O(2^i)$. Set $\mathsf{eachEpoch} = W/2^{i-2} \in O(1)$.
- **The task:**
  1. If $i = L$:
     (a) Run $X := \mathsf{C}_{L-1}^{\mathsf{F}}.\mathsf{Extract}()$ and $Y := \mathsf{C}_L^{\mathsf{F}}.\mathsf{Extract}()$.
     (b) Run $Z := \mathsf{Dedup}(X, Y)$.
     (c) Run $\mathsf{C}_L^{\mathsf{HF}} := \mathsf{HT}.\mathsf{Build}(Z)$.
  2. Otherwise:
     (a) If $i = \ell+1$, run $\mathsf{C}_{i-1}.\mathsf{PopTime}(2^\ell, 2^{\ell+1}-1)$ repeatedly for 2^ℓ times. That is, we extract all elements with $\mathsf{ctr} \bmod 2^{\ell+1} \in [2^\ell, 2^{\ell+1} - 1]$, i.e., all elements that were added to the dictionary while building $\mathsf{A}_i^{\mathsf{F}}$ and $\mathsf{B}_i^{\mathsf{F}}$. Let X be the list of popped elements, and then obliviously shuffle X.
     (b) Else $i > \ell + 1$, run $X := \mathsf{C}_{i-1}^{\mathsf{F}}.\mathsf{Extract}()$.
     (c) Initialize an array Y of 2^{i-1} dummies.
     (d) Intersperse X and Y into Z and run $\mathsf{C}_i^{\mathsf{HF}}.\mathsf{Build}(Z)$.

[5]Note that this implies that we run $\mathsf{poly}\log\log N$ work per each access for the first level.

5.1.1 Analysis

We next prove the following theorem:

Theorem 5.3. *Let N be the capacity of the ORAM and let $\lambda \in \mathbb{N}$ be a security parameter. Assuming the existence of* HT *as in Assumption 5.1, Construction 5.2 obliviously implements the ORAM functionality, and each* Access *takes* $O(\log N + \log^4 \log \lambda)$ *in the worst case.*

Proof. We start with the efficiency analysis. Each access requires two lookups (PopKey) at the dictionaries $\mathsf{A}_\ell, \mathsf{B}_\ell$ (Steps 2 and 3) and writing back to the two dictionaries (Step 8). Each dictionary contains at most $2^\ell \leq \log^{12} \lambda$, and each access costs $O(\log^4 \log \lambda)$ time (see Section 3.3).

Then, we perform one access to each one of the tables $\mathsf{A}_{\ell+1}, \ldots, \mathsf{A}_L$, $\mathsf{B}_{\ell+1}, \ldots, \mathsf{B}_L$, each takes $O(1)$ time by Assumption 5.1, and overall it takes $O(\log N)$ times. So overall, the lookup and write back take $O(\log N + \log^4 \log \lambda)$ work.

In the rebuild process, by construction we have exactly one task being rebuilt in each level, and start the next task only when the previous one finishes. It is easy to see that each process takes a linear time in the size of the level, and therefore we spend $O(1)$ per task with each Access to the ORAM, except for level $\ell + 1$. The procedures for level $\ell + 1$ require 2^ℓ accesses to the dictionaries, each translates to $O(\log^4 \log \lambda)$ work (total $O(2^\ell \cdot \log^4 \log \lambda)$, and oblivious shuffle of a list of size 2^ℓ which can also be implemented in total $O(2^\ell \cdot \log^4 \log \lambda)$. Therefore, we spend $O(\log N + \log^4 \log \lambda)$ work for the rebuilding of all levels, combined.

Security. Since the ORAM functionality is deterministic, it is enough to separately consider correctness and obliviousness. We show here obliviousness, and then discuss correctness.

We show security in the hybrid model where we invoke $\mathcal{F}_{\mathsf{2KeyDict}}$, $\mathcal{F}_{\mathsf{HT}}$, $\mathcal{F}_{\mathsf{Dedup}}$, $\mathcal{F}_{\mathsf{Shuffle}}$ instead of oblivious dictionary, oblivious hash table, oblivious deduplication and intersperse, respectively. Replacing all ideal functionalities with the corresponding construction is straightforward using the composition theorem.

It is easy to simulate Construction 5.2: We access the two dictionaries, and then access the two hash tables in each level and finally write back to the dictionaries. The rebuild process and which hash table we use has a public schedule known to the adversary. Likewise which tasks are currently running.

We now show how to simulate the two procedures: $\mathsf{RebuildF}$ and $\mathsf{RebuildHF}$. For the case of $i > \ell + 1$, in $\mathsf{RebuildF}$: We just have two ideal calls to $\mathsf{Extract}$. Since $\mathsf{Extract}$ returns an oblivious permutation of the element in the hash table, this implies that the input assumption of Dedup is preserved. We then obtain an array of size 2^i which is randomly shuffled and therefore the input assumption of $\mathcal{F}_{\mathsf{HT}}$ is preserved. Simulation is just these three ideal calls. Simulating $\mathsf{RebuildHF}$ is similar for the case of $i = L$, and for the case of $i \in \{\ell + 2, \ldots, L - 1\}$ it is also just ideal calls to $\mathcal{F}_{\mathsf{HT}}.\mathsf{Extract}$, $\mathcal{F}_{\mathsf{Shuffle}}$ (to intersperse the two arrays) and $\mathcal{F}_{\mathsf{HT}}.\mathsf{Build}$. As for $i = \ell + 1$, in both procedures we have ideal calls to $\mathcal{F}_{\mathsf{2KeyDict}}$ and we shuffle the output so that input assumptions are preserved.

Correctness. We also prove the correctness in the hybrid model, and our goal is to show that every Access to an address addr reads the data that was most recently written to addr, i.e., satisfying the ORAM functionalities: Each $\mathsf{Access}(\mathsf{read}, \mathsf{addr}, \perp)$ for a given addr will have the answer data, according to the last operation $\mathsf{Access}(\mathsf{write}, \mathsf{addr}, \mathsf{data})$ that was given to the ORAM (with the same addr). We begin with describing two invariants in Definitions 5.4 and 5.5 and show that they hold.

Definition 5.4 (Vertical invariant)**.** *Fixing any* $\mathsf{addr} \in [N]$*, we say that* $(\mathsf{addr}, \mathsf{data})$ *is the* freshest version *at some given time* ctr *if the pair* $(\mathsf{addr}, \mathsf{data})$ *is the most recent pair having* addr *read or written by* Access *operation to the ORAM. Then, for every* $\mathsf{addr} \in [N]$*, it holds that*

- *every level in the hierarchy* $H_{\mathsf{A}} := \{\mathsf{A}_i\}_{i \in [\ell, L]}$ *consists of at most one version of* addr*, and*
- *Among all levels in* $\{\mathsf{A}_i^{\mathsf{HF}}, \mathsf{A}_i^{\mathsf{F}}\}_{i \in [\ell, L]}$ *that contain* addr *(in which some might be rebuilt and unavailable), the freshest version of* addr *must reside in the smallest level.*

This holds symmetrically for hierarchy $H_{\mathsf{B}} := \{\mathsf{B}_i\}_{i \in [\ell, L]}$*.*

Definition 5.5 (Horizontal invariant)**.** *For every* $\mathsf{addr} \in [N]$*, it holds that the freshest version of* addr *must fall in one of the following cases:*

1. *It is in the same level of the two hierarchies, i.e., it is in both* A_i *and* B_i *for some* $i \in [\ell, L]$*.*
2. *It is in* $\mathsf{RebuildF}(\mathsf{A}_i^{\mathsf{F}})$ *and* $\mathsf{B}_i^{\mathsf{HF}}$ *for some* $i \in [\ell+1, L]$*, and* $\mathsf{B}_i = \mathsf{B}_i^{\mathsf{HF}}$*.*
3. *It is in* $\mathsf{A}_i^{\mathsf{F}}$ *and* $\mathsf{RebuildF}(\mathsf{B}_i^{\mathsf{F}})$ *for some* $i \in [\ell+1, L]$*, and* $\mathsf{A}_i = \mathsf{A}_i^{\mathsf{F}}$*.*
4. *It is in* B_i *and either in* $\mathsf{RebuildHF}(\mathsf{A}_{i+1}^{\mathsf{HF}})$ *or in* $\mathsf{RebuildF}(\mathsf{A}_{i+1}^{\mathsf{F}})$ *for some* $i \in [\ell, L-1]$*.*
5. *It is in* A_{i+1} *and either in* $\mathsf{RebuildHF}(\mathsf{B}_{i+1}^{\mathsf{HF}})$ *or in* $\mathsf{RebuildF}(\mathsf{B}_{i+1}^{\mathsf{F}})$ *for some* $i \in [\ell, L-1]$*.*

The invariants imply correctness. Using the above vertical and horizontal invariants (whose proofs are below), it suffices to syntactically check the correctness: we list all possible locations of the newest version below and conclude the correctness of Access.

- In both A_ℓ and B_ℓ: the element from A_ℓ is outputted (following Step 2).
- In B_ℓ while $\mathsf{A}_{\ell+1}$ is rebuilding ($\mathsf{A}_{\ell+1}$ takes elements from A_ℓ but $\mathsf{A}_{\ell+1}$ is still unavailable): the element from B_ℓ is outputted (following Step 3).
- In $\mathsf{A}_{\ell+1}$ while $\mathsf{B}_{\ell+1}$ is rebuilding ($\mathsf{B}_{\ell+1}$ takes elements from B_ℓ but $\mathsf{B}_{\ell+1}$ is still unavailable): the element from $\mathsf{A}_{\ell+1}$ is outputted (Step 5(a)i).
- In both A_i and B_i, $i \in [\ell+1, L]$: the element from A_i is outputted (Step 5(a)i).
- In B_i while A_{i+1} is rebuilding down, for $i \in [\ell+1, L-1]$: there are two cases, either A_{i-1} has finished its rebuild down to A_i, or A_{i-1} has not yet. In both cases, the element from B_i is outputted by Step 5(c)i.
- In A_{i+1} while B_i is rebuilding down, $i \in [\ell+1, L-1]$: the element from A_{i+1} is outputted (Step 5(a)i).

Notice that for two addresses $\mathsf{addr}, \mathsf{addr}'$, it may happen that addr is in Case 4 for some i and that addr' is in Case 5 for $i' = i-1$. This means addr is in B_i while addr' is in A_i, so that both A_i and B_i are available for lookup, but they have disjoint contents addr and addr'. This special case explains the reason we perform lookup on both A_i and B_i in Steps 5(a)i and 5(c)i.

Lemma 5.6. *The vertical invariant defined above (Definition 5.4) holds in the construction.*

Proof. We prove the invariant for H_{A} and H_{B} follows symmetrically. In A_ℓ, each addr can appear at most once by the functionality of 3-key dictionary (Functionality 3.3). For each A_i, $i \in [\ell+1, L]$, it is created by either $\mathsf{RebuildF}(\mathsf{A}_i^{\mathsf{F}})$ or $\mathsf{RebuildHF}(\mathsf{A}_i^{\mathsf{HF}})$, and then A_i is next set to $\mathsf{A}_i^{\mathsf{F}}$ or $\mathsf{A}_i^{\mathsf{HF}}$ respectively. In the former case, it holds as addr appeared at most once in each of $\mathsf{A}_{i-1}^{\mathsf{F}}$ and $\mathsf{A}_i^{\mathsf{HF}}$ inductively, and then the deduplication leaves only one version (i.e., Step 3 in $\mathsf{RebuildF}$). In the latter case, each addr appears at most once in $\mathsf{A}_i^{\mathsf{HF}}$ as it was at most once in $\mathsf{A}_{i-1}^{\mathsf{F}}$ inductively.

The claim of the freshest version is argued in the following steps.

1. A *fresher* version must be in a smaller level than an older one. It is because the fresher is the more recent, and thus the fresher one is processed by a less number of rebuilds by the rebuilding schedule. Since each rebuild either moves the version down (in $\mathsf{RebuildF}$ or $\mathsf{RebuildHF}$) or keeps it in the same level (only in $\mathsf{RebuildHF}$), the fresher one is in a smaller level.
2. We always *discard* the older version whenever two versions meet. Observe that this happens only in two cases: a) the fresher version overwrites the older in dictionary A_ℓ or B_ℓ when inserting the fresher one (Step 8), b) the two versions reside in each of $(\mathsf{A}^{\mathsf{F}}_{i-1}, \mathsf{A}^{\mathsf{HF}}_{i})$ and the two levels are going to be merged where we applied Dedup and discarded the older from A_{i+1} (Step 3 in $\mathsf{RebuildF}$).
3. The freshest version was inserted into A_ℓ when it was Accessed (Step 8), and thus it resides in H_{A} as we discard only older versions. Putting together, the freshest is always in H_{A} and in a smaller level (compared to any older versions).

□

Lemma 5.7. *The horizontal invariant defined above (Definition 5.5) holds in the construction.*

Proof. Case 1 holds for $i = \ell$ as the element addr was inserted into both $(\mathsf{A}_\ell, \mathsf{B}_\ell)$ in the same time ctr. For $i = \ell + 1$, we show when the elements move to either $(\mathsf{A}^{\mathsf{HF}}_{\ell+1}, \mathsf{B}^{\mathsf{HF}}_{\ell+1})$ or $(\mathsf{A}^{\mathsf{F}}_{\ell+1}, \mathsf{B}^{\mathsf{F}}_{\ell+1})$.

1. If the element was inserted to $(\mathsf{A}_\ell, \mathsf{B}_\ell)$ at time $\mathsf{ctr} \in [0, 2 \cdot 2^{\ell-1} \bmod 2^{\ell+1})$ then $\mathsf{RebuildHF}(\mathsf{A}^{\mathsf{HF}}_{\ell+1})$ and $\mathsf{RebuildHF}(\mathsf{B}^{\mathsf{HF}}_{\ell+1})$ that occur at time $[0, 2 \cdot 2^{\ell-1} \bmod 2^{\ell+1})$ will ignore it (as $\mathsf{RebuildHF}$ does not pull elements that are not old enough in the dictionary). The element will be pushed into levels $\mathsf{A}_{\ell+1} := \mathsf{A}^{\mathsf{F}}_{\ell+1}$ and $\mathsf{B}_{\ell+1} := \mathsf{B}^{\mathsf{F}}_{\ell+1}$ at time $0 \bmod 2^{\ell+1}$ (by $\mathsf{RebuildF}$).
2. If the element was insert to $(\mathsf{A}_\ell, \mathsf{B}_\ell)$ at time $\mathsf{ctr} \in [2 \cdot 2^{\ell-1} \bmod 2^{\ell+1}, 4 \cdot 2^{\ell-1} \bmod 2^{\ell+1})$ then $\mathsf{RebuildF}(\mathsf{A}^{\mathsf{F}}_{i})$ and $\mathsf{RebuildF}(\mathsf{B}^{\mathsf{F}}_{i})$ will ignore those elements and those will be pushed into levels $\mathsf{A}_{\ell+1} := \mathsf{A}^{\mathsf{HF}}_{\ell+1}$ and $\mathsf{B}_{\ell+1} := \mathsf{B}^{\mathsf{HF}}_{\ell+1}$ at time $2 \cdot 2^{\ell-1} \bmod 2^{\ell+1}$ (by $\mathsf{RebuildHF}$).

Once the elements are in level $\ell+1$, we can apply a similar argument for moving it into levels $\ell+2$ and then into all levels $\ell+2, \ldots, L$ inductively.

Case 2 happens right after addr was moved into $(\mathsf{A}^{\mathsf{HF}}_{i}, \mathsf{B}^{\mathsf{HF}}_{i})$. It follows by the third case in Step 10a (i.e., when $\mathsf{ctr} \in [2 \cdot 2^{i-2} \bmod 2^i, 3 \cdot 2^{i-2} \bmod 2^i)$).

Case 3 happens right after Case 2, and follows by the fourth case in Step 10a (i.e., $\mathsf{ctr} \in [3 \cdot 2^{i-2} \bmod 2^i, 4 \cdot 2^{i-2} \bmod 2^i)$).

Case 4 follows after Case 3, when we had $\mathsf{A}_i = \mathsf{A}^{\mathsf{F}}_{i}$ and $\mathsf{B}_i = \mathsf{B}^{\mathsf{F}}_{i}$, and addr was in both A_i and B_i. Depending on ctr, addr is then moved into level $i+1$ by either $\mathsf{RebuildHF}(\mathsf{A}^{\mathsf{HF}}_{i+1})$ or $\mathsf{RebuildF}(\mathsf{A}^{\mathsf{F}}_{i+1})$.

Case 5 happens right after Case 4 as $\mathsf{RebuildHF}(\mathsf{B}^{\mathsf{HF}}_{i+1})$ or $\mathsf{RebuildF}(\mathsf{B}^{\mathsf{F}}_{i+1})$ continues to move addr down into level $i+1$. □

This concludes the proof of Theorem 5.3. □

Moreover, in Appendix C we also use the combined stash technique and show how to deamortize it as well. We show:

Theorem 5.8 (Restatement of Theorem C.3)**.** *Let N be the capacity of the ORAM and let $\lambda \in \mathbb{N}$ be a security parameter. Assuming the existence of one-way functions, the construction described above obliviously implements the ORAM functionality. The construction has $O(\log N + \log^4 \log \lambda)$ worst case overhead.*

Acknowledgements

This work is supported in part by a DARPA Brandeis award, by NSF under the award numbers CNS-1601879, CNS-2044679, by Packard Fellowship, an ONR YIP award, by the Israel Science Foundation (grants No. 2439/20 and 1774/20), by an Alon Young Faculty Fellowship, and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 891234.

References

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *ACM STOC*, pages 1–9, 1983.

[2] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: optimal oblivious RAM. In *Advances in Cryptology - EUROCRYPT*, pages 403–432, 2020.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Optimal oblivious parallel RAM. *IACR ePrint Arch.*, 2020:1292, 2020.

[4] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *ACM CCS*, pages 837–849, 2015.

[5] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ITCS*, pages 357–368, 2016.

[6] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.

[7] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *ASIACRYPT*, pages 660–690, 2017.

[8] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In *TCC*, pages 636–668, 2018.

[9] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In *TCC*, pages 72–107, 2017.

[10] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *ASIACRYPT*, 2014.

[11] Samuel Dittmer and Rafail Ostrovsky. Oblivious tight compaction in $o(n)$ time with smaller constant. In *SCN*, pages 253–274, 2020.

[12] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution. In *Advances in Cryptology - EUROCRYPT*, pages 338–369, 2021.

[13] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *ACM workshop on Scalable trusted computing*, pages 3–8, 2012.

[14] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In *ASPLOS*, pages 103–116, 2015.

[15] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[16] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. In *CANS*, pages 172–191, 2015.

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

[18] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

[19] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587, 2011.

[20] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 95–100, 2011.

[21] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167, 2012.

[22] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, pages 1353–1364, 2016.

[23] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS*, 2012.

[24] Ilan Komargodski and Wei-Kai Lin. A logarithmic lower bound for oblivious RAM (for all parameters). In *Advances in Cryptology - CRYPTO*, pages 579–609, 2021.

[25] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[26] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *CRYPTO*, pages 523–542, 2018.

[27] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *IEEE S&P*, 2015.

[28] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396, 2013.

[29] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In *ACM CCS*, pages 311–324, 2013.

[30] Rafail Ostrovsky and Victor Shoup. Private information storage. In *ACM STOC*, pages 294–303, 1997.

[31] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. In *IEEE FOCS*, 2018.

[32] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *The 40th Annual International Symposium on Computer Architecture, ISCA*, pages 571–582, 2013.

[33] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[34] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S&P*, pages 253–267, 2013.

[35] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[36] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS*, pages 299–310, 2013.

[37] Mikkel Thorup. Randomized sorting in o(n log log n) time and linear space using addition, shift, and bit-wise boolean operations. *J. Algorithms*, 42(2):205–230, 2002.

[38] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *ACM CCS*, pages 850–861, 2015.

[39] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *ACM CCS*, pages 191–202, 2014.

[40] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *ACM CCS*, 2012.

[41] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE S&P*, pages 218–234, 2016.

[42] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX*, pages 707–720, 2016.

A CombHT: Construction of Theorem 3.6

$\mathsf{Build}(\mathbf{I})$. Build receives as input an array $\mathbf{I} = (a_1, \ldots, a_n)$ containing n elements, where each a_i is either real or dummy. The real elements are a key-value pair. The input assumption is that $\mathbf{I}$ is randomly shuffled with some permutation that is hidden from the adversary, and it is assumed that each real key appears at most once. The Build algorithm is as follows:

1. Let $\mu := \log^9 \lambda$, which is the size of each bin. Let $\epsilon := 1/\log^2 \lambda$, which is the ratio of elements that will be moved to the overflow pile. (The overflow pile is just a subset of arbitrary elements – this is a beautiful trick originated in PanORAMa, see also OptORAMa [2, Section 2.1.2].) Let $B := \lceil n/\mu \rceil$ be the number of bins.
2. Sample a random PRF secret key sk.
3. Directly hash all elements into the B bins. That is, for each real element $a_i = (k_i, v_i)$, place a_i in the bin $\mathsf{PRF}_{\mathsf{sk}}(k_i)$. If $a_i = \mathsf{dummy}$, then throw it uniformly at random. Let $\mathsf{Bin}_1, \ldots, \mathsf{Bin}_B$ denote the bins, and let $(n_1, \ldots, n_B)$ denote the loads of the bins.
   Note that the access pattern of this step reveals the loads $(n_1, \ldots, n_B)$ and therefore by itself it is not enough. What we do next is intended to hide the final load of each bin. .
4. Obliviously move $\epsilon \cdot n$ elements to an overflow pile. This is done via sampling secret loads by throwing $n' = n \cdot (1 - \epsilon)$ balls into B bins, resulting with secret loads $(L_1, \ldots, L_B)$. With all but negligible probability, for every $i \in [B]$ it holds that $L_i < n_i$. Obliviously move all elements in positions $L_i, \ldots, n_i$ into the overflow pile while replacing them with dummy balls in the bin. At the end of this process, the top elements in each bin are replaced with dummies, and exactly $\epsilon \cdot n$ elements are moved to the overflow pile. Crucially, the loads $(L_1, \ldots, L_B)$ are secret and not revealed during this process.
5. **Building the bins:**
   (a) Each bin is implemented using a (variant of) oblivious Cuckoo hash. This allows us to locate every element in a bin in using only two cell accesses in addition to a scan of a $O(\log \lambda)$ size stash. We denote the bins (without the stashes) $\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B$.
   (b) We combine all stashes from all bins to allow for faster access. We apply an oblivious Cuckoo hash on the combined stashes. This results in a combined table and a stash for that table, denoted $\mathsf{CombS} = (\mathsf{CombS}_\mathsf{T}, \mathsf{CombS}_\mathsf{S})$, where $\mathsf{CombS}_\mathsf{T}$ is the main table and $\mathsf{CombS}_\mathsf{S}$ is the stash (of size $O(\log \lambda)$).
6. **Building the overflow pile:**
   (a) The overflow pile is of size $n/\log^2 \lambda$ and we index it using an oblivious Cuckoo hash, resulting in $\mathsf{OF} = (\mathsf{OF}_\mathsf{T}, \mathsf{OF}_\mathsf{S})$, where OF_T is the table and OF_S is a $O(\log \lambda)$ size stash.
7. Finally, the output of this process is a secret key sk, bins $\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B$, a combined stash for all bins $\mathsf{CombS} = (\mathsf{CombS}_\mathsf{T}, \mathsf{CombS}_\mathsf{S})$, and an overflow pile $\mathsf{OF} = (\mathsf{OF}_\mathsf{T}, \mathsf{OF}_\mathsf{S})$.

$\mathsf{Lookup}(k)$: To lookup a key k in this table (where k is either real or dummy), we do the following: We first perform lookup on k in the overflow pile. If found, we perform a dummy access to a random major bin OBin_i (for $i \leftarrow [B]$). If not found, we perform lookup on k in the major bin OBin_i for $i = \mathsf{PRF}_{\mathsf{sk}}(k)$. A lookup in a major bin starts by a lookup in the combined stashes CombS and then in the corresponding bin. In a more detail, an execution of lookup on k starts by scanning the stash of the overflow pile OF_S and then two locations in its table OF_T. Then, before touching a major bin, we scan the stash $\mathsf{CombS}_\mathsf{S}$ and access two locations in $\mathsf{CombS}_\mathsf{T}$. Finally, we access two locations in the relevant bin OBin_i. Here is the pseudocode:

1. Perform lookup in the overflow pile OF. This amounts to:

   (a) Linearly scan the stash of the overflow pile, that is, OF_S, and look for an element with the key k.
   (b) Access two locations in the main table of the overflow pile, that is, OF_T.
   (c) If the element is found in OF, the rest of the lookup procedure will be done with dummy.
2. Perform lookup in the combined stashes CombS. This amounts to:
   (a) Linearly scan the stash of the combined stashes, that is, $\mathsf{CombS}_\mathsf{S}$, and look for an element with the key k.
   (b) Access two locations in the main table of the combined stashes, that is, $\mathsf{CombS}_\mathsf{T}$.
3. Access a single bin (a random bin if the element was found during the lookup in OF and the real bin otherwise) out of the major bins and access two locations there. (Recall that while visiting the random major bin, we perform a *dummy lookup* on the bin; see OptORAMa [2, Section 2.1.2] for example.)

Extract. In Extract we just return all elements that were not accessed in each one of the tables.

The above description is somewhat high-level and omits many details. For instance, how do we build the small bins in linear time using Cuckoo hashing? Those details are not important for our discussion and can be found in [2]. The resulting hash table is denoted as CombHT, and achieves the complexity described in Theorem 3.6.

B Deferred Proof of Theorem 4.1

We start with analyzing the efficiency of our scheme.

Efficiency. In step 3, we scan the stashes of T_2 and then for each element perform a $O(\log\lambda)$-time lookup, and then we shuffle L. Since the stashes are of size $O(\log\lambda)$, the running time of this step is $O(\log^2\lambda) \leq O(n)$. Step 5 consist of a linear scan of a list and then an $O(1)$ lookup on each item. Steps 4 and 6 consume $O(n)$ time. Step 7 consumes $O(n)$ time, as well. Overall, the overhead is linear in n, as needed.

Security. We claim that in the hybrid model (i.e., when consider $\mathcal{F}_{\mathsf{HT}}$ instead of HT, $\mathcal{F}_{\mathsf{Shuffle}}$ instead of Intersperse, etc.) then the access pattern is actually *deterministic*, and it consists of just calls to functionalities. Therefore, we can separately consider obliviousness and correctness. We start with the correctness, proceed to obliviousness. We start with the efficiency analysis, correctness, and finally prove obliviousness.

Correctness. We first argue that if a key k (with associated value v) appears only once overall, then it will appear in the output (associated with v). Assume that it appears only in X_1. After steps 1 and 2, it could either be in one of the stashes of T_1 or in one of the main tables. In both cases, during step 3 it will not be touched, and then after step 4 it will reside in S_1. The lookups during Step 5 will not affect this element and eventually it will reside in the output Z.

If k appears only in X_2 and not in X_1, then it will not be found in T_1 and thus it would not appear in S_1 in Step 4. Moreover, since it does not appear in T_1, it will not be accessed in Step 5 and thus it will appear in the output Z.

Next, we argue that if a key k appears in both input lists, then it will appear in the output just once. Moreover, if in X_1 it is associated with value v_1 and in X_2 it is associated with value v_2, then in the output the key k will be associated with v_1. Here, we need to analyze four cases:

1. The pair (k, v_1) ended up in one of the stashes of T_1 and the pair (k, v_2) ended up in one of the stashes of T_2: in step 3 the copy from T_2 will be marked "`accessed`" and the one from T_1 will be moved into L. The copy from T_2 will therefore not be extracted during step 6. Thus, (k, v_1) will appear in the final output (via interspersing L in Step 4 and then S_1 in Step 7), as needed while the copy in T_2 will be ignored.
2. The pair (k, v_1) ended up in one of the main tables of T_1 and the pair (k, v_2) ended up in one of stashes of T_2: this case is identical to Case 1.
3. The pair (k, v_1) ended up in one of the stashes of T_1 and the pair (k, v_2) ended up in one of the main tables of T_2: in step 3 the copy from T_1 remains unvisited and then it will be extracted during step 4. During step 5c we will be executing lookup in T_2 for this key k, and we will find it as k is indeed in the main tables of T_2, marking it "`accessed`". Therefore, the copy from T_2 will not be in the output of the Extract executed in step 6, and only v_1 will appear in the output.
4. The pair (k, v_1) ended up in one of the main tables of T_1 and the pair (k, v_2) ended up in one of the main tables of T_2: the element will be extracted during step 4 and then we will perform lookup for the key in the main tables of T_2 during step 5c. This lookup will succeed (as k is in the main tables of T_2) and the item will be marked as "`accessed`" in T_2 and will therefore not be extracted in step 6.

Obliviousness. The simulator works as follows, where all elements are always dummies:

- Step 1: invoking the Build simulator of HT twice. Step 2 is just notation.
- Step 3: linearly scan the arrays $\mathsf{OF}_{2,\mathsf{S}}$ and $\mathsf{Comb}_{2,\mathsf{S}}$ and after each step invoke the Lookup simulator of HT. Write the outputs to an array denoted L.
- Step 4: invoke the Extract simulator of HT, and run Intersperse ($\mathcal{F}_{\mathsf{Shuffle}}$) with L. The output is denoted S_1.
- Step 5: linearly scan the arrays S_1 and after each step invoke the Lookup simulator of HT modified so that it does not scan the stashes and directly accesses the main tables.
- Step 6: invoke the Extract simulator of HT. The output is denoted S_2 (which is all dummies).
- Step 7: sequentially invoke the simulator of Intersperse (with input $S_1 \| S_2$), then oblivious tight compaction, and IntersperseRD on the output.

The fact that the access pattern of the above simulator is indistinguishable from the one resulting from executing our algorithm Dedup follows directly by the obliviousness of each of the underlying building blocks. The only subtle point is that we use the simulator of the Lookup procedure of HT in a non black-box way by *not* scanning the stashes and directly accessing the main tables of T_2, i.e., Step 5. We satisfy the input assumption of HT (never lookup the same element twice and the input given to the build procedure is randomly shuffled), moreover, we emulate every Lookup(k) exactly according to the cases that k is found in $\mathsf{OF}_{2,\mathsf{S}}$, $\mathsf{CombS}_{2,\mathsf{S}}$, or not found.[6] Therefore, directly accessing the main tables preserves obliviousness: Step 5 and its simulated counterpart are almost identical to CombHT and its simulator, where the only difference is scanning the stashes or not.

C Deamortized ORAM assuming our Hash Table with a Stash

In Section 5.1 we presented our construction while assuming that we are given an oblivious hash table for shuffled inputs whose lookup time is $O(1)$. Unfortunately, we do not know how to realize

[6]This is important in order to avoid the attack of Falk, Noble, and Ostrovsky [12].

such a hash table as CombHT from Section 3.4 requires a scan of a $O(\log \lambda)$ size stash. If we use that hash table as is, the lookup time will be increased to $O(\log N \cdot \log \lambda)$ which we want to avoid. In [2] this extra $O(\log \lambda)$ factor was avoided by *merging* all the stashes into one larger stash and performing a lookup there once upon every access to the ORAM. Using this directly in our construction is problematic as in our deamortized construction some tables may need to be rebuilt while we need to still support accesses and so we cannot "lock" the combined stash.

We resolve this by implementing the stashes using four dictionaries $\mathsf{Stash}_\mathsf{A}^\mathsf{HF}$, $\mathsf{Stash}_\mathsf{A}^\mathsf{F}$, $\mathsf{Stash}_\mathsf{B}^\mathsf{HF}$, $\mathsf{Stash}_\mathsf{B}^\mathsf{F}$. The dictionary $\mathsf{Stash}_\mathsf{C}^\mathsf{X}$ for $\mathsf{C} \in \{\mathsf{A}, \mathsf{B}\}$ and $\mathsf{X} \in \{\mathsf{HF}, \mathsf{F}\}$ collects all the elements that go to the stash from levels in the C copy that correspond to the X instance. Each of these dictionaries will support extracting a given key from the smallest level where it resides or alternatively extracting an element from a given level. So, when we do lookup and the element is in the stash, we can quickly find to which table it corresponds, and when we need to do a rebuild for a given level, we can "collect" all the elements that belong to it from the corresponding stash. Abstracting the stashes as dictionaries is what enables us to support rebuilding intermediate levels while allowing accesses to the ORAM.

We briefly describe the differences from the ideal hash table:

1. Recall that Build works in linear time but also puts elements in a stash of size $O(\log \lambda)$. In our construction, we will put those elements in the corresponding dictionary (from the above four) depending on which table it came from along with an identifier for the level it corresponds to. For instance, an element that goes into the stash of level i in the HF instance of the left side will be put in stash $\mathsf{Stash}_\mathsf{A}^\mathsf{HF}$ along with the identifier i for the level.
2. Lookup visits $O(1)$ locations in the table, in addition to a scan of the stash. In our construction, we will not visit the stash and instead we will lookup in all four stashes. If we find the element in one of the stashes, we will simulate a standard lookup, while keeping in mind that we should have found it in the corresponding location.
3. $\mathsf{Extract}()$ of HT inside the procedures $\mathsf{RebuildHF}$ and $\mathsf{RebuildF}$ does not take the elements from its local stashes. Instead, it collects all its elements from the global stashes, i.e., the dictionaries above depending on the level, the side A or B, and the type HF, F). This is done by extracting them one by one (from an efficiency perspective, as we argue below, this is fine since the stash is rather small to begin with so we can tolerate this extra overhead while rebuilding).

Therefore, the change to the access pattern is just skipping the linear scan of the stashes inside each of HT. Instead, we access the global stashes which are modeled as oblivious dictionaries. That is, we search for the element in the (constant-many) global stashes in the very beginning of the lookup phase, and then pretend to look for the real element until we reach the hash table in which the element was found in the global stashes.

The four dictionaries $\mathsf{Stash}_\mathsf{C}^\mathsf{X}$ are formalized in the following functionality $\mathcal{F}_{\mathsf{LevelDict}}$. The syntax and construction is very similar to the two-key dictionary $\mathcal{F}_{\mathsf{2KeyDict}}$, but we remark that $\mathsf{Lookup}(k)$ does not remove the found element and that an element is only removed by $\mathsf{PopLevel}$. (This dictionary can be viewed as "syntax sugar" of $\mathcal{F}_{\mathsf{2KeyDict}}$.) We describe the functionality and then theorem statement for completeness, where the proof is very similar to that of Theorem 3.4 and is skipped.

Functionality C.1: $\mathcal{F}_{\mathsf{LevelDict}}$ - Leveled Dictionary Functionality

- Initialization of the state: let M be an empty 2-dimensional list indexed by $(k, \mathsf{level}) \in [K] \times [L]$ for the given key space K and level space L, where all $M[k, \mathsf{level}]$ are initialized as $\perp$.

- $\mathcal{F}_{\mathsf{LevelDict}}.\mathsf{Insert}(k, \mathsf{level}, \mathsf{whichStash}, \mathsf{data})$:
  - **Input:** a key k, a level level, flag $\mathsf{whichStash} \in \{\text{“}\mathtt{OF}\text{”}, \text{“}\mathtt{CombS}\text{”}\}$, and a value data, where k might be $\perp$, i.e., a dummy insertion.
  - **The procedure:**
    1. If $k \neq \perp$, set $M[k, \mathsf{level}] := (\mathsf{whichStash}, \mathsf{data})$.
  - **Output:** The Insert operation has no input.
- $\mathcal{F}_{\mathsf{LevelDict}}.\mathsf{Lookup}(k)$:
  - **Input:** a key k (that might be $\perp$, i.e., dummy).
  - **The procedure:**
    1. Let level be the smallest index such that $M[k, \mathsf{level}] \neq \perp$. If no such level exists, output $\perp$. Otherwise, set $(\mathsf{whichStash}^*, \mathsf{data}^*) := M[k, \mathsf{level}]$.
  - **Output:** The tuple $(\mathsf{level}, \mathsf{whichStash}^*, \mathsf{data}^*)$.
- $\mathcal{F}_{\mathsf{LevelDict}}.\mathsf{PopLevel}(\mathsf{level}, \mathsf{whichStash})$:
  - **Input:** a level level and a flag $\mathsf{whichStash}$.
  - **The procedure:**
    1. Let $k \in [K]$ be the smallest index such that $M[k, \mathsf{level}] = (\mathsf{whichStash}, \cdot)$. If no such k exists, set $\mathsf{data}^* := \perp$. Otherwise, set $\mathsf{data}^* := M[k, \mathsf{level}]$ and then set $M[k, \mathsf{level}] := \perp$.
  - **Output:** The tuple (k, data^*).

Theorem C.2. *Assume the tuple of* $(k, \mathsf{level}, \mathsf{whichStash}, \mathsf{data})$ *(i.e., key, level, flag, and data) can be stored in a constant number of memory words. Assume further that the leveled dictionary needs to support at most n elements. There exists a perfectly oblivious implementation of functionality* $\mathcal{F}_{\mathsf{LevelDict}}$ *such that each operation* Insert*,* Lookup*, and* $\mathsf{PopLevel}$ *takes* $O(\log^4 n)$ *time in the worst-case.*

We now formalize the changes we make into Construction 5.2. We employ the following **Lookup** procedure (Steps 1–5 in Construction 5.2), while we keep the other parts of **Write back** and **Rebuild** unchanged:
Lookup:

1. Initialize $\mathsf{found} = \mathsf{false}$, $\mathsf{data}^* = \perp$.
2. Perform $\mathsf{fetched}_{\mathsf{A}} := \mathsf{A}_\ell.\mathsf{PopKey}(\mathsf{addr})$. If $\mathsf{fetched}_{\mathsf{A}} \neq \perp$ then set $\mathsf{found} = \mathsf{true}$, $\mathsf{data}^* = \mathsf{fetched}_{\mathsf{A}}$ and perform $\mathsf{B}_\ell.\mathsf{PopKey}(\perp)$. Otherwise, $\mathsf{fetched}_{\mathsf{B}} := \mathsf{B}_\ell.\mathsf{PopKey}(\mathsf{addr})$.
3. If $\mathsf{fetched}_{\mathsf{B}} \neq \perp$ then set $\mathsf{found} = \mathsf{true}$ and set $\mathsf{data}^* = \mathsf{fetched}_{\mathsf{B}}$.
4. Perform
$$\begin{aligned} \mathsf{inStash}^{\mathsf{HF}}_{\mathsf{A}} &= \mathsf{Stash}^{\mathsf{HF}}_{\mathsf{A}}.\mathsf{Lookup}(k), \quad & \mathsf{inStash}^{\mathsf{F}}_{\mathsf{A}} &= \mathsf{Stash}^{\mathsf{F}}_{\mathsf{A}}.\mathsf{Lookup}(k), \\ \mathsf{inStash}^{\mathsf{HF}}_{\mathsf{B}} &= \mathsf{Stash}^{\mathsf{HF}}_{\mathsf{B}}.\mathsf{Lookup}(k), \quad & \mathsf{inStash}^{\mathsf{F}}_{\mathsf{B}} &= \mathsf{Stash}^{\mathsf{F}}_{\mathsf{B}}.\mathsf{Lookup}(k). \end{aligned}$$
5. For each $i \in \{\ell + 1, \ldots, L\}$ in increasing order, do:
   (a) If $\mathsf{found} = \mathsf{false}$:
      i. If $\mathsf{A}_i = \mathsf{A}^{\mathsf{HF}}_i$ and $\mathsf{inStash}^{\mathsf{HF}}_{\mathsf{A}}.\mathsf{level} = i$ then set $\mathsf{inStash} = \mathsf{inStash}^{\mathsf{HF}}_{\mathsf{A}}$.
      ii. If $\mathsf{A}_i = \mathsf{A}^{\mathsf{F}}_i$ and $\mathsf{inStash}^{\mathsf{F}}_{\mathsf{A}}.\mathsf{level} = i$ then set $\mathsf{inStash} = \mathsf{inStash}^{\mathsf{F}}_{\mathsf{A}}$.
      iii. Otherwise, set $\mathsf{inStash} = \perp$.
      iv. Set $\mathsf{fetched} := \mathsf{A}_i.\mathsf{Lookup}(\mathsf{addr})$ with the following modifications:

A. Do not scan the local stash of overflow pile. Instead, when the lookup visits $\mathsf{OF_S}$, if $\mathsf{inStash.whichStash}$ = "`OF`" then use $\mathsf{inStash.data}$ as found in OF. Otherwise, proceed as it was not found in $\mathsf{OF_S}$.
B. Do not scan the local stash of the CombS. Instead, when the lookup visit $\mathsf{CombS_S}$, if $\mathsf{inStash.whichStash}$ = "`CombS`" then use $\mathsf{inStash.data}$ as found in $\mathsf{CombS_S}$. Otherwise, proceed as it was not found in $\mathsf{CombS_S}$.

v. If $\mathsf{fetched} \neq \bot$ then set $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.

(b) Else, perform $\mathsf{A}_i.\mathsf{Lookup}(\bot)$.

(c) If $\mathsf{found} = \mathsf{false}$:

i. Repeat the same for $\mathsf{B}_i.\mathsf{Lookup}(\mathsf{addr})$ as in $\mathsf{A}_i, \mathsf{Lookup}(\mathsf{addr})$, while simulating the lookup in the stashes according to $\mathsf{inStash}^{\mathsf{HF}}_{\mathsf{B}}$ and $\mathsf{inStash}^{\mathsf{F}}_{\mathsf{B}}$.

ii. If $\mathsf{fetched} \neq \bot$ then set $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.

(d) Else, perform $\mathsf{B}_i.\mathsf{Lookup}(\bot)$.

We next explain the differences in the procedure $\mathsf{RebuildF}(\mathsf{C}^{\mathsf{F}}_i)$. Whenever we perform $\mathsf{Build}()$ to a hash table (Step 4) we perform the following:

1. Scan the $\mathsf{OF_S}$ stash, and for each element $(\mathsf{addr}, \mathsf{data})$ perform $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{Insert}(\mathsf{addr}, i, \text{"}\texttt{OF}\text{"}, \mathsf{data})$.
2. Scan the $\mathsf{CombS_S}$ stash, and for each element $(\mathsf{addr}, \mathsf{data})$ perform $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{Insert}(\mathsf{addr}, i, \text{"}\texttt{CombS}\text{"}, \mathsf{data})$.

Similarly, whenever we perform $\mathsf{C}^{\mathsf{F}}_{i-1}.\mathsf{Extract}()$ from a table (e.g., Step 2), we perform the following:

1. Call $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{PopLevel}(i-1, \text{"}\texttt{OF}\text{"})$ for $O(\log\lambda)$ times, obliviously shuffle the popped elements, and treat the shuffled list as $\mathsf{OF_S}$.
2. Call $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{PopLevel}(i-1, \text{"}\texttt{CombS}\text{"})$ for $O(\log\lambda)$ times, obliviously shuffle the popped elements, and treat the shuffled list as $\mathsf{CombS_S}$.

The case of $\mathsf{C}^{\mathsf{HF}}_i$ is defined analogously. The procedure $\mathsf{RebuildHF}$ is defined analogously as well.

Theorem C.3. *Let N be the capacity of the ORAM and let $\lambda \in \mathbb{N}$ be a security parameter. Assuming the existence of one-way functions, the construction described above obliviously implements the ORAM functionality. The construction has $O(\log N + \log^4 \log \lambda)$ worst case overhead.*

Proof. We need to argue correctness, security, and analyze efficiency. Let us start with efficiency where we argue that the worst case cost of every access remains logarithmic in the memory size. First, we observe that the cost of a lookup does not increase—this is true since we merely start a lookup by performing additional four accesses to an oblivious dictionary that holds up to $\mathsf{poly}\log\lambda$ elements. The cost of the latter is $\mathsf{poly}\log\log\lambda$ which does not affect the asymptotic overhead of a lookup. Analyzing the extra overhead incurred by the rebuild process is done in a similar way—every level that rebuilds causes an extract of at most $O(\log\lambda)$ elements from an appropriate stash and then adding so many elements back in. Tolerating this extra cost is not a problem since anyway the cost of build of every level is at least poly-logarithmic in λ (as this is the size of the smallest level). The only delicate point is that many levels may want to rebuild at the same time and we need to serialize their accesses to the dictionaries. Still, this is not something that affect the overhead since the extra "wait" incurred by the serialization is only a small poly-logarithmic in N and λ and our smallest table size is bigger than that.

For obliviousness, we first note that the ORAM functionality is deterministic and so it is enough to separately consider correctness and obliviousness. For security we argue that the observed access pattern is just a well-defined, deterministic, change from the construction where we assume a hash table without a stash (Section 5.1). Specifically, upon every access, instead of directly accessing the tables of the main ORAM, we first make an access to each of the four stashes $\mathsf{Stash}_{\mathsf{A}}^{\mathsf{HF}}$, $\mathsf{Stash}_{\mathsf{A}}^{\mathsf{F}}$, $\mathsf{Stash}_{\mathsf{B}}^{\mathsf{HF}}$, $\mathsf{Stash}_{\mathsf{B}}^{\mathsf{F}}$. Otherwise, the access pattern looks exactly the same and therefore obliviousness holds. Additionally, while doing rebuild of a given level, we first (before extract) pop all elements that "belong" to it from the appropriate stash and at the end (after build) put all stashed elements back into the appropriate global stash. This again uses the oblivious dictionaries in a black box way and does not change anything else and so obliviousness is maintained.

Lastly, we argue correctness. For this, we first imagine a hybrid (less efficient) construction where the stash of each table in each level sits next to it in the level and we access it only when accessing the appropriate table in the level. While this hybrid world is inefficient, we claim that if it satisfies correctness, then our construction above also satisfies correctness. Indeed, in terms of the accesses that are performed they are exactly the same—only the order of accesses changes when we access the stashes ahead of time once and for all levels together (this is because once we find an element we continue with doing dummy accesses and so it does not matter if an element appears multiple times). Therefore, to complete the proof, we need to prove correctness of the hybrid construction. This proof is an extension of the proof from Section 5.1. That is, we first note that an analogue of the vertical and horizontal invariants (Definitions 5.4 and 5.5) hold: instead of considering the elements of a given table we always consider the elements in the table together with the elements in the corresponding stash. Since the analogues of the two invariants hold (by the same reason they hold without the stashes), correctness of the scheme follows, as in Section 5.1. $\square$

D Figures

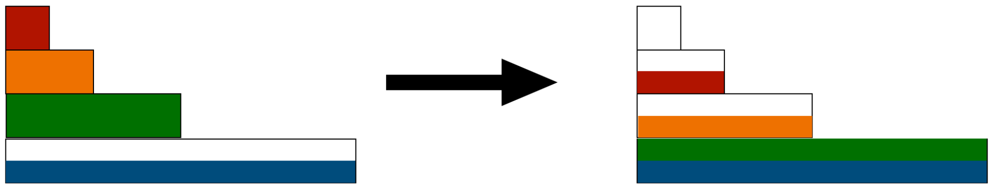


Figure 1: The rebuild process of [3]: The first three levels are "full" and the forth is the first level which is "half full". Each level is pushed down, while levels 3 and 4 are merged. After this operation, the first level is empty, two levels are "half full" and the last level is full.

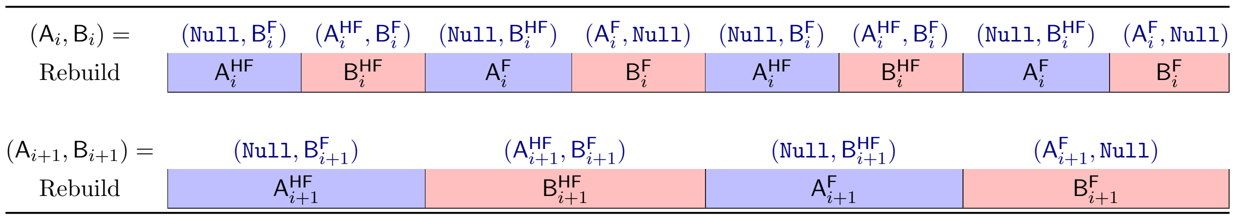
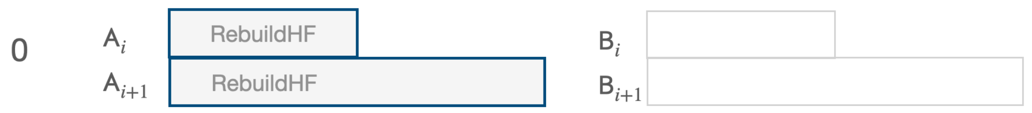
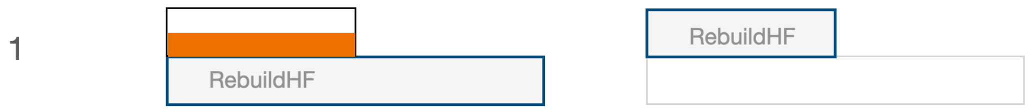

$(\mathsf{A}_i, \mathsf{B}_i) =$
$(\texttt{Null}, \mathsf{B}_i^{\mathsf{F}})$
$(\mathsf{A}_i^{\mathsf{HF}}, \mathsf{B}_i^{\mathsf{F}})$
$(\texttt{Null}, \mathsf{B}_i^{\mathsf{HF}})$
$(\mathsf{A}_i^{\mathsf{F}}, \texttt{Null})$
$(\texttt{Null}, \mathsf{B}_i^{\mathsf{F}})$
$(\mathsf{A}_i^{\mathsf{HF}}, \mathsf{B}_i^{\mathsf{F}})$
$(\texttt{Null}, \mathsf{B}_i^{\mathsf{HF}})$
$(\mathsf{A}_i^{\mathsf{F}}, \texttt{Null})$
Rebuild
$\mathsf{A}_i^{\mathsf{HF}}$
$\mathsf{B}_i^{\mathsf{HF}}$
$\mathsf{A}_i^{\mathsf{F}}$
$\mathsf{B}_i^{\mathsf{F}}$
$\mathsf{A}_i^{\mathsf{HF}}$
$\mathsf{B}_i^{\mathsf{HF}}$
$\mathsf{A}_i^{\mathsf{F}}$
$\mathsf{B}_i^{\mathsf{F}}$
$(\mathsf{A}_{i+1}, \mathsf{B}_{i+1}) =$
$(\texttt{Null}, \mathsf{B}_{i+1}^{\mathsf{F}})$
$(\mathsf{A}_{i+1}^{\mathsf{HF}}, \mathsf{B}_{i+1}^{\mathsf{F}})$
$(\texttt{Null}, \mathsf{B}_{i+1}^{\mathsf{HF}})$
$(\mathsf{A}_{i+1}^{\mathsf{F}}, \texttt{Null})$
Rebuild
$\mathsf{A}_{i+1}^{\mathsf{HF}}$
$\mathsf{B}_{i+1}^{\mathsf{HF}}$
$\mathsf{A}_{i+1}^{\mathsf{F}}$
$\mathsf{B}_{i+1}^{\mathsf{F}}$

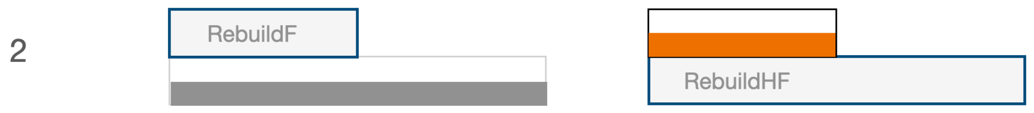
Figure 2: The Rebuild process (for levels i and $i+1$), demonstrating which table is being rebuilt at each stage and which tables we lookup in with each access. The timeline goes left-to-right, each colored box is rebuilding the enclosed table, and the left/right side of the box denotes the starting/ending time of the rebuild. Notice that the rebuild at level $i+1$ changes the status in both levels i and $i+1$, e.g., the starting of $\mathsf{B}_{i+1}^{\mathsf{F}}$ (on the bottom-right) switches both B_i and B_{i+1} to $\texttt{Null}$, and its ending assigns $\mathsf{B}_{i+1} := \mathsf{B}_{i+1}^{\mathsf{F}}$.



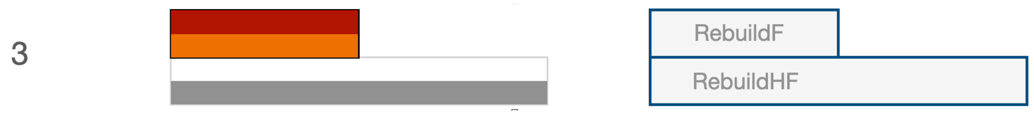
(a) At time (0) (i.e., $\mathsf{ctr} \equiv 0 \bmod 2^i$), we start rebuilding A_i^{HF} which pulls elements from level $i-1$ (colored orange next).



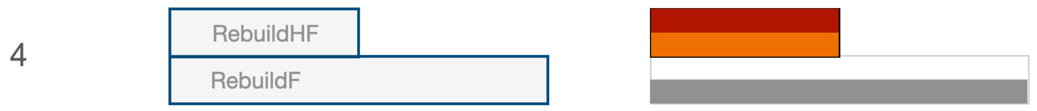
(b) At time (1) (i.e., $\mathsf{ctr} \equiv 2^{i-2} \bmod 2^i$), the table A_i^{HF} is ready, and we start rebuilding B_i^{HF}.



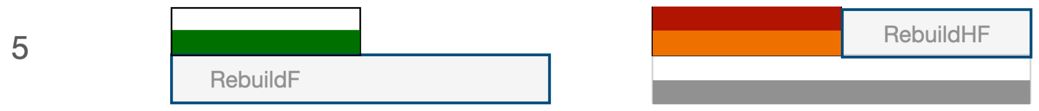
(c) At time (2), B_i^{HF} is ready, and we start rebuilding A_i^F – merging elements from A_i^{HF} and pulling new elements from level $i-1$ (colored red next).



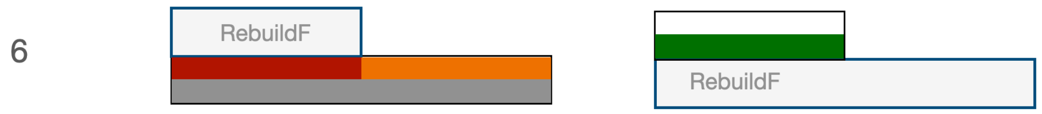
(d) At time (3) A_i^F is ready, start rebuilding B_i^F.



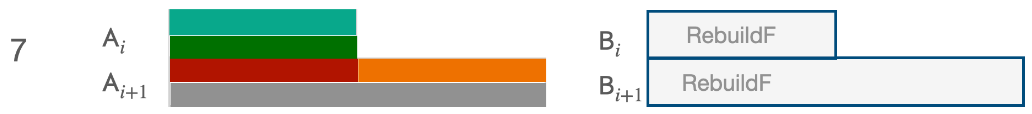
(e) At time (4) B_i^F is ready, and we again rebuild A_i^{HF}, pulling new elements (colored green next). A_{i+1}^{HF} start rebuilding, pulling the elements from A_i^F.



(f) At time (5), A_i^{HF} is ready with a new content (colored green), while B_i^F is still active, and B_i^{HF} starts to rebuild. A_{i+1}^F is still rebuilding, as it is bigger.



(g) At time (6), A_{i+1}^{HF} is ready with the old content, and B_i^{HF} is ready with the new content. We start rebuilding A_i^F to fetch new content from A_{i-1}.



(h) At time (7), A_i^F and A_{i+1}^F are both full, B_i^F and B_{i+1}^F are rebuilding.

Figure 3: The rebuilding process. A_i^{HF} and A_i^F are both shown in the same table, likewise B_i^{HF} and B_i^F.