Perfectly Secure Oblivious Parallel RAM

T-H. Hubert Chan[1], Kartik Nayak[2,3], and Elaine Shi[4]

[1] The University of Hong Kong, hubert@cs.hku.hk
[2] University of Maryland,
[3] VMware Research, nkartik@vmware.com
[4] Cornell University, elaine@cs.cornell.edu

Abstract. We show that PRAMs can be obliviously simulated with perfect security, incurring only $O(\log N \log\log N)$ blowup in parallel runtime, $O(\log^3 N)$ blowup in total work, and $O(1)$ blowup in space relative to the original PRAM. Our results advance the theoretical understanding of Oblivious (Parallel) RAM in several respects. First, prior to our work, no perfectly secure Oblivious Parallel RAM (OPRAM) construction was known; and we are the first in this respect. Second, even for the sequential special case of our algorithm (i.e., perfectly secure ORAM), we not only achieve logarithmic improvement in terms of space consumption relative to the state-of-the-art, but also significantly simplify perfectly secure ORAM constructions. Third, our perfectly secure OPRAM scheme matches the parallel runtime of earlier statistically secure schemes with negligible failure probability. Since we remove the dependence (in performance) on the security parameter, our perfectly secure OPRAM scheme in fact asymptotically outperforms known statistically secure ones if (sub-)exponentially small failure probability is desired. Our techniques for achieving small parallel runtime are novel and we employ special expander graphs to derandomize earlier statistically secure OPRAM techniques — this is the first time such techniques are used in the constructions of ORAMs/OPRAMs.

1 Introduction

Oblivious RAM (ORAM), originally proposed in the ground-breaking work by Goldreich and Ostrovsky [21, 22], is an algorithmic technique that transforms any RAM program to a secure version, such that an adversary learns nothing about the secret inputs from observing the program's access patterns to memory. The parallel extension of ORAM was first phrased by Boyle, Chung, and Pass [6]. Similar to ORAM, an Oblivious Parallel RAM (OPRAM) compiler transforms a Parallel RAM (PRAM) program into a secure form such that the resulting PRAM's access patterns leak no information about secret inputs. ORAMs and OPRAMs have been recognized as powerful building blocks in both theoretical applications such as multi-party computation [5, 25, 32], as well as in practical applications such as cloud outsourcing [13,40,43], and secure processor design [16,17,31,33,38].

Henceforth in this paper, *we consider ORAMs to be a special case of OPRAMs*, i.e., when both the original PRAM and the OPRAM have only one CPU. To characterize an OPRAM scheme's overhead, we will use the standard terminology *total work blowup* to mean the multiplicative increase in total computation comparing the OPRAM and the original PRAM; and we use the term *depth blowup* to mean the multiplicative increase in parallel runtime comparing the OPRAM and the original PRAM — assuming that *the OPRAM may employ more CPUs than the original PRAM* to help parallelize its computation [7]. Note that for the case of sequential ORAMs, total work blowup is equivalent to the standard notion of simulation overhead [21, 22], i.e., the multiplicative increase in runtime comparing the ORAM and the original RAM. Finally, we use the term *space blowup* to mean the multiplicative blowup in space when comparing the OPRAM (or ORAM) and that of the original PRAM (or RAM).

The original ORAM schemes, proposed by Goldreich and Ostrovsky [21,22], achieved poly-logarithmic overheads but required the usage of pseudo-random functions (PRFs); thus they defend only against computationally bounded adversaries. Various subsequent works [2, 9, 11, 12, 39, 41, 42], starting from Ajtai [2] and Damgård et al. [12] investigated information-theoretically secure ORAM/OPRAM schemes, i.e., schemes that do not rely on computational assumptions and defend against even unbounded adversaries. As earlier works point out [2, 12], the existence of efficient ORAM schemes without computational assumptions is not only theoretically intriguing, but also has various applications in cryptography. For example, information-theoretically secure ORAM schemes can be applied to the construction of efficient RAM-model, information-theoretically secure multi-party computation (MPC) protocols [4]. Among known information-theoretically secure ORAM/OPRAM schemes [2, 6, 9–12, 39, 41, 42], almost all of them achieve only *statistical* security [2,6,9–11,39,41,42], i.e., there is still some non-zero failure probability — either correctness or security failure — but the failure probability can be made negligibly small in N where N is the RAM/PRAM's memory size. Damgård et al. [12] came up with the first *perfectly* secure ORAM construction — they achieve zero failure probability against computationally unbounded adversaries. Although recent works have constructed statistically secure OPRAMs [6, 9, 10], there is no known (non-trivial) *perfectly* secure OPRAM scheme to date.

Motivation for perfect security. Perfectly secure ORAMs/OPRAMs are theoretically intriguing for various reasons:

1. First, to achieve $2^{-\kappa}$ failure probability (either in security or in correctness), the best known statistically secure OPRAM scheme [7, 9] incurs a $O(\kappa \log N)$ total work blowup and $O(\log \kappa \log N)$ depth blowup where N is the PRAM's memory size. Although for negligibly small in N failure probability the blowups are only poly-logarithmic in N, they can be as large as N^c for some constant $c < 1$ if one desires (sub-)exponentially small failure probability in N.

2. Second, perfectly secure ORAM schemes have been used as a building block in recent results on oblivious algorithms [3, 39] and searchable encryption schemes [14]. Typically these algorithmic constructions rely on divide-and-conquer to break down a problem into smaller sizes and then apply ORAM to a small instance — since the instance size N is small (e.g., logarithmic in the security parameter), negligible in N failure probability is not sufficient and thus these works demand *perfectly secure* ORAMs/OPRAMs and existing statistically secure schemes result in asymptotically poorer performance.
3. Third, understanding the boundary of perfect and statistical security has been an important theoretical question in cryptography. For example, a long-standing open problem in cryptography is to separate the classes of languages that admit perfect ZK and statistical ZK proofs. For ORAMs/OPRAMs too, it remains open whether there are any separations between statistical and perfect security (and we believe that this is an exciting future direction). Perfect security is also useful in other contexts such as multi-party computation (MPC). For example, Ishai et al. [28] and Genkin et al. [19] show that perfectly secure MPC is required to achieve their respective goals matching the "circuit complexity" of the underlying application. Perfectly secure ORAMs/OPRAMs can enable perfectly secure RAM-model MPC, and thus we believe that they can be an important building block in other areas of theoretical cryptography.

1.1 Our Results and Contributions

In this paper, we prove the following result which significantly advances our theoretical understanding of *perfectly* secure ORAMs and OPRAMs in multiple respects. We present the informal theorem statement below and then discuss its theoretical significance.

Theorem 1 (Informal statement of main theorem). *Any PRAM that consumes N memory blocks each of which is at least $\log N$-bits long* [5] *can be simulated by a perfectly oblivious PRAM, incurring $O(\log^3 N)$ total work blowup, $O(\log N \log\log N)$ depth blowup, and $O(1)$ space blowup.*

The above theorem improves the theoretical state of the art on perfectly secure ORAMs/OPRAMs in multiple dimensions:

1. First, our work gives rise to the first perfectly secure (non-trivial) OPRAM construction. No such construction was known before and it is not clear how to directly parallelize the perfectly secure ORAM scheme by Damgård et al. [12].
2. Second, even for the sequential special case, we improve Damgård et al. [12] asymptotically by reducing a $\log N$ factor in the ORAM's space consumption.
3. Third, our perfectly secure OPRAM's parallel runtime matches the best known statistically secure construction [7, 9] for negligibly small in N failure probabilities;

[5] All existing ORAM and OPRAM works [21–23, 30, 39] make this assumption.

4. Finally, when (sub-)exponentially small (in N) failure probabilities are required, our perfectly secure OPRAM scheme asymptotically outperforms all known statistically secure constructions both in terms of total work blowup and depth blowup. For example, suppose that we require $2^{-\kappa}$ failure probability and $N = \mathsf{poly}(\kappa)$ — then all known statistically secure OPRAM constructions [6, 9, 10] would incur at least N^c total work blowup and $\Omega(\log^2 N)$ depth blowup and thus our new perfectly secure OPRAM construction is asymptotically better for this scenario.

Theorem 1 applies to general block sizes. We additionally show that for sufficiently large block sizes, there exists a perfectly secure OPRAM construction with $O(\log^2 N)$ total work blowup and $O(\log m + \log\log N)$ depth blowup where m denotes the number of CPUs of the original PRAM (Corollary 2). Finally, we point out that this work focuses mostly on the theoretical understanding of perfect security in ORAMs/OPRAMs, and we leave it as a future research direction to investigate their practical performance (see also Section 8).

Technical highlights. Our most novel and non-trivial technical contribution is the use of *expander graphs* techniques, allowing our OPRAM to achieve as small as $O(\log N \log\log N)$ depth blowup. To the best of our knowledge, this is the first time such techniques have been used in the construction of ORAM/OPRAM schemes. Besides this novel technique, our scheme requires carefully weaving together many algorithmic tricks that have been used in earlier works [7, 9, 21, 22]

1.2 Related Work

Oblivious RAM (ORAM) was first proposed in a ground-breaking work by Goldreich and Ostrovsky [21, 22]. Goldreich and Ostrovsky first showed a computationally secure ORAM scheme with poly-logarithmic simulation overhead. Therefore, one interesting question is whether ORAMs can be constructed without relying on computational assumptions. Ajtai [2] answered this question and showed that statistically secure ORAMs with poly-logarithmic simulation overhead exist. Although Ajtai removed computational assumptions from ORAMs, his construction has a (negligibly small) statistical failure probability, i.e., with some negligibly small probability, the ORAM construction can leak information. Subsequently, Shi et al. [39] proposed the tree-based paradigm for constructing statistically secure ORAMs. Tree-based constructions were later improved further in several works [9, 11, 20, 41, 42], and this line of works improve the practical performance of ORAM by several orders of magnitude in comparison with earlier constructions. It was also later understood that the tree-based paradigm can be used to construct computationally secure ORAMs saving yet another $\log\log$ factor in cost in comparison with statistical security [9, 15].

Perfect security requires that the (oblivious) program's memory access patterns be *identically distributed* regardless of the inputs to the program; and thus with probability 1, no information can be leaked about the secret inputs to the program. Perfectly secure ORAM was first studied by Damgård et al. [12].

Their construction achieves $O(\log^3 N)$ simulation overhead and $O(\log N)$ space blowup relative to the original RAM program. Their construction is a Las Vegas algorithm and there is a negligibly small failure probability that the algorithm exceeds the stated runtime. Raskin et al. [37] and Demertzis et al. [14] achieve a *worst-case* bandwidth of $O(\sqrt{N}\frac{\log N}{\log\log N})$ and $O(N^{1/3})$, respectively. As mentioned, even for the sequential case, our paper asymptotically improves Damgård et al.'s result [12] by avoiding the $O(\log N)$ blowup in space; and moreover, our ORAM construction is conceptually simpler than that of Damgård et al.'s.

Oblivious Parallel ORAM (OPRAM) was first proposed in an elegant work by Boyle, Chung, and Pass [6], and subsequently improved in several followup works [7–10, 35]. All known results on OPRAM focus on the statistically secure or the computationally secure setting. To the best of our knowledge, until this paper, we know of no efficient OPRAM scheme that is perfectly secure. Chen, Lin and Tessaro [10] introduced a generic method to transform any ORAM into an OPRAM at the cost of a $\log N$ blowup. Their techniques achieve *statistical* security since security (or correctness) is only guaranteed with high probability (specifically, when some queue does not become overloaded in their scheme).

Defining a good performance metric for OPRAMs turned out to be more interesting and non-trivial than for ORAMs. Boyle et al. [6] were the first to define a notion of simulation overhead for OPRAM: if an OPRAM's simulation overhead is X, it means that if the original PRAM consumes m CPUs and completes in parallel runtime T, then the oblivious counterpart must complete within $X \cdot T$ time also consuming m CPUs. The recent work of Chan, Chung, and Shi [7] observes that if the OPRAM could consume more CPUs than the original PRAM, then the oblivious simulation can benefit from the additional parallelism and be additionally sped up by asymptotic factors. Under the assumption that the OPRAM can consume more CPUs than the original PRAM, Chan, Chung, and Shi [7, 9] show that statistically secure OPRAM schemes can be constructed with $O(\log^2 N)$ blowup in total work and only $\widetilde{O}(\log N)$ blowup in depth (where depth characterizes the parallel runtime of a program assuming ample number of CPUs). Our paper is the first to construct an OPRAM scheme with perfect security, and our OPRAM's depth matches existing schemes with statistical security assuming negligible in N security failure; however, if (sub-)exponentially small failure probability is required, our new OPRAM scheme can asymptotically outperform all known statistically secure OPRAMs!

2 Technical Roadmap

In this section, we present an informal roadmap of our technical approach to aid understanding.

2.1 Simplified Perfectly Secure ORAM with Asymptotically Smaller Space

First, we propose a new perfectly secure ORAM scheme that is conceptually simpler than that of Damgård et al. [12] and asymptotically gains a logarithmic factor in space. Our construction is inspired by the hierarchical ORAM paradigm originally proposed by Goldreich and Ostrovsky [21,22]. However, most existing hierarchical ORAMs achieve only computational security since they rely on a pseudorandom function (PRF) for looking up hash tables in the hierarchical data structure. Thus our focus is how to get rid of this PRF and achieve perfect security.

Background: hierarchical ORAM. The recent work by Chan et al. [8] gave a clean and modular exposition of the hierarchical paradigm. A hierarchical ORAM consists of $O(\log N)$ levels that are geometrically increasing in size. Specifically, level i is capable of storing 2^i memory blocks. One could think of this hierarchical data structure as a hierarchy of stashes where smaller levels act as stashes for larger levels. In existing schemes with computational security, each level is an *oblivious hash-table* [8]. To access a block at logical address addr, the CPU sequentially looks up every level of the hierarchy (from small to large) for the logical address addr. The physical location of a logical address addr within the oblivious hash-table is determined using a PRF whose secret key is known only to the CPU but not to the adversary. Once the block has already been found in some level, for all subsequent levels the CPU would just look for a dummy element, denoted by $\perp$. When a requested block has been found, it is marked as deleted in the corresponding level where it is found. Every 2^i memory requests, we perform a rebuild operation and merge all levels smaller than i (including the block just fetched and possibly updated if this is a write request) into level i — at this moment, the oblivious hash-table in level i is rebuilt, where every block's location in the hash table is determined using a PRF.

As Chan et al. [8] point out, the hierarchical ORAM paradigm effectively reduces the problem of constructing ORAM to constructing an oblivious hash-table supporting two operations: 1) **rebuild** takes in a set of blocks each tagged with its logical address, and constructs a hash-table data structure that facilitates lookups later; and 2) **lookup** takes a request that is either a logical address addr or dummy (denoted $\perp$), and returns the corresponding block requested. Obliviousness (defined w.r.t. the joint access patterns of the rebuild and lookup phases) is guaranteed as long as during the life-time of the oblivious hash-table, the sequence of lookup requests never ask for the same real element twice — and this invariant is guaranteed by the specific way the hierarchical ORAM framework uses the oblivious hash-table as a building block (more specifically, the fact that once a block is found, it is moved to a smaller level and a dummy block is requested from all subsequent levels).

Removing the PRF. As mentioned, an oblivious hash-table relies on a PRF to determine each block's location within a hash-table instance; and both the rebuilding phase and the lookup phase use the same PRF for placing and fetching blocks respectively. Since we wish to achieve perfect security, we would like to remove the PRF. One simple idea is to randomly permute all blocks within a level — this way, each lookup of a real block would visit a random location and we could hope to retain security as long as every real block is requested *at* *most once* for every level (in between rebuilds)[6]. Using techniques from earlier works [7, 9], it is possible to obliviously perform such a random permutation without disclosing the permutation; however, difficulty arises when one wishes to perform a look up — if blocks are randomly permuted within a level during rebuild, lookup must know where each block resides to proceed successfully. Thus if the CPU could hold a position map for free to remember where each block is in the hierarchical data structure, the problem would have been resolved: during every lookup, the CPU could first look up the physical location of the logical address requested, and then proceed accordingly.

Actually storing such a position map, however, would consume too much CPU space. To avoid storing this position map, we are inspired by the recursion technique that is commonly adopted by tree-based ORAM schemes [39] — however, as we point out soon, making the recursion idea work for the hierarchical ORAM paradigm is more sophisticated. The high-level idea is to recursively store the position map in a smaller ORAM rather than storing it on the CPU side; we could then recurse and store the position map of the position map in an even smaller ORAM, and so on — until the ORAM's size becomes $O(1)$ at which point we would have the CPU store the entire ORAM. Henceforth, we use the notation ORAM_D to denote the ORAM that stores the actual data blocks where $D = O(\log N)$; and we use ORAM_d to denote the ORAM at depth d of this recursion where $d \in [0..D-1]$. Thus, the larger d is, the larger the ORAM.

Although this recursion idea was very simple in the tree-based paradigm, it is not immediately clear how to make the same recursion idea work in the hierarchical ORAM paradigm. One trickiness arises since in a hierarchical ORAM, every 2^i requests, the ORAM would reshuffle and merge all levels smaller than i into level i — this is called a rebuild of level i. When a level-i rebuild happens, the position labels in the position-map ORAM must be updated as well to reflect the blocks' new locations. In a similar fashion, the position labels in all of $\mathsf{ORAM}_0, \mathsf{ORAM}_1, \ldots, \mathsf{ORAM}_{D-1}$ must be updated. We make the following crucial observation that will enable a *coordinated rebuild* technique which we will shortly explain:

> *(Invariant necessary for coordinated rebuild:)* If a data block resides at level i of ORAM_D, then its position labels in all recursion depths must reside in level i or smaller[7].

This invariant enables a *coordinated rebuild* technique: when the data ORAM (i.e., ORAM_D) merges all levels smaller than i into level i, all smaller recursion depths would do the same (unless the recursion depth is too small and does not have level i, in which case the entire ORAM would be rebuilt). During this coordinated rebuild, ORAM_D would first perform its rebuild, and propagate the position labels of all blocks involved in the rebuild to recursion depth $D-1$;

[6] As we point out later, randomly permuting real blocks is in fact not sufficient; we also need to allow dummy lookups by introducing an oblivious dummy linked list.

[7] A similar observation was adopted by Goodrich et al. [24] in their statistically secure ORAM construction.

then ORAM_{D-1} would perform its rebuild based on the position labels learned from ORAM_D, and propagate the new position labels involved to recursion depth $D-2$, and so on. As we shall discuss in the technical sections, rebuilding a level (in any recursion depth) can be accomplished through the help of $O(1)$ oblivious sorts and an oblivious random permutation.

Handling dummy blocks with oblivious linked lists. The above idea almost works, but not quite so. There is an additional technical subtlety regarding how to handle and use dummy blocks. Recall that during a memory access, if a block requested actually resides in a hierarchical level, we would read the memory location that contains the block (and this memory location could be retrieved through a special recursive position map technique). If a block does not reside in a level (or has been found in a smaller level), we still need to read a dummy location within the level to hide the fact that the block does not reside within the current level.

Recall that the i-th level must support up to 2^i lookups before the level is rebuilt. Thus, one idea is to introduce 2^i dummy blocks, and obliviously and randomly permute all blocks, real and dummy alike, during the rebuild. All dummy blocks may be indexed by a dummy counter, and every time one needs to look up a dummy block in a level, we will visit a new dummy block. In this way, we can retain obliviousness by making sure that every real block and every dummy block is visited at most once before the level is rebuilt again.

To make this idea fully work, there must be a mechanism for finding out where the next dummy block is every time a dummy lookup must be performed. One naïve idea would be to use the same recursion technique to store position maps for dummy blocks too — however, since each memory request might involve reading $O(\log N)$ dummy blocks, one per level, doing so would incur extra blowup in runtime and space. Instead, we use an *oblivious dummy linked list* to resolve this problem — this oblivious dummy linked list is inspired by technical ideas in the Damgård et al. construction [12]. In essence, each dummy block stores the pointer to the next dummy block, and the head pointer of the linked list is stored at a designated memory location and updated upon each read of the linked list. In the subsequent technical sections, we will describe how to rely on oblivious sorting to rebuild such an oblivious dummy linked list to support dummy lookups.

Putting it altogether. Putting all the above ideas together, the formal presentation of our perfectly secure ORAM scheme adopts a modular approach[8]. First, we define and construct an abstraction called an "oblivious one-time memory". An oblivious one-time memory allows one to obliviously create a data structure given a list of input blocks. Once created, one can look up real or dummy blocks in the data structure, and to look up a real block one must provide a correct position label indicating where the block resides (imagine for now that the position

[8] In fact, later in our paper, we omit the sequential version and directly present the parallel version of all algorithms.

label comes from an "oracle" but in the full ORAM scheme the position label comes from the recursion). An oblivious one-time memory retains obliviousness as long as every real block is looked up *at most once* and moreover, dummy blocks are looked up at most n times where n is a predetermined parameter (that the scheme is parametrized with).

Once we have this "oblivious one-time memory" abstraction, we show how to use it to construct an intermediate abstraction referred to as a "position-based ORAM". A position-based ORAM contains a hierarchy of oblivious one-time memory instances, of geometrically growing sizes. A position-based ORAM is almost a fully functional ORAM except that we assume that upon every memory request, an "oracle" will somehow provide a correct position label indicating where the requested block resides in the hierarchy.

Finally, we go from such a "position-based ORAM" to a fully functional ORAM using the special recursive position-map technique as explained. At this point, we have constructed a perfectly secure ORAM scheme with $O(\log^3 N)$ simulation overhead. Specifically, one $\log N$ factor arises from the $\log N$ depths of recursion, the remaining $\log^2 N$ factor arises from the cost of the ORAM at each recursion depth. Intuitively, our perfectly secure ORAM is a logarithmic factor more expensive than existing computationally-secure counterparts in the hierarchical framework [8, 23, 30] since the computationally-secure schemes [8, 23, 30] avoid the recursion by adopting a PRF to compute the pseudorandom position labels of blocks.

2.2 Making Our ORAM Scheme Parallel

Our next goal is to make our ORAM scheme parallel. Instead of compiling a sequential RAM program to a sequential ORAM, we are now interested in compiling a PRAM program to an OPRAM.

When the OPRAM Consumes the Same Number of CPUs as the PRAM. Suppose that the original program is a PRAM that completes in T parallel steps consuming m CPUs. We now would like to parallelize our earlier ORAM scheme and construct an OPRAM that completes in $T \cdot O(\log^3 N)$ parallel steps consuming also exactly m CPUs. To accomplish this, first, we need to parallelize within each position-based ORAM so m CPUs can perform work concurrently. This is not too difficult to accomplish given the simplicity of our position-based ORAM construction. Next, when m CPUs have all fetched position labels at one recursion depth, they need to pass these position labels to the CPUs at the next depth. The main technique needed here is oblivious routing: when the m CPUs at recursion depth d have fetched the position labels for the next recursion depth, the m CPUs at depth d must now obliviously route the position labels to the correct fetch CPU at the next recursion depth. As shown in earlier works [6, 7, 9], such oblivious routing can be accomplished with m CPUs in $O(\log m)$ parallel steps.

We stress that the *simplicity of our sequential ORAM construction makes it easy to parallelize — in comparison, we are not aware how to parallelize Damgård et al. [12]'s construction*[9].

When the OPRAM May Consume Unbounded Number of CPUs. The more interesting question is the following: *if the OPRAM is allowed to consume more CPUs than the original PRAM, can we further reduce its parallel runtime?* If so, it intuitively means that the overheads arising due to obliviousness are parallelizable in nature. This model was first considered by Chan et al. [7] and can be considered as a generalization of the case when the OPRAM must consume the same number of CPUs as the original PRAM.

So far, in our OPRAM scheme, although within each recursion depth, up to m requests can be served concurrently, the operations over all $O(\log N)$ recursion depths must be performed sequentially. There are two reasons that necessitate this sequentiality:

1. *Fetch phase:* first, to fetch from recursion depth d, one must wait for the appropriate position labels to be fetched from recursion depth $d-1$ and routed to recursion depth d;
2. *Maintain phase:* recall that coordinated rebuilding (see Section 2.1) is performed across all recursion depths in the reverse direction: recursion depth d must rebuild first and then propagate the new positions labels back to recursion depth $d-1$ before $d-1$ can rebuild (recall that recursion depth $d-1$ must store the position labels for blocks in depth d).

Note that for the fetch phase, oblivious routing between any two adjacent recursion depths would consume $O(\log m)$ depth; for the maintain phase, rebuilding a hierarchical level can consume up to $O(\log N)$ depth (due to oblivious sorting of up to $O(N)$ blocks). Thus, the current OPRAM algorithm incurs a depth blowup of $O(\log^2 N)$ for moderate sizes of m, e.g., when $\log m = \Theta(\log N)$. Our next goal is to reduce the depth blowup to $\widetilde{O}(\log N)$, and this turns out to be highly non-trivial.

Reducing the depth of the fetch phase with expander graphs. Using the recursion technique, it seems inherent that one must fetch from smaller recursion depths before embarking on larger ones. To reduce the depth of the fetch phase, we ask whether the depth incurred by oblivious routing in between adjacent recursion depths can be reduced. In the statistically and computationally secure settings, the recent work by Chan, Chung, and Shi have tried to tackle a similar problem

[9] In Damgård et al. [12], the shuffle phase incurs an $O(\log^3 N)$ depth which is the same as the overhead for accessing a block. Specifically, a $\log N$ factor arises due to oblivious sorting, a $\log N$ factor due to the existence of hierarchies, and another $\log N$ factor due to the extra $\log N$ dummies stored for every real element. Though an offline/online technique like ours may be conceivable for their scheme, the existence of the extra $\log N$ dummies makes it inherently hard to improve the depth by another $\log N$ factor.

for tree-based OPRAMs [7]. Their idea is to construct an offline/online routing algorithm. Although the offline phase incurs $O(\log N)$ depth per recursion depth, the offline work of all recursion depths can be performed concurrently rather than sequentially. On the other hand, the online phase of their routing algorithm must be performed sequentially among the recursion depths, but happily the online routing phase incurs only $O(1)$ depth per recursion depth. Unfortunately, the offline/online routing algorithm of Chan et al. [7] is a randomized algorithm that leverages some form of statistical "load balancing", and such load balancing can fail with negligibly small probability — this makes their algorithm unsuitable for the perfect security setting.

We propose a novel offline/online routing algorithm that achieves *perfect* security using special expander graphs — our techniques can be viewed as a method for derandomizing a new variant of the offline/online routing techniques described by Chan et al. [7]. Like Chan et al. [7], our offline/online routing algorithm achieves $O(\log N)$ depth for each recursion depth in the offline stage but the work in all recursion depths can be performed in parallel in the offline stage. By contrast, the online phase must traverse the recursion depths sequentially, but the online stage of routing can be accomplished in $O(1)$ depth per recursion depth. To achieve this, we rely on a core building block called a "loose compactor". Leveraging special expander graphs, we show how to build a loose compactor with small online depth — since this part of our techniques are novel, we present a more expanded overview in Section 2.3 while deferring a detailed, formal description to later technical sections (Sections 6 and 7).

Reducing the depth of the maintain phase. We also must reduce the depth of the maintain phase. Although a naïve implementation of *coordinated rebuild* is to do it sequentially from recursion depth D down to recursion depth 0, we devise a method for performing the coordinated rebuild in parallel among all recursion depths. Recall that in the naïve solution, recursion depth $d-1$ must wait for recursion depth d to relocate its blocks and be informed of the new position labels chosen before it starts reshuffling.

In our new algorithm, we introduce the notion of a rehearsal step called "mock shuffle" which determines the new positions of each of the blocks. Note that during this step, the newly chosen block contents (position labels) at the recursion depths are not available. Now, instead of sequentially performing the shuffle, in a mock shuffle, every recursion depth performs eager reshuffling without having updated the block's contents (recall that each block in recursion depth d is supposed to store position labels for the next recusion depth $d+1$). After this mock shuffle, all blocks' new positions are determined though their contents are not known. Each mock reshuffle incurs $O(\log N)$ depth, but they are independent and can be performed in parallel. At this moment, recursion depth d informs the newly chosen position labels to recursion depth $d-1$ — now recursion depth $d-1$ relies on oblivious routing to deliver each block's contents to the block. Note that recursion depth $d-1$ has already chosen each block's position at this point and thus in this content update step, each block's contents will be routed to the corresponding block and all blocks will maintain their chosen positions.

Using this idea, although each recursion depth incurs $O(\log N)$ depth for the maintain phase, all recursion depths can now perform the maintain-phase operations in parallel.

Additional techniques. Besides the above, additional tricks are needed to achieve $\widetilde{O}(\log N)$ depth. For example, within each recursion depth, all the hierarchical levels must be read in parallel during the fetch phase rather than sequentially like in existing hierarchical ORAMs [21, 22], and the result of these fetches can be aggregated using an oblivious select operation incurring $O(\log \log N)$ depth (see Section 3.3). It is possible for us to read all the hierarchical levels in parallel since each recursion depth must have received the position labels of all real blocks requested before its fetch phase starts — and thus we know for each requested block which level to look for a real element and which level to visit dummies. We defer additional algorithmic details and tricks to the later technical sections.

2.3 Offline/Online Routing with Special Expander Graphs

Informal problem statement. Without going into excessive details, consider the following abstract problem: imagine that m CPUs at a parent depth have fetched m real or dummy blocks, and each real block contains two position labels for the next depth — thus in total up to $2m$ position labels have been fetched. Meanwhile, m CPUs at the next depth are waiting to receive m position labels before they can start their fetch. Our task is to obliviously route the (up to) $2m$ position labels at the parent depth to the m CPUs at the child depth. Using oblivious routing directly would incur $\Omega(\log m)$ depth and thus is too expensive.

A blueprint: using an offline/online algorithm. As mentioned earlier, our high-level idea is to leverage an offline-online paradigm such that the online phase, which must be performed sequentially for all recursion depths, should have small parallel runtime for each recursion depth.

Here is another idea: suppose that we are somehow able to compress the $2m$ position labels down to m, removing the ones that are not needed by the next recursion depth — this is in fact non-trivial but for now, suppose that somehow it can be accomplished.

Our plan is then the following: in the offline phase, we obliviously and randomly permute the m position labels to be routed (without leaking the permutation), and we obliviously compute the routing permutation π preserving the following invariant: the CPU at position $\pi(i)$ (in the child depth) is waiting for the i-th position label in the permuted array. In other words, the i-th position label wants to be routed to the CPU in position $\pi(i)$; and in the offline phase, we want to route down this π.

If we can accomplish all of the above, then in the online phase we simply apply the routing permutation that has been recorded and it takes a single parallel step to complete the routing. Moreover, for the offline phase, as long as we can perform the operations in parallel across all recursion depths, we are allowed to incur $\log m$ depth.

Informally, obliviousness holds because of the following: recall that the m labels to be routed have been obliviously and randomly permuted. Now, although the routing permutation π is revealed in the online phase, the revealed permutation is uniform at random to an observer.

Technical challenges: compaction (and more). The above blueprint seems promising, but there are multiple technical challenges. One critical ingredient that is missing is how to perform compaction from $2m$ elements down to m, removing the labels not needed by the next recursion depth — in fact, even if we can solve this compaction problem, additional challenges remain in putting these techniques together. However, for the time being, let us focus on the compaction problem alone, leaving the remaining challenges to Sections 6 and 7. The most naïve method is again to leverage oblivious sorting but unfortunately that takes $\Omega(\log m)$ depth and thus is too expensive for our purpose.

Pippenger's factory-facility problem. Our approach is inspired by the elegant techniques described by Pippenger in constructing a self-routing super-concentrator [36]. Pippenger's elegant construction can be used to solve a "factory-facility" problem described as follows. Suppose that $2m$ factories and m facilities form a special bipartite expander graph: each factory is connected to $\mathfrak{d}$ facilities and each facility is connected to $2\mathfrak{d}$ factories, where $\mathfrak{d}$ is a constant. Among the factories, $m/64$ of them are *productive* and actually end up manufacturing products. Each productive factory produces $\mathfrak{d}/2$ products; these products must be routed to a facility to be stored, and each facility has a storage capacity of $\mathfrak{d}/2$. Now, the question is: given the set of productive factories (and assuming that the bipartite graph is known), can we find a *satisfying assignment* for routing products to facilities, such that 1) every edge in the bipartite graph routes carries at most one unit of flow; 2) all products manufactured are routed; and 3) no facility exceeds its storage capacity.

In his ingenious work [36], Pippenger described a distributed protocol for finding such an assignment: imagine that the factories and facilities are Interactive Turing Machines. Now the factories and facilities exchange messages over edges in the bipartite graph. Pippenger's protocol completes after $O(\log m)$ rounds of interaction and a total of $O(m)$ number of messages. Pippenger proved that as long as the underlying bipartite graph satisfies certain expansion properties, his protocol is guaranteed to find a satisfying assignment.

Using Pippenger's protocol for oblivious loose compaction. Now we can reduce the problem of (loose) compaction to Pippenger's factory-facility problem. Imagine that there are twice as many factories as there are facilities. Another way to think of the factory-facility problem is the following: imagine that the factories initially store real elements (i.e., the manufactured products) as well as dummies, and in total $2m \cdot (\mathfrak{d}/2)$ amount of storage is consumed since each factory can produce at most $\mathfrak{d}/2$ products. We ensure that only $m/64$ factories are productive by appropriately adding a constant factor of dummy elements (i.e., dummy factories and facilities). Now, when routed to the facilities, the storage amount is compressed down by a factor of 2 since each facility can store up to $\mathfrak{d}/2$ products and the number of facilities is half that of factories. Further, for any satisfying assignment, we guarantee that no real element is lost during the routing, and that is why the compaction algorithm satisfies correctness. Note that such compaction is *loose*, i.e., we do not completely remove dummies during compaction although we do cut down total storage by a half while preserving all real elements. In our OPRAM algorithm, it turns out that such *loose* compaction is sufficient, since CPUs who have received dummy position labels can always perform dummy fetch operations.

Pippenger's protocol can be easily simulated on a PRAM incurring $O(m)$ total work and $O(\log m)$ parallel runtime — however, a straightforward PRAM simulation of their protocol is *not* oblivious. In particular, the communication patterns between the factories and facilities (which translate to memory access patterns when simulated on a PRAM) leak information about which factories are productive. Thus it remains for us to show how to *obliviously* simulate his protocol on a PRAM. We show that this can be done incurring $O(m \log m)$ total work and $O(\log m)$ parallel runtime — note that the extra $\log m$ overhead arises from the obliviousness requirement.

Finally, we apply the loose compaction algorithm in an offline/online fashion too. In the offline phase, we execute Pippenger's protocol obliviously on a PRAM to compute the satisfying assignment — the offline phase can be parallelized over all recursion depths, thus incurring $O(\log m)$ parallel runtime overall. In the online phase, we only have to carry out the satisfying assignment that has already been recorded in the offline phase to perform the actual routing of the fetched position labels, and this can be accomplished in $O(1)$ online parallel runtime.

We defer a detailed description of the techniques to the formal technical sections.

3 Definitions

3.1 Parallel Random-Access Machines

We review the concepts of a parallel random-access machine (PRAM) and an oblivious parallel random-access machine (OPRAM). Some of the definitions in this section are borrowed verbatim from Boyle et al. [6] or Chan and Shi [9].

Although we give definitions only for the parallel case, we point out that this is without loss of generality, since a sequential RAM can be thought of as a special case PRAM with one CPU.

Parallel Random-Access Machine (PRAM). A *parallel random-access machine* consists of a set of CPUs and a shared memory denoted by mem indexed by the address space $\{0, 1, \ldots, N-1\}$, where N is a power of 2. In this paper, we refer to each memory word also as a *block*, which is at least $\Omega(\log N)$ bits long.

In a PRAM, each step of the execution can employ multiple CPUs, and henceforth we use m_t to denote the number of CPUs involved in executing the t-th step for $t \in \mathbb{N}$. In each step, each CPU executes a next instruction circuit denoted Π, updates its CPU state; and further, CPUs interact with memory through request instructions $\boldsymbol{I}^{(t)} := (I_i^{(t)} : i \in [m_t])$. Specifically, at time step t, CPU i's instruction is of the form $I_i^{(t)} := (\mathsf{read}, \mathsf{addr})$, or $I_i^{(t)} := (\mathsf{write}, \mathsf{addr}, \mathsf{data})$ where the operation is performed on the memory block with address addr and the block content data.

If $I_i^{(t)} = (\mathsf{read}, \mathsf{addr})$ then the CPU i should receive the contents of $\mathsf{mem}[\mathsf{addr}]$ at the beginning of time step t. Else if $I_i^{(t)} = (\mathsf{write}, \mathsf{addr}, \mathsf{data})$, CPU i should still receive the contents of $\mathsf{mem}[\mathsf{addr}]$ at the beginning of time step t; further, at the end of step t, the contents of $\mathsf{mem}[\mathsf{addr}]$ should be updated to data.

Write conflict resolution. By definition, multiple read operations can be executed concurrently with other operations even if they visit the same address. However, if multiple concurrent write operations visit the same address, a conflict resolution rule will be necessary for our PRAM to be well-defined. In this paper, we assume the following:

- The original PRAM supports concurrent reads and concurrent writes (CRCW) with an arbitrary, parametrizable rule for write conflict resolution. In other words, there exists some priority rule to determine which write operation takes effect if there are multiple concurrent writes in some time step t.
- Our compiled, oblivious PRAM (defined below) is a "concurrent read, exclusive write" PRAM (CREW). In other words, our OPRAM algorithm must ensure that there are no concurrent writes at any time.

We note that a CRCW-PRAM with a parametrizable conflict resolution rule is among the most powerful CRCW-PRAM model, whereas CREW is a much weaker model. Our results are stronger if we allow the underlying PRAM to be more powerful but our compiled OPRAM uses a weaker PRAM model. For a detailed explanation on how stronger PRAM models can emulate weaker ones, we refer the reader to the work by Hagerup [26].

CPU-to-CPU communication. In the remainder of the paper, we sometimes describe our algorithms using CPU-to-CPU communication. For our OPRAM algorithm to be oblivious, the inter-CPU communication pattern must be oblivious too. We stress that such inter-CPU communication can be emulated using shared memory reads and writes. Therefore, when we express our performance metrics, we assume that all inter-CPU communication is implemented with shared memory reads and writes. In this sense, our performance metrics already account for any inter-CPU communication, and there is no need to have separate metrics that characterize inter-CPU communication. In contrast, some earlier works [10] adopt separate metrics for inter-CPU communication.

Additional assumptions and notations. Henceforth, we assume that *each CPU can only store $O(1)$ memory blocks*. Further, we assume for simplicity that the runtime T of the PRAM is *fixed* a priori and *publicly known*. Therefore, we can consider a PRAM to be parametrized by the following tuple

$$\mathsf{PRAM} := (\mathit{\Pi}, N, T, m_1, m_2, \ldots, m_T),$$

where $\mathit{\Pi}$ denotes the next instruction circuit, N denotes the total memory size (in terms of number of blocks), T denotes the PRAM's total runtime, and m_t denotes the number of CPUs in the t-th step for $t \in [T]$.

Finally, in this paper, we consider PRAMs that are *stateful* and can evaluate a sequence of inputs, carrying state in between. Without loss of generality, we assume each input can be stored in a single memory block.

3.2 Oblivious Parallel Random-Access Machines

An OPRAM is a (randomized) PRAM with certain security properties, i.e., its access patterns leak no information about the inputs to the PRAM.

Randomized PRAM. A *randomized PRAM* is a PRAM where the CPUs are allowed to generate private random numbers. For simplicity, we assume that a randomized PRAM has a priori known, deterministic runtime, and that the CPU activation pattern in each time step is also fixed a priori and publicly known.

Memory access patterns. Given a PRAM program denoted PRAM and a sequence inp of inputs, we define the notation $\mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp})$ as follows:

- Let T be the total number of parallel steps that PRAM takes to evaluate inputs inp.
- Let $A_t := (\mathsf{addr}_1^t, \mathsf{addr}_2^t, \ldots, \mathsf{addr}_{m_t}^t)$ be the list of addresses such that the ith CPU accesses memory address addr_i^t in time step t.
- We define $\mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp})$ to be the random object $[A_t]_{t \in [T]}$.

Oblivious PRAM (OPRAM). We say that a PRAM is *perfectly oblivious*, iff for any two input sequences inp_0 and inp_1 of equal length, it holds that the following distributions are identically distributed (where $\equiv$ denotes identically distributed):

$$\mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp}_0) \equiv \mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp}_1)$$

We remark that for statistical and computational security, some earlier works [8, 9] presented an adaptive, composable security notion. The perfectly oblivious counterpart of their adaptive, composable notion is equivalent to our notion defined above. In particular, our notion implies security against an adaptive adversary who might choose the input sequence inp adaptively over time after having observed partial access patterns of PRAM.

We say that OPRAM is a *perfectly oblivious simulation* of PRAM iff OPRAM is perfectly oblivious, and moreover OPRAM(inp) is identically distributed as PRAM(inp) for any input inp. In the remainder of the paper, we always assume that the original PRAM has a fixed number of CPUs (denoted m) in all steps of execution. For the compiled OPRAM, we consider two models 1) when the OPRAM always consumes exactly m CPUs in every step (i.e., the same number of CPUs as the original PRAM); and 2) when the OPRAM can consume an unbounded number of CPUs in every step; in this case, the actual number of CPUs consumed in each step may vary. We leave it as an open problem how to obliviously simulate a PRAM with a varying number of CPUs (without naïvely padding the number of CPUs to the maximum which can incur large overhead).

Oblivious simulation metrics. We adopt the following metrics to characterize the overhead of (parallel) oblivious simulation of a PRAM. In the following, when we say that an OPRAM scheme consumes T parallel steps (or W total work), we mean that the OPRAM scheme consumes T parallel steps (or W total work) except with negligible in N probability. In other words, the definition of our metrics allows the OPRAM to sometimes, but with negligibly small (in N) probability, exceed the desired runtime or total work bound; however, note that the security or correctness failure probability must be 0 [10].

- Simulation overhead *(when the OPRAM consumes the same number of CPUs as the PRAM).* If a PRAM that consumes m CPUs and completes in T parallel steps can be obliviously simulated by an OPRAM that completes in $\gamma \cdot T$ steps also with m CPUs (i.e., the same number of CPUs as the original PRAM), then we say that the simulation overhead is γ. Note that this means that every PRAM step is simulated by *on average* γ OPRAM steps.
- Total work blowup *(when the OPRAM may consume unbounded number of CPUs).* A PRAM's total work is the number of steps necessary to simulate the PRAM under a single CPU, and is equal to the sum $\sum_{t \in [T]} m_t$. If a PRAM of total work W can be obliviously simulated by an OPRAM of total work $\gamma \cdot W$ we say that the total work blowup of the oblivious simulation is γ.
- Depth blowup *(when the OPRAM may consume unbounded number of CPUs).* A PRAM's depth is defined to be its parallel runtime when there are an unbounded number of CPUs. If a PRAM of depth D can be obliviously simulated by an OPRAM of depth $\gamma \cdot D$ we say that the depth blowup of the oblivious simulation is γ.

Note that the simulation overhead is a good standalone metric if we assume that the OPRAM must consume the same number of CPUs as the PRAM. If the OPRAM is allowed to consume more CPUs than the PRAM, we typically use the metrics total work blowup and depth blowup in conjunction with each other: total work blowup alone does not characterize how much the OPRAM

[10] Similarly, the perfectly secure ORAM by Damgård et al. [12] also allowed a negligible small probability for the algorithm to exceed the desired complexity bound but the security or correctness failure probability must be 0.

preserves parallelism; and depth blowup alone does not capture the extent to which the OPRAM preserves total work.

Finally, the following simple fact is useful for understanding the complexity of (oblivious) parallel algorithms.

Fact 2 *Let $C > 1$. If an (oblivious) parallel algorithm can complete in T steps consuming m CPUs, then it can complete in CT steps consuming $\lceil \frac{m}{C} \rceil$ CPUs.*

3.3 Building Blocks

We now introduce several useful oblivious building blocks. With the exception of oblivious random permutation, we assume that all remaining building blocks are deterministic: for a deterministic algorithm, obliviousness means that the algorithm's memory access pattern is independent of its input.

Oblivious sort. Ajtai, Komlós, and Szemerédi [1] show how to construct a circuit with $n \log n$ comparators that can correctly sort any input sequence containing n comparable elements. This immediately gives rise to a parallel oblivious sorting algorithm with $O(n \log n)$ total work and $O(\log n)$ depth.

Oblivious routing. Oblivious routing solves the following problem. Suppose n source CPUs each holds a data block with a distinct key (or a dummy block). Further, n destination CPUs each holds a key and requests a data block identified by its key — multiple destination CPUs can possibly request the same key. An oblivious routing algorithm routes the requested data block to the destination CPU in an oblivious manner. We may assume that the destination CPUs are represented by an ordered array X. Initially the payload of each entry of X is left empty. After the routing, each entry of X receives a data block (the received data block is dummy if no source CPUs hold the same key as requested). The ordering of elements in X is preserved between the input and output.

Boyle et al. [6] showed that through a combination of oblivious sorts and oblivious aggregation, oblivious routing can be achieved in $O(\log n)$ parallel runtime with $O(n)$ CPUs.

Obliviously computing the routing permutation. Suppose that we are given a source array src of length n where each entry holds a distinct key, and a destination array dst also of length n where each entry holds a distinct key. Further, it is guaranteed that the set of keys in src is the same as the set of keys in dst. We would like to write down a permutation π (henceforth referred to as the routing permutation) such that applying π to src would result in the same order of keys as dst. The recent work by Chan and Shi [9] showed how to implement the above task obliviously using $O(1)$ number of oblivious sorts. Thus, with $O(n)$ CPUs the routing permutation can be computed in $O(\log n)$ parallel runtime.

Oblivious select. Consider the following problem: given a set of n elements among which at most one element is distinguishing, output the distinguishing element (and if no element is distinguishing, output $\perp$). It is not difficult to see that by building an aggregation tree over the n elements, one can accomplish oblivious select with n CPUs in $\log n$ parallel steps.

Oblivious prefix sum. Given an array X of length n, every $i \in [n]$ wants to compute the sum of the prefix $X[1..i]$. There exists a parallel oblivious algorithm to achieve this in $O(\log n)$ steps consuming n CPUs [27].

Oblivious random permutation. Let ORP be an algorithm that upon receiving an input array X, outputs a permutation of X. Let $\mathcal{F}_{\text{perm}}$ denote an ideal functionality that upon receiving the input array X, outputs a perfectly random permutation of X.

We say that ORP is a *perfectly oblivious* random permutation, iff there exists a simulator Sim such that the joint distribution $(\mathcal{F}_{\text{perm}}(X), \mathsf{Sim}(|X|))$ is identically distributed as the joint distribution of the output and the addresses incurred by running ORP on X. Note that the simulator Sim is given only the input length $|X|$ but not the contents of X.

Chan, Chung, and Shi [7] recently describe a perfectly oblivious random permutation algorithm, which, except with negligible in λ probability, completes in $O(\log n)$ parallel steps consuming n CPUs assuming each block is large enough to store $\log \lambda$ bits. We summarize their construction in the following theorem:

Theorem 3 (Perfectly oblivious random permutation [7]). *Assume that each memory block is large enough to store at least $\log \lambda$ bits and that $n \leq \lambda \leq 2^{O(n^2)}$. Then, there exists a perfectly oblivious random permutation algorithm that consumes n CPUs. Except with negligible in λ probability, the algorithm completes in $O(\log n)$ parallel steps and $O(n \log n)$ work.*

We note that the failure is in terms of the algorithm's runtime — there is a negligibly small probability that the algorithm will run for longer, but the algorithm guarantees perfect security regardless.

4 Parallel One-Time Oblivious Memory

We define and construct an abstract datatype to process non-recurrent memory lookup requests. Although the abstraction is similar to the oblivious hashing scheme in Chan et al. [8], our one-time memory scheme needs to be perfectly secure and does not use a hashing scheme. Furthermore, we assume that every real lookup request is *tagged with a correct position label* that indicates where the requested block is — in this section, we simply assume that the correct position labels are simply provided during lookup; but later in our full OPRAM scheme, we will use a recursive ORAM/OPRAM technique reminiscent of those used in binary-tree-based ORAM/OPRAM schemes [9, 11, 39, 41, 42] such that we can obtain the position label of a block first before fetching the block.

4.1 Definition: One-Time Oblivious Memory

Intuition. We describe the intuition using the *sequential* special case but our formal presentation later will directly describe the parallel version. An oblivious one-time memory supports three operations: 1) Build, 2) Lookup, and 3) Getall. Build is called once upfront to create the data structure: it takes in a set of real blocks (each tagged with its logical address) and creates a data structure that facilitates lookup. After this data structure is created, a sequence of lookup operations can be performed: each lookup can request a real block identified by its logical address or a dummy block denoted $\perp$ — if the requested block is a real block, we assume that the correct position label is supplied to indicate where in the data structure the requested block is. Finally, when the data structure is no longer needed, one may call a Getall operation to obtain a list of blocks (tagged with their logical addresses) that have not been looked up yet — in our OPRAM scheme later, this is the set of blocks that need to be preserved during rebuilding.

We require that our oblivious one-time memory data structure retain obliviousness as long as 1) the sequence of real blocks looked up all exist in the data structure (i.e., it appeared as part of the input to Build), and moreover, each logical address is looked up at most once; and 2) at most $\widetilde{n}$ number of dummy lookups may be made where $\widetilde{n}$ is a predetermined parameter (that the scheme is parametrized with).

Formal Definition Our formal presentation will directly describe the parallel case. In the parallel version, lookup requests come in batches of size $m > 1$.

A (parallel) one-time memory scheme denoted $\mathsf{OTM}^{[n,m,t]}$ is parametrized by three parameters: n denotes the upper bound on the number of real elements; m is the batch size for lookups; t is the upper bound on the number of batch lookups supported. We use three parameters because we use different versions of OTM. For the basic version in Section 5, we have $t = \frac{n}{m}$ number of batch lookups, whereas in Section 7, the number of batch lookups is larger (which means that some of the lookup addresses must be dummy).

The (parallel) one-time memory scheme $\mathsf{OTM}^{[n,m,t]}$ is comprised of the following possibly randomized, stateful algorithms to be executed on a *Concurrent-Read, Exclusive-Write* PRAM — note that since the algorithms are stateful, every invocation will update an implicit data structure in memory. Henceforth we use the terminology key and value in the formal description but in our OPRAM scheme later, a real key will be a logical memory address and its value is the block's content.

– $U \leftarrow \mathsf{Build}(\{(k_i, v_i) : i \in [n]\})$: given a set of n key-value pairs (k_i, v_i), where each pair is either real or of the form $(\perp, \perp)$, the Build algorithm creates an implicit data structure to facilitate subsequent lookup requests, and moreover outputs a list U of exactly n key-position pairs where each pair is of the form (k, pos). Further, every real key input to Build will appear exactly once in the list U; and the list U is padded with $\perp$ to a length n. Note that U

does not include the values v_i's. Later in our scheme, this key-position list U will be propagated back to the parent recursion depth during a coordinated rebuild[11].

– $(v_i : i \in [m]) \leftarrow \mathsf{Lookup}(\{(k_i, \mathsf{pos}_i) : i \in [m]\})$: there are m concurrent Lookup operations in a single batch, where we allow each key k_i requested to be either real or $\perp$. Moreover, in each batch, at most n/t of the keys are real.
– $R \leftarrow \mathsf{Getall}$: the Getall algorithm returns an array R of length n where each entry is either $\perp$ or real and of the form (k, v). The array R should contain all real entries that have been inserted during Build but have not been looked up yet, padded with $\perp$ to a length of n.

Valid request sequence. Our oblivious one-time memory ensures obliviousness only if lookups are non-recurrent (i.e., never look for the same real key twice); and moreover the number of lookups requests must be upper bounded by a predetermined parameter. More formally, a sequence of operations is valid, iff the following holds:

– The sequence begins with a single call to Build upfront; followed by a sequence of at most t batch Lookup calls, each of which supplies a batch of m keys and the corresponding position labels; and finally the sequence ends with a single call to Getall.
– The Build call is supplied with an input array $S := \{(k_i, v_i)\}_{i \in [n]}$, such that any two real entries in S must have distinct keys.
– For every $\mathsf{Lookup}(\{(k_i, \mathsf{pos}_i) : i \in [m]\})$ query in the sequence, if each k_i is a real key, then k_i must be contained in S that was input to Build earlier. In other words, Lookup requests are not supposed to ask for real keys that do not exist in the data structure[12]; moreover, each (k_i, pos_i) pair supplied to Lookup must exist in the U array returned by the earlier invocation of Build, i.e., pos_i must be a correct position label for k_i; and
– Finally, in all Lookup requests in the sequence, no two keys requested (either in the same or different batches) are the same.

Correctness. Correctness requires that

1. for any valid request sequence, with probability 1, for every $\mathsf{Lookup}(\{(k_i, \mathsf{pos}_i) : i \in [m]\})$ request, the i-th answer returned must be $\perp$ if $k_i = \perp$; else if $k_i \neq \perp$, Lookup must return the correct value v_i associated with k_i that was input to the earlier invocation of Build.

[11] Note that we do not explicitly denote the implicit data structure in the output of Build, since the implicit data structure is needed only internally by the current oblivious one-time memory instance. In comparison, U is explicitly output since U will later on be (externally) needed by the parent recursion depth in our OPRAM construction.

[12] We emphasize this is a major difference between this one-time memory scheme and the oblivious hashing abstraction of Chan et al. [8]); Chan et al.'s abstraction [8] allows lookup queries to ask for keys that do not exist in the data structure.

2. for any valid request sequence, with probability 1, Getall must return an array R containing every (k, v) pair that was supplied to Build but has not been looked up; moreover the remaining entries in R must all be $\perp$.

Perfect obliviousness. We say that two valid request sequences are *length-equivalent*, if the input sets to Build have equal size, and the number of Lookup requests (where each request asks for a batch of m keys) in the two sequences are the same.

We say that a (parallel) one-time memory scheme is perfectly oblivious, iff for any two length-equivalent request sequences that are valid, the distribution of access patterns resulting from the algorithms are *identically distributed.*

4.2 Construction

Intuition. We first explain the intuition for the sequential case, i.e., $m = 1$. The intuition is simply to permute all elements received as input during Build. However, since subsequent lookup requests may be dummy (also denoted $\perp$), we also need to pad the array with sufficiently many dummies to support these lookup requests. The important invariant is that *each real element as well as each dummy will be accessed at most once* during lookup requests. For reals, this is guaranteed since the definition of a valid request sequence requires that each real key be requested no more than once, and that each real key requested must exist in the data structure. For dummies, every time a $\perp$-request is received, we always look for an unvisited dummy. To implement this idea, one tricky detail is that unlike real lookup requests, dummy requests do not carry the position label of the next dummy to be read — thus our data structure itself must maintain an *oblivious linked list* of dummies such that we can easily find out where the next dummy is. Since all real and dummies are randomly permuted during Build, and due to the aforementioned invariant, every lookup visits a completely random location of the data structure thus maintaining perfect obliviousness.

It is not too difficult to make the above algorithm parallel (i.e., for the case $m > 1$). To achieve this, one necessary modification is that instead of maintaining a single dummy linked list, we now must maintain m dummy linked lists. These m dummy linked lists are created during Build and consumed during Lookup.

Detailed Construction. At the end of Build, our algorithm creates an in-memory data structure consisting of the following:

1. An array A of length $n + \widetilde{n}$, where $\widetilde{n} := tm$ denotes the number of dummies and n denotes the number of real elements. Each entry of the array A (real or dummy alike) has four fields (key, val, next, pos) where
   - key is a key that is either real or dummy; and val is a value that is either real or dummy.
   - the field next $\in [0..n + \widetilde{n})$ matters only for dummy entries, and at the end of the Build algorithm, the next field stores the position of the next entry in the dummy linked list (recall that all dummy entries form m linked lists); and

  - the field $\mathsf{pos} \in [0..n + \widetilde{n})$ denotes where in the array an entry finally wants to be — at the end of the Build algorithm it must be that $A[i].\mathsf{pos} = i$. However, during the algorithm, entries of A will be permuted transiently; but as soon as each element i has decided where it wants to be (i.e., $A[i].\mathsf{pos}$), it will always carry its desired position around during the remainder of the algorithm.
2. An array that stores the head pointers of all m dummy linked lists. Specifically, we denote the m head pointers as $\{\mathsf{dpos}_i : i \in [m]\}$ where each $\mathsf{dpos}_i \in [0..n + \widetilde{n})$ is the head pointer of one dummy linked list.

These in-memory data structures, including A and the dummy pointers will then be updated during Lookup.

Build. Our oblivious $\mathsf{Build}(\{(k_i, v_i)\}_{i\in[n]})$ algorithm proceeds as follows.

1. *Initialize.* Construct an array A of length $n+\widetilde{n}$ whose entries are of the form described above. Specifically, the keys and values for the first n entries of A are copied from the input. Recall that the input may contain dummies too, and we use $\perp$ to denote a dummy key from the input.
   The last $\widetilde{n}$ entries of A contain *special* dummy keys that are numbered. Specifically, for each $i \in [1..\widetilde{n}]$, we denote $A_n[i] := A[n-1+i]$, and the entry stored at $A_n[i]$ has key $\perp_i$ and value $\perp$.
2. *Every element decides at random its desired final position.* Specifically, perform a perfectly oblivious random permutation on the entries of A — this random permutation decides where each element finally wants to be.
   Now, for each $i \in [0..n+\widetilde{n})$, let $A[i].\mathsf{pos} := i$. At this moment, $A[i].\mathsf{pos}$ denotes where the element $A[i]$ finally wants to be. Henceforth in the algorithm, the entries of A will be moved around but each element always carries around its desired final position.
3. *Construct the key-position map U.* Perform oblivious sorting on A using the field key. We assume that real keys have the highest priority followed by $\perp < \perp_1 < \cdots < \perp_{\widetilde{n}}$ (where smaller keys come earlier).
   At this moment, we can construct the key-position map U from the first n entries of A — recall that each entry of U is of the form (k, pos).
4. *Construct m dummy linked lists.* Observe that the last $\widetilde{n}$ entries of A contain special dummy keys, on which we perform the following to build m disjoint singly-linked lists (each of which has length t). For each $i \in [1..\widetilde{n}]$, if $i \bmod t \neq 0$ we update the entry $A_n[i].\mathsf{next} := A_n[i + 1].\mathsf{pos}$, i.e., each dummy entry (except the last entry of each linked list) records its next pointer.
   We next record the positions of the heads of the m lists. For each $i \in [m]$, we set $\mathsf{dpos}_i := A_n[t(i - 1)].\mathsf{pos}$.
5. *Move entries to their desired positions.* Perform an oblivious sort on A, using the fourth field pos. (This restores the ordering according to the previous random permutation.)

At this moment, the data structure $(A, \{\mathsf{dpos}_i : i \in [m]\})$ is stored in memory. The key-position map U is explicitly output and later in our OPRAM scheme it will be passed to the parent recursion depth during coordinated rebuild.

Fact 4 *Consuming $O(\widetilde{n} + n)$ CPUs and setting $(\widetilde{n} + n)^2 \leq \lambda \leq 2^{\widetilde{n}+n}$, the* Build *algorithm completes in $O(\log(\widetilde{n}+n))$ parallel steps, except with probability negligible in λ.*

Proof. Observe that the algorithm's cost is dominated by $O(1)$ number of oblivious sorts which can be realized with the AKS sorting network [1].

Moreover, the algorithm incurs one application of oblivious random permutation, whose performance is stated in Theorem 3.

Lookup. We implement a batch of m concurrent lookup operations $\{\mathsf{Lookup}(\{(k_i, \mathsf{pos}_i) : i \in [m]\})$ as follows. For each $i \in [m]$, we perform the following *in parallel.*

1. *Decide position to fetch from.* If $k_i \neq \perp$ is real, set $\mathsf{pos} := \mathsf{pos}_i$, i.e., we want to use the position label supplied from the input. Else if $k_i = \perp$, set $\mathsf{pos} := \mathsf{dpos}_i$, i.e., the position to fetch from is the next dummy in the i-th dummy linked lists. (To ensure obliviousness, the algorithm can always pretend to execute both branches of the if-statement.)
   At this moment, pos is the position to fetch from (for the i-th request out of m concurrent requests).
2. *Read and remove.* Read the value from $A[\mathsf{pos}]$ and mark $A[\mathsf{pos}] := \perp$.
3. *Update dummy head pointer if necessary.* If $\mathsf{pos} = \mathsf{dpos}_i$, update the dummy head pointer $\mathsf{dpos}_i := \mathsf{next}$. (To ensure obliviousness, the algorithm can pretend to modify dpos_i in any case.)
4. *Return.* Return the value read in the above Step 2.

The following fact is straightforward from the description of the algorithm.

Fact 5 *The* Lookup *algorithm completes in $O(1)$ parallel steps with $O(m)$ CPUs.*

Getall. Getall is implemented by the following simple procedure: obliviously sort A by the key such that all real entries are packed in front. Return the first n entries of the resulting array (and removing the metadata entries next and pos in the result).

Fact 6 *The* Getall *algorithm completes in $\log(\widetilde{n} + n)$ parallel steps consuming $O(\widetilde{n} + n)$ CPUs.*

Proof. Straightforward by observing that the algorithm's cost is dominated by $O(1)$ number of oblivious sorts which can be realized with the AKS sorting network [1].

Lemma 1 (Perfect obliviousness of the one-time memory scheme). *The above (parallel) one-time memory scheme satisfies perfect obliviousness.*

Proof. It suffices to prove that for any valid request sequence, the memory access patterns are identically distributed as those output by the following simulator that knows only n, m and the number of Lookup requests in the sequence.

First, almost all parts of Build are deterministic and data oblivious and thus the algorithm's access patterns can be simulated in the most straightforward fashion. The only randomized part of access patterns for Build is due to the oblivious random permutation. To simulate this part, the simulator calls the oblivious random permutation's simulator algorithm.

Second, to simulate the access patterns of Lookup, the simulator would read the memory location storing dpos_i for every $i \in [m]$. Then, it reads a random unread index of the array A and writes to it once too. Finally, it writes to dpos_i for every $i \in [m]$.

Third, simulating the access patterns of Getall is done in the most natural manner since Getall is deterministic.

It is not difficult to see that the real-world access patterns are identically distributed as the simulated ones due to the definition of oblivious random permutation (see Section 3.3) Particularly, observe that the above way of simulating the access patterns of Build is the same in nature as if we randomly permuted the data structure A upfront by a random permutation, (that is chosen independently from the simulated access patterns), then every real element and $\perp_i$ will be in a random location. Note also that as long as no two real keys requested collide and every real key requested exists in the data structure A, then the real-world algorithm accesses each real or $\perp_i$ element at most once, and thus every real-world access visits a random position of the array A (besides reading and writing $\{\mathsf{dpos}_i : i \in [m]\}$).

Summarizing the above, we conclude with the following theorem.

Theorem 7 (One-time oblivious memory). *Let $\lambda \in \mathbb{N}$ be a parameter related to the probability that the algorithm's runtime exceeds a desired bound. Assume that each memory block can store at least $\log n + \log \lambda$ bits. There exists a perfectly oblivious one-time scheme such that* Build *takes $O(\log n)$ parallel steps (except with negligible in λ probability) consuming n CPUs,* Lookup *for a batch of m requests takes $O(1)$ parallel steps consuming m CPUs, and* Getall *takes $O(\log n)$ parallel steps consuming n CPUs.*

5 Basic OPRAM with $O(\log^3 N)$ Simulation Overhead

Recall that N denotes the number of logical memory blocks consumed by the original PRAM, and each memory block can store at least $\Omega(\log N)$ bits. For clarity, in this section, we will first describe an OPRAM construction such that each batch of m memory requests takes $O(\log^3 N)$ parallel steps to satisfy with m CPUs. Later in Section 7, we will describe how to further parallelize the OPRAM when the OPRAM can consume more CPUs than the original PRAM.

Roadmap. We briefly explain the technical roadmap of this section:

- In Section 5.1, we will first describe a *position-based OPRAM* that supports two operations: Lookup and Shuffle. A position-based OPRAM is *an almost*

fully functional OPRAM scheme except that every real lookup request must supply a correct position label. In our OPRAM construction, these position labels will have been fetched from small recursion depths and therefore will be ready when looking up the position-based OPRAM.
Our position-based OPRAM relies on the hierarcial structure proposed by Goldreich and Ostrovsky [21,22], as well as techniques by Chan et al. [8] that showed how to parallelize such a hierarchical framework.
– In Section 5.2, we explain how to leverage "coordinated rebuild" and recursion techniques to build a recursive OPRAM scheme that composes logarithmically many instances of our position-based OPRAM, of geometrically decreasing sizes.

5.1 Position-Based OPRAM

Our basic OPRAM scheme (Section 5.2) will consist of logarithmically many position-based OPRAMs of geometrically increasing sizes, henceforth denoted OPRAM_0, OPRAM_1, OPRAM_2, ..., OPRAM_D where $D := \log_2 N - \log_2 m$. Specifically, OPRAM_d stores $\Theta(2^d \cdot m)$ blocks where $d \in \{0, 1, \ldots, D\}$. The last one OPRAM_D stores the actual data blocks whereas every other OPRAM_d where $d < D$ recursively stores the position labels for the next depth $d + 1$.

Data Structure. As we shall see, the case OPRAM_0 is trivial and is treated specially at the end of this section (Section 5.1). Below we focus on describing OPRAM_d for some $1 \le d \le D = \log N - \log m$. For $d \neq 0$, each OPRAM_d consists of $d + 1$ *levels* geometrically growing in size, where each level is a *one-time oblivious memory scheme* as defined and described in Section 4. We specify this data structure more formally below.

Hierarchical levels. The position-based OPRAM_d consists of $d + 1$ levels henceforth denoted as $(\mathsf{OTM}_j : j = 0, \ldots, d)$ where level j is a one-time oblivious memory scheme,

$$\mathsf{OTM}_j := \mathsf{OTM}^{[2^j \cdot m, m, 2^j]}$$

with at most $n = 2^j \cdot m$ real blocks and m concurrent lookups in each batch (which can all be real). This means that for every OPRAM_d, the smallest level is capable of storing up to m real blocks. Every subsequent level can store twice as many real blocks as the previous level. For the largest OPRAM_D, its largest level is capable of storing N real blocks given that $D = \log N - \log m$ — this means that the total space consumed is $O(N)$.

Every level j is marked as either *empty* (when the corresponding OTM_j has not been rebuilt) or *full* (when OTM_j is ready and in operation). Initially, all levels are marked as empty, i.e., the OPRAM initially is empty.

Position label. Henceforth we assume that a position label of a block specifies 1) which level the block resides in; and 2) the position within the level the block resides at.

Additional assumption. We assume that each block is of the form (logical address, payload), i.e., each block carries its own logical address.

Operations. Each position-based OPRAM supports two operations, Lookup and $\mathsf{Shuffle}$. For every OPRAM_d consisting of $d+1$ levels, we rely on the following algorithms for Lookup and $\mathsf{Shuffle}$.

Lookup. Every batch lookup operation, denoted $\mathsf{Lookup}(\{(\mathsf{addr}_i, \mathsf{pos}_i) : i \in [m]\})$ receives as input the logical addresses of m blocks as well as a correct position label for each requested block. To complete the batch lookup request, we perform the following.

1. For each level $j = 0, \ldots, d$ *in parallel*, perform the following:
   - For each $i \in [m]$ in parallel, first check the supplied position label pos_i to see if the requested block resides in the current level j: if so, let $\mathsf{addr}'_i := \mathsf{addr}_i$ and let $\mathsf{pos}'_i := \mathsf{pos}_i$ (and specifically the part of the position label denoting the offset within level j); else, set $\mathsf{addr}'_i := \perp$ and $\mathsf{pos}'_i := \perp$ to indicate that this should be a dummy request.
   - $(v_{ij} : i \in [m]) \leftarrow \mathsf{OTM}_j.\mathsf{Lookup}(\{\mathsf{addr}'_i, \mathsf{pos}'_i : i \in [m]\})$.
2. At this point, each of the m CPUs has d answers from the d levels respectively, and only one of them is the valid answer. Now each of the m CPUs chooses the correct answer as follows.
   For each $i \in [m]$ in parallel: set val_i to be the only non-dummy element in $(v_{ij} : j = 0, \ldots, d)$, if it exists; otherwise set $\mathsf{val}_i := \perp$. This step can be accomplished using an oblivious select operation (see Section 3.3) in $\log d$ parallel steps consuming d CPUs.
3. Return $(\mathsf{val}_i : i \in [m])$.

We remark that in Goldreich and Ostrovsky's original hierarchical ORAM [21, 22], the hierarchical levels must be visited sequentially — for obliviousness, if the block is found in some smaller level, all subsequent levels must perform a dummy lookup. Here we can visit all levels in parallel since the position label already tells us which level it is in. Now the following fact is straightforward to observe:

Fact 8 *For* OPRAM_d*,* Lookup *consumes* $O(\log d)$ *parallel steps consuming* $m \cdot d$ *CPUs where* m *is the batch size.*

Shuffle. Similar to earlier hierarchical ORAMs [21,22] and OPRAMs [8], a shuffle operation merges consecutively full levels into the next empty level (or the largest level). However, in our $\mathsf{Shuffle}$ abstraction, there is an input U that contains some logical addresses together with new values to be updated. Moreover, the shuffle operation is associated with an update function that determines how the new values in U should be incorporated into the OTM during the rebuild.

In our full OPRAM scheme later, the update array U will be passed from the immediate next depth OPRAM_{d+1}, and contains the new position labels that

OPRAM_{d+1} has chosen for recently accessed logical addresses. These position labels must then be recorded by OPRAM_d appropriately.

More formally, each position-based OPRAM_d supports a shuffle operation, denoted $\mathsf{Shuffle}(U, \ell; \mathsf{update})$, where the parameters are explained as follows:

1. An update array U in which each (non-dummy) entry contains a logical address that needs to be updated, and a new value for this block. (Strictly speaking, we allow a block to be partially updated.)
   We will define additional constraints on U subsequently.
2. The level ℓ to be rebuilt during this shuffle.
3. An update function that specifies how the information in U is used to compute the new value of a block in the OTM.
   The reason we make this rule explicit in the notation is that a block whose address that appears in U may only be partially modified; hence, we later need to specify this update function carefully. However, to avoid cumbersome notation, we may omit the parameter update, and just write $\mathsf{Shuffle}(U, \ell)$, when the context is clear.

For each OPRAM_d, when $\mathsf{Shuffle}(U, \ell; \mathsf{update})$ is called, it must be guaranteed that $\ell \leq d$; and moreover, either level ℓ must either be empty or $\ell = d$ (i.e., this is the largest level in OPRAM_d). Moreover, there is an extra OTM'_0; jumping ahead, we shall see that OTM'_0 contains the blocks that are freshly fetched.

The $\mathsf{Shuffle}$ algorithm then combines levels $0, 1, \ldots, \ell$ (of OPRAM_d), together with the extra OTM'_0, into level ℓ, updating some blocks' contents as instructed by the update array U and the update function update. At the end of the shuffle operation, all levels $0, 1, \ldots, \ell - 1$ are now marked as empty and level ℓ is now marked as full.

We now explain the assumptions we make on the update array U and how we want the update procedure to happen:

– We require that each logical address appears at most once in U.
– Let A be all logical addresses remaining in levels 0 to ℓ in OPRAM_d: it must hold that the set of logical addresses in U is a subset of those in A. In other words, a subset of the logical addresses in A will be updated before rebuilding level ℓ.
– If some logical address addr exists only in A but not in U, after rebuilding level ℓ, the block's value from the current OPRAM_d should be preserved. If some logical address addr exists in both A and in U, we use the update function to modify its value: update takes a pair of blocks $(\mathsf{addr}, \mathsf{data})$ and $(\mathsf{addr}, \mathsf{data}')$ with the same address but possibly different contents (the first of which coming from the current OPRAM_d and the second coming from U), and computes the new block content data^* appropriately.
  We remark that the new value data^* might depend on both data and data'. Later, we will describe how the update rule is implemented.

Upon receiving $\mathsf{Shuffle}(U, \ell; \mathsf{update})$, proceed with the following steps:

1. Let $A := \cup_{i=0}^{\ell}\mathsf{OTM}_i.\mathsf{Getall} \cup \mathsf{OTM}'_0.\mathsf{Getall}$, where the operator $\cup$ denotes concatenation. Moreover, for an entry in A that comes from OTM_i, then it also carries a label i.
   At this moment, the old $\mathsf{OTM}_0, \ldots, \mathsf{OTM}_\ell$ instances may be destroyed.
2. We obliviously sort $A \cup U$ in increasing order of logical addresses, and moreover, placing all dummy entries at the end. If two blocks have the same logical address, place the entry coming from A in front of the one coming from U.
   At this moment, in one linear scan, we can operate on every adjacent pair of entries using the aforementioned update operation, such that if they share the same logical address, the first entry is preserved and updated to a new value, and the second entry is set to dummy.
   We now obliviously sort the resulting array moving all dummies to the end. We truncate the resulting array preserving only the first $2^\ell \cdot m$ elements and let A' denote the outcome (note that only dummies and no real blocks will truncated in the above step).
3. Next, we call $U' \leftarrow \mathsf{Build}(A')$ that builds a new OTM' and U' contains the positions of blocks in OTM'.
4. OTM' is now the new level ℓ and henceforth it will be denoted OTM_ℓ. Mark level ℓ as full and levels $0, 1, \ldots, \ell - 1$ as empty. Finally, output U' (in our full OPRAM construction later, U' will be passed to the next (i.e., immediately smaller) position-based OPRAM as the update array for performing its shuffle).

If we realize the oblivious sort with the AKS network [1] that sorts n items in $O(\log n)$ parallel steps consuming n CPUs, we easily obtain the following fact — note that there is a negligible in N probability that the algorithm runs longer than the stated asymptotic time due to the oblivious random permutation building block (see Section 3.3).

Fact 9 *Suppose that the* update *function can be evaluated by a single CPU in $O(1)$ steps. For OPRAM_d, let $\ell \leq d$, then except with negligible in N probability, $\mathsf{Shuffle}(U, \ell)$ takes $O(\log(m \cdot 2^\ell))$ parallel steps consuming $m \cdot 2^\ell$ CPUs.*

Observe that in the above fact, the randomness comes from the oblivious random permutation subroutine used in building the one-time oblivious memory data structure.

Trivial case: OPRAM_0*.* In this case, OPRAM_0 simply stores its entries in an array $A[0..m)$ of size m and we assume that the entries are indexed by a $(\log_2 m)$-bit string. Moreover, each address is also a $(\log_2 m)$-bit string, whose block is stored at the corresponding entry in A.

– *Lookup.* Upon receiving a batch of m depth-m truncated addresses where all the real addresses are distinct, use oblivious routing to route $A[0..m)$ to the requested addresses. This can be accomplished in $O(m \log m)$ total work and $O(\log m)$ depth. Note that OPRAM_0's lookup does not receive any position labels.
– *Shuffle.* Since there is only one array A (at level 0), $\mathsf{Shuffle}(U, 0)$ can be implemented by oblivious sorting.

5.2 OPRAM Scheme from Position-Based OPRAM

Recursive OPRAMs. The OPRAM scheme consists of $D + 1$ position-based OPRAMs henceforth denoted as $\mathsf{OPRAM}_0, \mathsf{OPRAM}_1, \mathsf{OPRAM}_2, \ldots, \mathsf{OPRAM}_D$. OPRAM_D stores the actual data blocks, whereas every other OPRAM_d where $d \neq D$ recursively stores the position labels for the next data structure OPRAM_{d+1}. Our construction is in essence recursive although in presentation we shall spell out the recursion for clarity. Henceforth we often say that OPRAM_d is at *recursion depth d* or simply *depth d*.

Although we are inspired by the recursion technique for tree-based ORAMs [39], using this recursion technique in the context of hierarchical ORAMs/OPRAMs raises new challenges. In particular, we cannot use the recursion in a blackbox fashion like in tree-based constructions since all of our (position-based, hierarchical) OPRAMs must reshuffle in sync with each other in a non-blackbox fashion as will become clear later.

Format of depth-d block and address. Suppose that a block's logical address is a $\log_2 N$-bit string denoted $\mathsf{addr}^{\langle D \rangle} := \mathsf{addr}[1..(\log_2 N)]$ (expressed in binary format), where $\mathsf{addr}[1]$ is the most significant bit. In general, at depth d, an address $\mathsf{addr}^{\langle d \rangle}$ is the length-$(\log_2 m + d)$ prefix of the full address $\mathsf{addr}^{\langle D \rangle}$. Henceforth, we refer to $\mathsf{addr}^{\langle d \rangle}$ as a depth-d address (or the depth-d truncation of addr).

When we look up a data block, we would look up the full address $\mathsf{addr}^{\langle D \rangle}$ in recursion depth D; we look up $\mathsf{addr}^{\langle D-1 \rangle}$ at depth $D - 1$, $\mathsf{addr}^{\langle D-2 \rangle}$ at depth $D - 2$, and so on. Finally at depth 0, the $\log_2 m$-bit address uniquely determines one of the m blocks stored at OPRAM_0. Since each batch consists of m concurrent lookups, one of them will be responsible for this block in OPRAM_0.

A block with the address $\mathsf{addr}^{\langle d \rangle}$ in OPRAM_d stores the position labels for two blocks in OPRAM_{d+1}, at addresses $\mathsf{addr}^{\langle d \rangle}||0$ and $\mathsf{addr}^{\langle d \rangle}||1$ respectively. Henceforth, we say that the two addresses $\mathsf{addr}^{\langle d \rangle}||0$ and $\mathsf{addr}^{\langle d \rangle}||1$ are *siblings* to each other; $\mathsf{addr}^{\langle d \rangle}||0$ is called the left sibling and $\mathsf{addr}^{\langle d \rangle}||1$ is called the right sibling. We say that $\mathsf{addr}^{\langle d \rangle}||0$ is the left child of $\mathsf{addr}^{\langle d \rangle}$ and $\mathsf{addr}^{\langle d \rangle}||1$ is the right child of $\mathsf{addr}^{\langle d \rangle}$.

Operations Each batch contains m requests denoted as $((\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i) : i \in [m])$, where for $\mathsf{op}_i = \mathsf{read}$, there is no data_i. We perform the following steps.

1. **Conflict resolution.** For every depth $d \in \{0, 1, \ldots, D\}$ in parallel, perform oblivious conflict resolution on the depth-d truncation of all m addresses requested.
   For $d = D$, we suppress duplicate addresses. If multiple requests collide on addresses, we would prefer a write request over a read request (since write requests also fetch the old memory value back before overwriting it with a new value). In the case of concurrent write operations to the same address, we use the properties of the underlying PRAM to determine which write operation prevails.

For $0 \leq d < D$, after conflict resolution, the m requests for OPRAM_d become

$$((\mathsf{addr}_i^{\langle d \rangle}, \mathsf{flags}_i) : i \in [m]),$$

where each non-dummy depth-d truncated address $\mathsf{addr}_i^{\langle d \rangle}$ is distinct and has a two-bit flags_i that indicates whether each of two addresses $(\mathsf{addr}_i^{\langle d \rangle}||0)$ and $(\mathsf{addr}_i^{\langle d \rangle}||1)$ is requested in OPRAM_{d+1}. As noted by earlier works on OPRAM [6, 9, 10], conflict resolution can be completed through $O(1)$ number of oblivious sorting operations. We thus defer the details of the conflict resolution procedure to Appendix A.1.

2. **Fetch.** For $d = 0$ to D sequentially, perform the following:
   - For each $i \in [m]$ in parallel: let $\mathsf{addr}_i^{\langle d \rangle}$ be the depth-d truncation of $\mathsf{addr}_i^{\langle D \rangle}$.
   - Call $\mathsf{OPRAM}_d.\mathsf{Lookup}$ to look up the depth-d addresses $\mathsf{addr}_i^{\langle d \rangle}$ for all $i \in [m]$; observe that position labels for the lookups of non-dummy addresses will be available from the lookup of the previous OPRAM_{d-1} for $d \geq 1$, which is described in the next step. Recall that for OPRAM_0, no position labels are needed.
   - If $d < D$, each lookup from a non-dummy $(\mathsf{addr}_i^{\langle d \rangle}, \mathsf{flags}_i)$ will return two positions for the addresses $\mathsf{addr}_i^{\langle d \rangle}||0$ and $\mathsf{addr}_i^{\langle d \rangle}||1$ in OPRAM_{d+1}. The two bits in flags_i will determine whether each of these two position labels are needed in the lookup of OPRAM_{d+1}.
     We can imagine that there are m CPUs at recursion depth $d+1$ waiting for the position labels corresponding to $\{\mathsf{addr}_i^{\langle d+1 \rangle} : i \in [m]\}$. Now, using oblivious routing (see Section 3.3), the position labels can be delivered to the CPUs at recursion depth $d+1$.
   - If $d = D$, the outcome of Lookup will contain the data blocks fetched.
     Recall that conflict resolution was used to suppress duplicate addresses. Hence, oblivious routing can be used to deliver each data block to the corresponding CPUs that request it.
   - In any case, the freshly fetched blocks are updated if needed in the case of $d = D$, and are placed in OTM'_0 in each OPRAM_d.
3. **Maintain.** We first consider depth D. Set depth-D's update array $U^{\langle D \rangle} := \emptyset$. Suppose that $\ell^{\langle D \rangle}$ is the smallest empty level in OPRAM_D.
   We have the invariant that for all $0 \leq d < D$, if $\ell^{\langle D \rangle} < d$, then $\ell^{\langle D \rangle}$ is also the smallest empty level in OPRAM_d.
   For $d := D$ downto 0, do the following:
   - If $d < \ell^{\langle D \rangle}$, set $\ell := d$; otherwise, set $\ell := \ell^{\langle D \rangle}$.
   - Call $U \leftarrow \mathsf{OPRAM}_d.\mathsf{Shuffle}(U^{\langle d \rangle}, \ell; \mathsf{update})$ where update is the following natural function: recall that in $U^{\langle d \rangle}$ and OPRAM_{d-1}, each depth-$(d-1)$ logical address stores the position labels for both children addresses. For each of the child addresses, if $U^{\langle d \rangle}$ contains a new position label, choose the new one; otherwise, choose the old label previously in OPRAM_{d-1}.

– If $d \geq 1$, we need to send the updated positions involved in U to depth $d-1$.
  We use the Convert subroutine to convert U into an update array for depth-$(d-1)$ addresses, where each entry may pack the position labels for up to two sibling depth-d addresses. Convert can be realized with $O(1)$ oblivious sorting operations and we defer its detailed presentation to Appendix A.2.
  Now, set $U^{\langle d-1 \rangle} \leftarrow \mathsf{Convert}(U, d)$, which will be used in the next iteration for recursion depth $d-1$ to perform its shuffle.

With the above basic OPRAM construction, we can achieve the following theorem whose proof is deferred to Appendix B.

Theorem 10. *The above construction is a perfectly secure OPRAM scheme satisfying the following performance overhead:*

– *When consuming the same number of CPUs as the original PRAM, the scheme incurs $O(\log^3 N)$ simulation overhead;*
– *When the OPRAM is allowed to consume an unbounded number of CPUs, the scheme incurs $O(\log^3 N)$ total work blowup and $O((\log m + \log\log N)\log N)$ depth blowup.*

In either case, the space blowup is $O(1)$.

Proof. We defer the obliviousness proof and performance analysis to Appendix B.

Note that at this moment, even for the sequential special case, we already achieve asymptotic savings over Damgård et al. [12] in terms of space consumption. Furthermore, Damgård et al. [12]'s construction is sequential in nature and does not immediately give rise to an OPRAM scheme.

6 Oblivious Loose Compaction from Expander Graphs

The basic OPRAM construction of Section 5 has a depth blowup $O(\log^2 N)$ assuming that the OPRAM may consume an unbounded number of CPUs — assuming that the original PRAM has sufficient parallelism m, e.g., when $\log m = \Omega(\log N)$. Our next objective is to improve the depth blowup to $\widetilde{O}(\log N)$, but doing so is highly non-trivial. One barrier arises from the online fetch phase: the basic OPRAM scheme of Section 5 requires that position labels be fetched from OPRAM_{d-1} before before fetching OPRAM_d, and consequently fetches are sequential in nature across all logarithmically many recursion depths. Now, Chan et al. [7] proposed an interesting *offline/online* algorithmic paradigm to overcome this problem but achieving only statistical security. We will also rely on an offline/online paradigm, but our instantiation of this paradigm is different from that of Chan et al. [7], not only in that we achieve perfect security (as opposed to statistical), but also that we do so through a new algorithmic abstraction called an "offline/online loose compactor".

Definition: offline/online loose compactor. Let C be an appropriate universal constant. Loose compaction is the following abstraction. Given an input array of $2Cm$ elements out of which at most m are real and the remaining are dummies, construct an output array of length Cm that contains all real elements of the input array padded with dummies. The compaction is "loose" in the sense that although we reduce the number of dummies in the output, we do not completely remove the dummies. In our paper, we define a new abstraction of loose compaction that consists of an offline and an online phase, where the offline phase computes the necessary instructions regarding how to route inputs to outputs, and the online phase performs the actual work of moving elements around.

We will leverage expander graphs and techniques from the self-routing superconcentrator work by Pippenger [36] to construct such an offline-online oblivious loose compaction algorithm.

6.1 Preliminary: Bipartite Expander Graphs

We use $G = (A, B, E)$ to denote a bipartite multi-graph, where A and B are the vertex sets and E is the multi-set of edges between A and B. For $u \in A \cup B$ and multi-set $F \subseteq E$, we denote $F[u]$ as the subset of edges in F that are incident to u. The degree $\mathsf{deg}(u)$ is the number of edges in E incident to u; for $F \subseteq E$, $\mathsf{deg}_F(u) := |F[u]|$. For $S \subset A \cup B$, we use $\mathsf{deg}_S(u)$ to denote the number of edges between u and S.

Explicit construction of expander graphs. Many prior works (e.g., Margulis [34], Gabber and Galil [18], and Jimbo and Maruoka [29]) have shown how to construct bipartite expander graphs with varying parameters.

In particular, based on the explicit construction by Jimbo and Maruoka [29], Pippenger [36, Proposition 4] gave a construction for a family of bipartite graphs with the following properties.

Proposition 1 (Bipartite Expander Graphs). *There exists a universal constant $\mathfrak{d}$ (that is even) such that the following holds. For any square number n, a bipartite multi-graph $G_n = (A, B, E)$ with $|A| = 2n$ and $|B| = n$ can be explicitly constructed such that the following holds.*

1. *For any $a \in A$, $\mathsf{deg}(a) = \mathfrak{d}$; for any $b \in B$, $\mathsf{deg}(b) = 2\mathfrak{d}$.*
2. *For any $R \subset A$ such that $|R| \leq \frac{n}{64}$, define $S := \{b \in B : \mathsf{deg}_R(b) > \frac{\mathfrak{d}}{2}\}$ and $T := \{a \in R : \mathsf{deg}_S(a) > \frac{\mathfrak{d}}{2}\}$. Then, $|T| \leq \frac{|R|}{4}$.*

6.2 Preliminary: A Factory-Facility Problem

Now, let us consider the following factory-facility problem. Suppose $G_n = (A, B, E)$ is the bipartite multi-graph where $|A| = 2n$ and $|B| = n$ as given in Proposition 1. Specifically, we will think of each vertex in A as a factory, and each vertex in B as a storage facility (or facility for short).

Now, some factories in A will manufacture products, and each manufactured product needs be routed to some facility to be stored. Every edge between $a \in A$ and $b \in B$ allows at most one product produced by the factory a to be routed to the facility b. From Proposition 1 we know that each factory $a \in A$ has a total of $\mathfrak{d}$ edges to facilities in B, and each facility $b \in B$ has exactly $2\mathfrak{d}$ edges from factories in A.

Now, imagine that at most $n/64$ factories in A actually end up manufacturing any product at all — if so, we say that such a factory is productive. Moreover, suppose that every productive factory in A produces at most $\mathfrak{d}/2$ products; and each facility in B has a storage capacity of $\mathfrak{d}/2$ as well, i.e., it can receive no more than $\mathfrak{d}/2$ products.

Our goal is to design a (non-uniform) algorithm that is provided with a non-uniform advice string that describes such an expander graph G satisfying Proposition 1:

- *Input.* The algorithm receives as input the number of products produced by each factory in A satisfying the aforementioned requirements;
- *Output.* For each factory, the algorithm will output a set of incident edges for routing each of its products — henceforth the output (for all factories) is referred to as an *assignment.*

An assignment is satisfactory iff 1) all productive factories in A can route *all* of their products to some facility in B (recall that each productive factory produces no more than $\mathfrak{d}/2$ products); and 2) every facility in B receives no more than $\mathfrak{d}/2$ products.

6.3 Preliminary: Pippenger's "Propose-Accept-Finalize" Protocol

Pippenger [36] presented an elegant protocol that solves the factory-facility problem by finding a satisfying assignment, henceforth called the "propose-accept-finalize" protocol. Pippenger's result is described in the form of a protocol (as opposed to a PRAM algorithm) where factories and facilities behave like automatons that interact with each other. Therefore below we will describe Pippenger's result in a protocol format. Although there is a straightforward PRAM algorithm that efficiently emulates Pippenger's protocol, the most naïve PRAM emulation of the protocol is *not* oblivious. In the subsequent Section 6.4, we will instead explain how to efficiently emulate this protocol as an *oblivious* parallel algorithm.

The propose-accept-finalize protocol consists of $O(\log n)$ phases: in each phase, factories first make "proposals" to facilities; the facilities then respond accordingly based on some decision procedure; upon hearing the responses, factories finalize their decisions of this phase. At the end of each phase, some more factories may become satisfied. When the next phase begins, these satisfied factories stop participating. More formally, "propose-accept-finalize" protocol works as follows. We will the notation $\mathsf{req}(a)$ to denote the number of products manufactured by some factory $a \in A$.

ProposeAcceptFinalize: (A protocol for solving the factory-facility problem)
Initially, each factory $a \in A$ with zero requirement $\mathsf{req}(a) = 0$ is *satisfied*; else, it is *unsatisfied*.

Repeat the following for $\lceil \frac{1}{2} \log_2 \frac{n}{64} \rceil$ times:

1. *Propose:* Each unsatisfied factory sends a proposal along each of its incident edges.
2. *Accept:* If a facility $b \in B$ received no more than $\mathfrak{d}/2$ proposals, it sends an acceptance message along each of its $2\mathfrak{d}$ incident edges.
3. *Finalize:* Each currently unsatisfied factory $a \in A$ checks if it received at least $\frac{\mathfrak{d}}{2}$ acceptance messages. If so, it picks an arbitrary subset of the edges over which acceptance messages were received, such that the subset is of size $\mathsf{req}(a)$. The factory records these edges and these edges will be used to route all its products. At this moment, this factory becomes satisfied.

Pippenger [36] proved that if the graph G satisfies Proposition 1, then in every phase of the propose-accept-finalize protocol, at least $3/4$ fraction of the unsatisfied factories will become satisfied. Thus in $O(\log n)$ number of phases, all factories in A will become satisfied. Further, it is not difficult to see that the total number of messages exchanged in the protocol is upper bounded by $O(n)$.

6.4 Oblivious Simulation of the Propose-Accept-Finalize Protocol on a PRAM

We would like to have a (deterministic) parallel algorithm that obliviously emulates the aforementioned propose-accept-finalize protocol. In other words, the algorithm's memory access patterns should not depend on the inputs to the factory-facility problem, i.e., how many products each factory produces (see Section 6.2). Recall that the graph G is a non-uniform advice string provided to the algorithm and it is assumed to be public information.

We will accomplish oblivious simulation of the propose-accept-finalize protocol in two steps. First, we make the protocol communication-oblivious (which we will define shortly below); next, we describe how to obliviously efficiently emulate this communication-oblivious protocol on a PRAM.

Making the Protocol Communication-Oblivious Recall that in the propose-accept-finalize protocol, in each phase not all factories may send messages to facilities and not all facilities may send messages to factories. Therefore, the communication patterns of the protocol (i.e., who talks to whom in each phase) can leak information about the inputs, i.e., how many products are manufactured by each factory in A. Our first step is to transform the protocol into a *communication-oblivious* form, i.e., the protocol's communication patterns must not depend on the inputs.

To this end, our idea is very simple: in each phase, we can have every factory always send a message over each of its incident edges: if the factory is unsatisfied and wants to make proposals, then all messages would be 1; else all messages would be 0. Similarly, every facility should always send a response over each of its incident edges: if the facility wants to accept, all messages would be 1; else all messages would be 0.

Protocol complexity. In this communication-oblivious variant, a factory (or facility) must send (possibly dummy) messages over all incident edges whether or not it actually wants to send messages. It is not hard to see that this modification blows up the original protocol's communication complexity (i.e., total number of messages exchanged) by a logarithmic factor since now in each of the logarithmically many phases, $O(n)$ messages need to be sent.

We thus have that the communication-oblivious propose-accept-finalize protocol completes in $O(\log n)$ rounds and consumes $O(n \log n)$ total messages.

Oblivious Simulation on a PRAM: the Propose-Accept-Finalize Algorithm Once the protocol has been made communication-oblivious, we simulate it on a PRAM as follows. Abstractly, each phase of the protocol consists of 1) message passing between the factories and the facilities; and 2) after receiving messages, local computation performed by the factories or facilities. We thus need to discuss how to emulate both message passing and local computation:

- *Obliviously emulate message passing.* We focus on describing how to emulate (on a PRAM) factories in A sending messages to facilities in B, since the reverse direction is symmetric. Henceforth, imagine that each factory in A and each facility in B is a CPU. To emulate a factory in A sending a message over each of its incident edges, we can imagine that every edge in the graph G_n corresponds to a designated location in memory. Thus a factory $a \in A$ simply writes the message to the memory location corresponding to each edge that a is incident to (in a fixed, predetermined order).
  For every facility $b \in B$ to receive messages collected over all edges it is incident to, we can imagine that a facility $b \in B$ reads the memory locations corresponding to all edges it is incident to (in a fixed, predetermined order), and writes each message fetched (along with its sender) into a local array that is $2\mathfrak{d}$ in length. Henceforth, all computation performed by $b \in B$ will touch only local memory.
- *Obliviously emulate each factory/facility's local computation.* First, assume that in some phase of the communication-oblivious protocol, a facility in B has successfully received messages and stored the received messages in a local array of length $O(1)$. At this point it is not difficult to see that there is a (possibly non-oblivious) algorithm consuming only $O(1)$ time and space for each facility in B to perform its subsequent computation. Such computation can be obliviously simulated in a trivial manner: every memory access can be performed with a linear scan of its local memory, incurring only $O(1)$ blowup.

Henceforth, we refer to the resulting PRAM algorithm as the "*propose-accept-finalize algorithm*".

Obliviousness. Suppose that the protocol being simulated satisfies communication-obliviousness. In the above oblivious simulation of the protocol, the memory access patterns for simulating message passing between vertices in A and B depend only on the communication patterns of the underlying protocol; and all other memory accesses of the algorithm are deterministic and depend only on the length of the input array but not its contents. Thus the following fact is not difficult to see.

Fact 11 *The resulting propose-accept-finalize algorithm has deterministic memory access patterns that do not depend on the inputs (i.e., how many products are produced by each factory).*

PRAM Complexity. It is not difficult to see that the resulting propose-accept-finalize algorithm can be completed in $O(\log n)$ depth and $O(n \log n)$ total work since $\mathfrak{d} = O(1)$.

6.5 Reduction from Loose Compaction to the Factory-Facility Problem

Loose compaction problem. The input is an array Input of length $2Cm$, where $C = 32\mathfrak{d}$ (recall that $\mathfrak{d}$ is the universal constant from Proposition 1) and m is a square number[13], such that at most m entries are real, where the real elements are distinct. The output is an array Output of length Cm that contains all the real entries in the input (while the rest of the entries are dummy). We would like a deterministic algorithm that is oblivious, i.e., its access patterns are deterministic and depend only on the length of the input array but not on the input array's contents.

Offline preparation stage vs. online routing stage. We assume that the input is released in two stages. In the *offline preparation* stage, the algorithm is given the indices of the entries of the input array that contain real elements, but the elements are not available yet. The algorithm might perform some pre-computation in this stage, and we aim for $O(m \log m)$ total work and $O(\log m)$ depth in this stage. At the end of the preparation stage, the algorithm has (obliviously) computed some intermediate data structure Route that will be used in the online routing stage.

In the *online routing* stage, the real elements in the input array are ready, and using the pre-computed Route, they are routed *obliviously* to the output array (of size Cm). We aim to have $O(m)$ total work and $O(1)$ depth for this stage.

Reduction to the factory-facility problem. Our idea is to reduce the loose compaction problem to the factory-facility assignment problem mentioned earlier.

[13] If m is not a square number, we can always round it up to the next square incurring only $O(1)$ blowup.

Let $n := 64m$ and suppose $G_n = (A, B, E)$ is the bipartite multi-graph (which can be constructed explicitly) as in Proposition 1 where $|A| = 2n$ and $|B| = n$.

In the offline stage, the algorithm receives an input $\mathsf{S}[1..2Cm]$ where $\mathsf{S}[i] = 1$ denotes that the i-th element is real — but at this moment, the algorithm has not yet received this element that needs to be routed to the output. The indices of the input array $\mathsf{S}[1..2Cm]$ are partitioned into $2n$ parts, where each part consists of $\frac{\mathfrak{d}}{2}$ consecutive indices. We assign each such part to a vertex in A, and hence, each vertex in A is associated with $\frac{\mathfrak{d}}{2}$ contiguous entries of S. We define the requirement function $\mathsf{req} : A \to \{0, 1, \ldots, \frac{\mathfrak{d}}{2}\}$ such that $\mathsf{req}(a)$ is the number of real entries in the input array associated with a. Observe that at most $m = \frac{n}{64}$ vertices in A have non-zero requirements.

Recall that the factory-facility problem allows each vertex $a \in A$ to choose a multi-set of exactly $\mathsf{req}(a)$ incident edges for routing all real entries assigned to the vertex a. After the routing actually happens, every vertex $b \in B$ receives at most $\mathfrak{d}/2$ real entries of the input array — now if each vertex in B writes down an array of length $\mathfrak{d}/2$ containing all real entries it has received (and padded with dummies to a length of $\mathfrak{d}/2$), then the concatenation of all arrays written down by vertices in B will become the output array containing all real entries of the input array, but whose size is only half that of the input array.

In this way, the offline stage simply calls the oblivious propose-accept-finalize algorithm of Section 6.4 such that each factory in A computes a set of incident edges to route its products. The actual routing of the elements happen in the online stage when the algorithm receives the actual elements to be routed. Since in the offline stage, each vertex in A has learned a set of edges over which to route elements, the online stage can simply execute this plan that the offline stage has decided. To make sure that the algorithm is oblivious, even when a factory in $a \in A$ does not want to route anything over an incident edge, it will send a dummy message over that edge anyway.

Thus we obtain the following theorem:

Theorem 12 (Offline-online oblivious loose compaction). *There exists a (non-uniform) offline-online loose compaction algorithm that is deterministic and oblivious, such that its offline stage completes in $O(m \log m)$ total work and $O(\log m)$ depth; and its online stage completes with $O(m)$ total work and $O(1)$ depth.*

7 Improving the OPRAM's Depth to $\widetilde{O}(\log N)$

In the scheme described in Section 5, each batch of m requests can be served with $O(m \log^3 N)$ amortized total work and (worst-case) depth $O(\log^2 N)$.

We now ask the question, can we improve the OPRAM's depth, i.e., if the OPRAM is allowed to have unbounded number of CPUs, then what parallel runtime can be achieved?

The bottleneck for depth in the previous scheme in Section 5 comes from both the fetch and maintain phases. Both phases process the recursion depths in a sequential manner, the fetch phase from small to large depths d and the maintain phase in the reverse order. Specifically, the fetch phase obtains m blocks (storing position labels) from OPRAM_d for some $d < D$, and then obliviously route the position labels to the next recursion depth OPRAM_{d+1}. Thus, the total depth of the fetch phase is $\Omega(\log m \log N)$ where $\Omega(\log m)$ comes from the depth of obliviously routing m objects to their destinations, and $\Omega(\log N)$ comes from the number of recursion depths d. On the other hand, during the maintain phase, the position-based OPRAMs at various recursion depths perform shuffling in sequential order: each $d \geq 0$, OPRAM_{d+1} would perform reshuffling through $\Omega(1)$ number of oblivious sorts incurring possibly $\Omega(\log m)$ depth (on average), then it emits an update array U to pass to the immediately smaller recursion depth which then embarks on its own shuffling. Thus, in total, the average depth of the maintain phase is $\Omega(\log m \log N)$.

Our goal is to reduce the depth of our OPRAM to $\widetilde{O}(\log N)$ without increasing total work asymptotically, i.e., we would like to shave an additional logarithmic factor off the depth, for the case when m is large (i.e., $\log m = \Theta(\log N)$). As described in the remainder of the section, different techniques are required to improve the depths of the fetch and maintain phases respectively.

7.1 Modifications to the OPRAM's Data Structure

In the improved scheme, the data structure is almost identical to our previous scheme (Section 5) except that now the one-time oblivious memory scheme in each hierarchical level would over-provision by some constant factor C. We note that C is a universal constant that is independent of m or N and its concrete choice is discussed in Section 6.5.

Henceforth in this section, we assume that in each OPRAM_d, each hierarchical level $j \in [d]$ is a (parallel) one-time oblivious memory scheme with at most $2^j \cdot m$ real elements and supporting 2^j batch requests (each having size Cm) as follows:

$$\mathsf{OTM}_j := \mathsf{OTM}^{[2^j \cdot m,\ Cm,\ 2^j]}.$$

7.2 Improving the Depth of the Fetch Phase

To asymptotically improve the depth of the fetch phase, we would like to improve the depth required to route fetched position labels to the next recursion depth. Earlier, we adopted a naïve oblivious routing algorithm which incurs $\Theta(\log m)$ depth.

Our idea is to employ an offline-online oblivious routing algorithm. Although the offline phase still has depth $\Theta(\log m)$, the offline phases among all recursion depths can be performed in parallel.

On the other hand, the online phase still must be performed sequentially among the recursion depths — however, the online routing now consumes only $O(1)$ depth per recursion depth! In the recent work by Chan, Chung, and Shi [7], they also employed a similar offline-online routing paradigm, but their approach incurs (negligibly small) statistical failures and cannot work in our context where perfect security is required. Instead, we devise a new technique that achieves offline-online routing with no failures. Our algorithm employs a loose compaction algorithm as described in Section 6.5 whose construction fundamentally relies on expander graphs [36].

Offline Preparation Recall that a batch access consists of m requests $((\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i) : i \in [m])$. Conflict resolution can be performed in parallel over each recursion depth. At the highest level D, duplicate addresses are suppressed, while for $0 \leq d < D$, the m requests at OPRAM_d are:

$$((\mathsf{addr}_i^{\langle d \rangle}, \mathsf{flags}_i) : i \in [m]),$$

where each non-dummy depth-d truncated address $\mathsf{addr}_i^{\langle d \rangle}$ is distinct and has a two-bit flags_i that indicates whether each of two addresses $(\mathsf{addr}_i^{\langle d \rangle}||0)$ and $(\mathsf{addr}_i^{\langle d \rangle}||1)$ is requested in OPRAM_{d+1}. In the offline preparation stage, the depth-d truncated address and the two-bit flag are available while the data (position labels) is not. The goal of the offline phase is to compute a "route" from depth d to depth $d+1$ for all $0 \leq d < D$. Moreover, this needs to be done in parallel in $O(\log m)$ steps, so that actual routing in the online phase can be performed in $O(1)$ steps per depth. We use the notation that $\overline{\mathsf{Data}}$ denotes the mock copy of Data used in offline preparation.

At depth d, there is a randomly permuted $\mathsf{Receiver}^{\langle d \rangle}$ array which stores the requests for this depth, and the offline phase computes a route to next depth, i.e., $\mathsf{Receiver}^{\langle d+1 \rangle}$. Note that $\mathsf{Receiver}^{\langle d \rangle}$ and $\mathsf{Receiver}^{\langle d+1 \rangle}$ do not store the same set of keys; each block at depth-d stores position labels for two blocks at depth $d+1$. Thus, we need to first convert the keys to the ones at depth $d+1$. This is performed using the depth-d address and the flags at $\mathsf{Receiver}^{\langle d \rangle}$ to obtain an array $\overline{\mathsf{Fetched}}^{\langle \mathsf{d} \rangle}$ of size $2Cm$ but still containing up to m real keys. We now need to reduce the number of keys from $2Cm$ to Cm. We use the offline phase of the offline-online oblivious loose compaction problem (Section 6.5) to do this and (1) obtain the array $\overline{\mathsf{Result}}^{\langle d \rangle}$ of size Cm and (2) the route that will be used by the loose compactor algorithm in the online phase. The routing permutation from $\overline{\mathsf{Result}}^{\langle d \rangle}$ to the next recursion depth $\mathsf{Receiver}^{\langle d+1 \rangle}$ can now be computed using the oblivious routing permutation from Section 3.3.

We now describe the offline phase in detail. During an offline preparation stage, every recursion depth d outputs the following:

1. **Receiver array $\mathsf{Receiver}^{\langle d \rangle}$, for $1 \leq d \leq D$.** This is an array of length Cm such that m of its random locations hold the m requests of OPRAM_d (in random relative order), while the remaining entries hold dummies. We assume that a total ordering is defined on the Cm entries (for instance, even the dummies are uniquely tagged) such that sorting can be carried out with a unique resulting order.
2. **Pre-computed compaction routing information $\mathsf{Route}^{\langle d \rangle}$, for $0 \leq d < D$.**

3. **Routing permutation $\pi^{\langle d \rangle} : [1..Cm] \to [1..Cm]$, for $0 \le d < D$:** This pre-computed routing permutation $\pi^{\langle d \rangle}$ (stored as an array) will be applied in the online phase to route fetched and processed position labels to the next recursion depth $d+1$.

Algorithm. We devise the following offline preparation algorithm. In this algorithm, $\overline{\mathsf{Data}}$ is used to denote mock copy of Data that is used in the offline preparation algorithm.

- *Create randomly permuted receiver array.* For each $1 \le d \le D$, the following can be performed in parallel. Take the batch of m requests (after conflict resolution) in OPRAM_d and extend them to an array of size Cm by inserting entries with dummy addresses at the end.
  We emphasize that later oblivious sort will be performed using the depth-d address as preference. Each dummy address is labeled uniquely according to the relative rank among dummies in the current array. For instance, the first dummy in the array has label $\perp_1$, the second dummy has label $\perp_2$, and so on. This can be achieved by either oblivious prefix sum or oblivious sorting, both of which take $O(m \log m)$ total work and $O(\log m)$ depth.
  Oblivious permutation is applied to the length-Cm array and the resulting array is $\mathsf{Receiver}^{\langle d \rangle}$. For $d = 0$, $\mathsf{Receiver}^{\langle 0 \rangle}$ can be constructed similarly, except that the oblivious permutation step is unnecessary.
- *Emulate online fetch phase.* For each $0 \le d < D$ in parallel, we emulate the fetching phase of OPRAM_d to construct a routing permutation $\pi^{\langle d \rangle} : [1..Cm] \to [1..Cm]$ that will be used to pass the fetched positions to the requests in OPRAM_{d+1}.
  Observe that at this moment, $\mathsf{Receiver}^{\langle d+1 \rangle}$ is already created.
  1. We construct a mock copy of the "result array" denoted $\overline{\mathsf{Fetched}}^{\langle d \rangle}$, which has length $2Cm$. Each entry of $\mathsf{Receiver}^{\langle d \rangle}[1..Cm]$ produces two entries in $\overline{\mathsf{Fetched}}^{\langle d \rangle}$ as follows.
     If $\mathsf{Receiver}^{\langle d \rangle}[j]$ contains a dummy address, then both $\overline{\mathsf{Fetched}}^{\langle d \rangle}[2j-1]$ and $\overline{\mathsf{Fetched}}^{\langle d \rangle}[2j]$ are dummies.
     Otherwise, if $\mathsf{Receiver}^{\langle d \rangle}[j]$ contains a real depth-d address addr_j and flags b_0, b_1. Then, if $b_0 = 1$, then $\overline{\mathsf{Fetched}}^{\langle d \rangle}[2j-1]$ contains the address $\mathsf{addr}_j||0$; else, $\overline{\mathsf{Fetched}}^{\langle d \rangle}[2j-1]$ is dummy. Similarly, if $b_1 = 1$, then $\overline{\mathsf{Fetched}}^{\langle d \rangle}[2j]$ contains $\mathsf{addr}_j||1$; else, $\overline{\mathsf{Fetched}}^{\langle d \rangle}[2j]$ is dummy.
  2. Apply the offline preparation stage of the loose compaction algorithm using $\overline{\mathsf{Fetched}}^{\langle d \rangle}[1..2Cm]$ as input to produce routing information at depth d.
  3. Apply the online routing stage of the loose compaction algorithm on $\overline{\mathsf{Fetched}}^{\langle d \rangle}[1..2Cm]$ using the routing information at depth d computed in the offline stage to produce $\overline{\mathsf{Result}}^{\langle d \rangle}[1..Cm]$.
     Observe that $\overline{\mathsf{Result}}^{\langle d \rangle}[1..Cm]$ contains all the real entries of $\overline{\mathsf{Fetched}}^{\langle d \rangle}[1..2Cm]$. Moreover, each dummy in $\overline{\mathsf{Result}}^{\langle d \rangle}[1..Cm]$ is uniquely tagged according

to the relative rank among dummies in the current array. For instance, the first dummy in the array is labeled $\perp_1$, the second dummy is labeled $\perp_2$, and so on. This can be achieved by either oblivious prefix sum or oblivious sorting, both of which take $O(m \log m)$ total work and $O(\log m)$ depth.

– *Compute routing permutation to next recursion depth.* We next (obliviously) compute a routing permutation $\pi^{\langle d \rangle} : [1..Cm] \to [1..Cm]$ that is supposed to match each entry in $\overline{\mathsf{Result}}^{\langle d \rangle}[1..Cm]$ to the corresponding one in $\mathsf{Receiver}^{\langle d+1 \rangle}[1..Cm]$. Observe that $\overline{\mathsf{Result}}^{\langle d \rangle}$ and $\mathsf{Receiver}^{\langle d+1 \rangle}$ have the same set of real elements, and the dummies also have the same set of labels. Hence, we can use the oblivious algorithm for computing the routing permutation described in Section 3.3.
  The routing permutation $\pi^{\langle d \rangle} : [1..Cm] \to [1..Cm]$ can be obliviously computed such that for each $i \in [1..Cm]$, $\overline{\mathsf{Result}}^{\langle d \rangle}[i]$ and $\mathsf{Receiver}^{\langle d+1 \rangle}[\pi^{\langle d \rangle}[i]]$ have the same real or dummy label.

Lemma 2. *The offline preparation step takes $O(\log m)$ depth and $O(Dm \log m)$ total work.*

Online Fetch and Route In the online phase, each recursion depth fetches the position labels needed for the next recursion depth, and routes them in a single parallel step to the next recursion depth. We describe the detailed algorithm below.

For d from 0 to D sequentially, perform the following:

1. Recall that $\mathsf{Receiver}^{\langle d \rangle}$ is an array of length Cm and m of its entries store the m (conflict resolved) requests in OPRAM_d.
   For each $j \in [1..Cm]$, the j-th entry of $\mathsf{Receiver}^{\langle d \rangle}$ contains an address addr_j, where a non-dummy addr_j is a depth-d address in OPRAM_d. Moreover, for $0 \le d < D$, the entry also contains two bits flags_j indicating whether the two depth-$(d+1)$ addresses $\mathsf{addr}_j||0$ and $\mathsf{addr}_j||1$ are needed in OPRAM_{d+1}.
   For $d = 0$, no position label is needed in OPRAM_0, which is simply an array of length m. For $1 \le d \le D$, we shall see that the step below ensures that the previous iteration has delivered the correct position pos_j to each non-dummy address addr_j in $\mathsf{Receiver}^{\langle d \rangle}$.
   Therefore, we call $\mathsf{OPRAM}_d.\mathsf{Lookup}(\{(\mathsf{addr}_j, \mathsf{pos}_j) : j \in [Cm]\})$. Since OPRAM_d consists of $d+1$ one-time oblivious memory data structures, this step takes $O(\log d)$ depth and $O(md)$ total work.
2. If $d = D$, then the result of the Lookup returns the values of the requested addresses. Then, oblivious routing can be used to deliver the blocks to the corresponding requesting CPUs; the whole fetch phase is completed.
   For $0 \le d < D$, for each $j \in [1..Cm]$, if addr_j is a non-dummy address, then the position labels $(\mathsf{pos}_j^0, \mathsf{pos}_j^1)$ of the depth-$(d+1)$ addresses $\mathsf{addr}_j||0$ and $\mathsf{addr}_j||1$ are returned.

We next construct an array $\mathsf{Fetched}^{\langle d\rangle}[1..2Cm]$. For each $j \in [1..Cm]$, the information $(\mathsf{addr}_j, \mathsf{flags}_j, (\mathsf{pos}_j^0, \mathsf{pos}_j^1))$ is used to create the two entries in $\mathsf{Fetched}^{\langle d\rangle}$ with indices $2j-1+b$, where $b \in \{0,1\}$, in the following way.
If flags_j indicates that $\mathsf{addr}_j||b$ is not requested in OPRAM_{d+1}, then the entry $\mathsf{Fetched}^{\langle d\rangle}[2j-1+b]$ is dummy; otherwise, $\mathsf{Fetched}^{\langle d\rangle}[2j-1+b]$ contains the pair $(\mathsf{addr}_j||b, \mathsf{pos}_j^b)$, where pos_j^b is the correct position label of the depth-$(d+1)$ address $\mathsf{addr}_j||b$ in OPRAM_{d+1}.
3. Using the pre-computed routing information at depth d, apply the routing online stage of the loose compaction algorithm to $\mathsf{Fetched}^{\langle d\rangle}[1..2Cm]$ to (obliviously) produce $\mathsf{Result}^{\langle d\rangle}[1..Cm]$.
4. Using the pre-computed permutation $\pi^{\langle d\rangle} : [1..Cm] \to [1..Cm]$, for each $j \in [1..Cm]$ in parallel, send the contents of $\mathsf{Result}^{\langle d\rangle}[j]$ to $\mathsf{Receiver}^{\langle d+1\rangle}[\pi^{\langle d\rangle}[j]]$. Observe that this step delivers the correct position labels for OPRAM_{d+1}; moreover, the permutation $\pi^{\langle d\rangle}$ is revealed. As mentioned above, since the elements in $\mathsf{Receiver}^{\langle d+1\rangle}$ have been permuted uniformly at random independently, $\pi^{\langle d\rangle}$ looks uniformly at random and independent to the adversary.

Lemma 3. *The online fetch and route step takes $O(D \log D)$ depth and $O(mD^2)$ total work.*

Proof. For each d, OPRAM_d has d levels. Hence, a batch of m concurrent lookups in OPRAM_d takes $O(\log d)$ depth and $O(md)$ total work. Observe that using the pre-computed information in the offline step, routing between successive recursive OPRAM's takes $O(1)$ depth and $O(m)$ total work. Summing up from $d = 0$ to D gives the result.

7.3 Improving the Depth of the Maintain Phase

Our earlier maintain phase algorithm (see Section 5) performs shuffling for each recursion depth OPRAM_d sequentially, starting from $d = D$ to 0. Specifically, the OPRAM_d must wait for the updated position labels of addresses in OPRAM_{d+1} before it begins reshuffling. Since the shuffling for each recursion depth takes $\Omega(\log N)$ depth, the sequential nature of the execution over all $\Theta(\log N)$ recursion depths lead to a total depth of $\Omega(\log^2 N)$.

Intuition. To improve the depth of the maintain phase to $O(\log N)$, we would like to perform shuffling for different recursion depths in parallel. Our idea is to separate the algorithm into a "mock shuffling" stage and an "index update" stage.

1. **Mock shuffle.** The mock shuffling stage achieves the following:
   – For $1 \leq d \leq D$, at the end of the mock shuffling stage, OPRAM_d will have finished building a one-time oblivious memory OTM_ℓ at some level ℓ, even though the contents of the stored blocks might be incorrect.
   However, the positions of the depth-d addresses stored in this OTM_ℓ will later be passed to OPRAM_{d-1} at the beginning of the index update stage.

  - As mentioned, after the mock shuffling stage, the contents of the blocks stored in the freshly built OTM_ℓ in OPRAM_d might be incorrect. Recall that the content of each block in OPRAM_d is supposed to store the positions of two depth-$(d+1)$ addresses in OPRAM_{d+1}.
    This information on the updated positions of the depth-$(d+1)$ addresses will be available at the end of the mock shuffling stage.
2. **Index update.** For $0 \leq d < D$, OPRAM_d receives the updated positions of the depth-$(d+1)$ addresses from OPRAM_{d+1}, which are available at the end of the mock shuffling stage. These updated positions are the correct contents of the blocks stored in the OTM_ℓ that is freshly built in OPRAM_d (where OPRAM_0 is the special case with just one array of length m).
   Observe that the contents of the blocks in this OTM_ℓ in OPRAM_d can be updated correctly using oblivious sorting.

Algorithm. We now describe the improved maintain-phase algorithm.

1. **Initialize.** For depth D, for every $i \in [m]$, let $u_i := (\mathsf{addr}_i^{\langle D \rangle}, \mathsf{data}_i)$, where $\mathsf{addr}_i^{\langle D \rangle}$'s are the depth-$D$ addresses after conflict resolution, and data_i is the updated content for the corresponding address $\mathsf{addr}_i^{\langle D \rangle}$. Denote $U^{\langle D \rangle} := (u_i : i \in [m])$.
   For depth $1 \leq d < D$, set $U^{\langle d \rangle} := \emptyset$.
   Suppose that $\ell^{\langle D \rangle}$ is the smallest available level in OPRAM_D; if every level is full, then set $\ell^{\langle D \rangle} \leftarrow D$.
2. **Mock shuffle.** For $1 \leq d \leq D$ in parallel:
   - Set $\ell_d := \min\{\ell^{\langle D \rangle}, d\}$.
   - Call $U \leftarrow \mathsf{OPRAM}_d.\mathsf{Shuffle}(U^{\langle d \rangle}, \ell_d)$
   - Compute $\widehat{U}_{d-1} \leftarrow \mathsf{Convert}(U, d)$.
3. **Index update.** For $0 \leq d < D$ in parallel:
   - In OPRAM_d, let A denote the data structure corresponding to OTM_{ℓ_d}, i.e., the array at level ℓ_d of the hierarchy. At this moment, the positions of the blocks with real addresses in A have already been determined, but the contents of these blocks might need to be updated with $\widehat{U}_d$ generated from OPRAM_{d+1} in the mock shuffling stage.
   - Relying on oblivious routing where $\widehat{U}_d$ acts as the source and A acts as the destination, depth-d addresses are used to send the contents of each entry in $\widehat{U}_d$ to the corresponding entry in A.
     After the entries in A receives the correct updated contents (which are position labels for addresses in OPRAM_{d+1}), it becomes the new data structure for OTM_{ℓ_d}.

Lemma 4. *The maintain phase has $O(\log N)$ depth.*

Proof. Since the D recursive OPRAM's are operated in parallel, it suffices to analyze the depth incurred by the largest OPRAM_D, which stores $O(N)$ blocks. Therefore, the oblivious shuffling and routing subroutines involved (which make use of oblivious sorting) have $O(\log N)$ depth.

7.4 Obliviousness

We next argue why our scheme satisfies obliviousness. Our argument follows the same approach used in Chan et al. [7]. The difference is that the loose compaction algorithm used here is perfectly secure.

The security of the improved OPRAM scheme is based on that of the basic OPRAM scheme in Section 5. From the description of the scheme, most parts of the scheme have deterministic access pattern that does not depend on the requested addresses. The only part of the access pattern that has randomness involves the subroutine oblivious random permutation (which by construction satisfies obliviousness) and the online routing of information between successive depths of recursive OPRAM's in the fetch phase. Hence, it suffices to show that the online fetch and route procedure is also secure.

Lemma 5 (Security of Position Identifiers Routing). *In the online fetch and route procedure described in Section 7.2, the resulting distribution of physical access pattern is independent of the requested addresses.*

Proof. It suffices to check that in the description of the scheme, the physical memory are accessed using the building blocks described in Section 3.3, which ensure that the access pattern is independent of the requested addresses. We next inspect each step more carefully.

Fix some $0 \leq d < D$. In the offline phase, the elements in the array $\mathsf{Receiver}^{\langle d+1 \rangle}$ have been randomly permuted in an oblivious manner using fresh randomness.

Therefore, the routing permutation $\pi^{\langle d \rangle}$ (that can be observed by the adversary later in the online phase) is a uniformly random permutation, even when conditioned on having observed the access patterns of the oblivious random permutation in the offline phase — note that this is implied by our formal definition of oblivious random permutation (in Section 3.3).

Other steps in the procedure invokes subroutines described in Section 3.3, which produces deterministic access pattern independent of the requested addresses.

In the online phase, the only part of the procedure that involves randomness concerns the routing of information from OPRAM_d to OPRAM_{d+1}, for $0 \leq d < D$.

As mentioned earlier, the routing permutation $\pi^{\langle d \rangle}$ is revealed, but it has an independent uniform distribution, because the destination array $\mathsf{Receiver}^{\langle d+1 \rangle}$ was permuted using a (secret) fresh random permutation.

With Lemma 5 and combining the security argument of basic OPRAM scheme in Section 5, it follows that our improved small-depth OPRAM construction is indeed perfectly secure, i.e., the following Theorem 13 holds.

Theorem 13 (Perfectly secure, small-depth OPRAM). *There exists a perfectly secure OPRAM scheme (for general block sizes) with $O(\log^3 N)$ total work blowup, $O(\log N \log\log N)$ depth blowup, and $O(1)$ space blowup; moreover, each CPU in the OPRAM consumes only $O(1)$ blocks of private cache.*

8 Conclusion and Future Work

In this paper, we constructed a perfectly secure OPRAM scheme with $O(\log^3 N)$ total work blowup, $O(\log N \log\log N)$ depth blowup, and $O(1)$ space blowup. To the best of our knowledge our scheme is the first perfectly secure (non-trivial) OPRAM scheme, and even for the sequential special case we asymptotically improve the space overhead relative to Damgård et al. [12]. Prior to our work, the only known perfectly secure ORAM scheme is that by Damgård et al. [12], where they achieve $O(\log^3 N)$ simulation overhead and $O(\log N)$ space blowup. No (non-trivial) OPRAM scheme was known prior to our work, and in particular the scheme by Damgård et al. [12] does not appear amenable to parallelization. Finally, in comparison with known statistically secure OPRAMs [9,42], our work removes the dependence (in performance) on the security parameter; thus we in fact asymptotically outperform known statistically secure ORAMs [42] and OPRAMs [9] when (sub-)exponentially small failure probabilities are required.

Exciting questions remain open for future research:

– Are there any separations between the performance of perfectly secure and statistically secure ORAMs/OPRAMs?
– Can we construct perfectly secure ORAMs/OPRAMs whose total work blowup matches the best known statistically secure ORAMs/OPRAMs assuming negligible security failures?
– Can we construct perfectly secure ORAM/OPRAM schemes whose concrete performance lends to deployment in real-world systems?

Acknowledgments

T-H. Hubert Chan was supported in part by the Hong Kong RGC under grant 17200418. This work is supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, CNS-1601879, CNS-1617676, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a DARPA Safeware grant (subcontractor under IBM), a Sloan Fellowship, Google Faculty Research Awards, a Google Ph.D. Fellowship Award, a Baidu Research Award, and a VMware Research Award.

We gratefully acknowledge Shai Halevi and Craig Gentry for helpful discussions and for suggesting the use of expander graphs to achieve low-online-depth routing of position labels. We are extremely grateful to Bruce Maggs for most patiently explaining Pippenger's result [36] to us and answering many of our technical questions. We acknowledge Kai-Min Chung for many helpful technical discussions regarding perfectly secure ORAM and OPRAM. We thank Ling Ren for many early discussions on perfectly secure ORAMs. We thank Muthuramakrishnan Venkitasubramaniam, Antigoni Polychroniadou, and Kai-Min Chung for helpful discussions on the significance of achieving perfect security in cryptographic primitives, and for helpful editorial comments. Elaine Shi is grateful to Bruce Maggs, Bobby Bhattacharjee, Kai-Min Chung, and Feng-Hao Liu for their unwavering moral support during the period this research was conducted.

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. An O(N log N) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.
2. Miklós Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, pages 181–190, New York, NY, USA, 2010. ACM.
3. Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Oblivious computation with data locality. *IACR Cryptology ePrint Archive*, 2017:772, 2017.
4. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, 1988.
5. Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 742–762, 2015.
6. Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 175–204, 2016.
7. T-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In *Asiacrypt*, 2017.
8. 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*, 2017.
9. T-H. Hubert Chan and Elaine Shi. Circuit OPRAM: A unifying framework for computationally and statistically secure ORAMs and OPRAMs. In *TCC*, 2017.
10. Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 205–234, 2016.
11. Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *Asiacrypt*, 2014.
12. Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography Conference (TCC)*, pages 144–163, 2011.
13. Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, San Diego, CA, August 2014. USENIX Association.
14. Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. Cryptology ePrint Archive, Report 2017/749, 2017. `https://eprint.iacr.org/2017/749`.
15. 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*, 2015.

16. Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.
17. Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.
18. Ofer Gabber and Zvi Galil. Explicit constructions of linear-sized superconcentrators. *Journal of Computer and System Sciences*, 22(3):407–420, June 1981.
19. Daniel Genkin, Yuval Ishai, and Mor Weiss. Binary AMD circuits from secure multiparty computation. In *Theory of Cryptography Conference*, pages 336–366. Springer, 2016.
20. Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.
21. O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1987.
22. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.
23. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 576–587, 2011.
24. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 157–167, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.
25. S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
26. Torben Hagerup. Fast and optimal simulations between CRCW PRAMs. In *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, pages 45–56, 1992.
27. Torben Hagerup. The log-star revolution. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '92, pages 259–278, London, UK, UK, 1992. Springer-Verlag.
28. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 406–425. Springer, 2011.
29. Shuji Jimbo and Akira Maruoka. Expanders obtained from affine transformations. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 88–97, New York, NY, USA, 1985. ACM.
30. Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.
31. Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Not.*, 50(4):87–101, March 2015.

32. Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376, 2015.
33. Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.
34. G. A. Margulis. Explicit construction of concentrators. *Problems of Information Transmission*, 9(4):325–332, 1973.
35. Kartik Nayak and Jonathan Katz. An oblivious parallel RAM with $o(log^2 n)$ parallel runtime blowup. *IACR Cryptology ePrint Archive*, 2016:1141, 2016.
36. Nicholas Pippenger. Self-routing superconcentrators. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 355–361, New York, NY, USA, 1993. ACM.
37. Michael Raskin and Mark Simkin. Oblivious ram with small storage overhead. Cryptology ePrint Archive, Report 2018/268, 2018. https://eprint.iacr.org/2018/268.
38. 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 *ISCA*, pages 571–582, 2013.
39. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O(\log^3 N)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.
40. Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (S & P)*, 2013.
41. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious RAM protocol. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.
42. Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS*, 2015.
43. Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

A Additional Algorithmic Details

For ease of understanding, we graphically illustrate our OPRAM's data structure in Figure 1. In the remainder of this section, we supply some missing algorithmic details.

A.1 Conflict Resolution

For completeness, we briefly describe the conflict resolution procedure for $1 \leq d < D$ as follows:

1. Consider the depth-$(d+1)$ truncated address: $A^{\langle d+1 \rangle} := (\mathsf{addr}_1^{\langle d+1 \rangle}, \ldots, \mathsf{addr}_m^{\langle d+1 \rangle})$, and use oblivious sorting to suppress duplicates of depth-$(d+1)$ addresses,

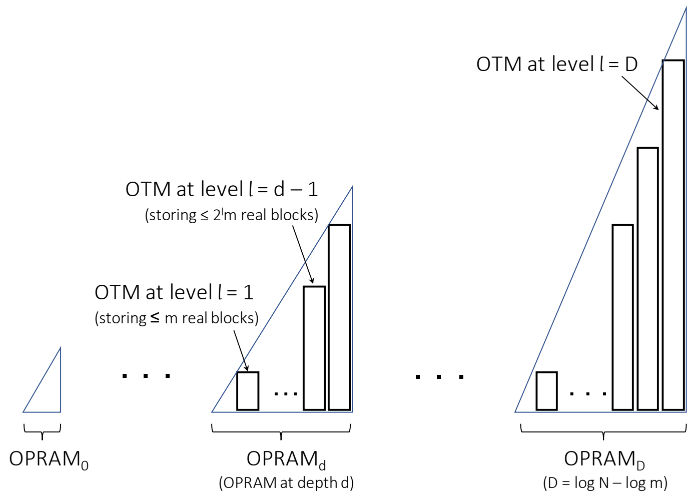

Fig. 1: **OPRAM data structures at a glance.** Each OTM is a one-time memory instance defined and constructed in Section 4; each $\mathsf{OPRAM}_0, \mathsf{OPRAM}_1, \ldots, \mathsf{OPRAM}_D$ is a position-based OPRAM defined and constructed in Section 5.1.

i.e., each repeated depth-$(d+1)$ address is replaced by a dummy. Let $\widehat{A}^{\langle d+1 \rangle}$ be the resulting array (of size m) sorted by the (unique) depth-$(d+1)$ addresses.

2. For each $i \in [1..m]$, we produce an entry $(\mathsf{addr}_i, \mathsf{flags}_i)$ according to the following rules:
   (a) If $\mathsf{addr}_i^{\langle d+1 \rangle}$ is a dummy, then $\mathsf{addr}_i := \perp$ is also dummy.
   (b) If $\mathsf{addr}_i^{\langle d+1 \rangle}$ does not share its length-d prefix with $\mathsf{addr}_{i-1}^{\langle d+1 \rangle}$ or $\mathsf{addr}_{i+1}^{\langle d+1 \rangle}$, then addr_i is set to be the length-d prefix of $\mathsf{addr}_i^{\langle d+1 \rangle}$. Moreover, if $\mathsf{addr}_i^{\langle d+1 \rangle}$ ends with 0, then $\mathsf{flags}_i := 10$; otherwise, $\mathsf{flags}_i := 01$.
   (c) If $\mathsf{addr}_i^{\langle d+1 \rangle}$ and $\mathsf{addr}_{i-1}^{\langle d+1 \rangle}$ share the same length-d prefix, then $\mathsf{addr}_i := \perp$; otherwise, if $\mathsf{addr}_i^{\langle d+1 \rangle}$ and $\mathsf{addr}_{i+1}^{\langle d+1 \rangle}$ share the same length-d prefix, then addr_i is set to the shared length-d prefix of the address, and $\mathsf{flags}_i := 11$.
3. Then, the batch access for OPRAM_d is $((\mathsf{addr}_i, \mathsf{flags}_i) : i \in [m])$.

A.2 The Convert Subroutine

The Convert subroutine takes an array that stores the position labels within OPRAM_d for depth-d addresses, and converts the array to one that contains depth-$(d-1)$ addresses where each entry may pack up to two position labels for its child addresses at depth-d.

The subroutine $\mathsf{Convert}(U, d)$ proceeds as follows. First, perform oblivious sort on the depth-d addresses to produce an array denoted as $\{(\mathsf{addr}_i^{\langle d \rangle}, \mathsf{pos}_i) : i \in [|U|]\}$.

Next, for $i \in [|U|]$ in parallel, look to the left and look to the right and do the following:

– If $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||0$ and $\mathsf{addr}_{i+1}^{\langle d \rangle} = \mathsf{addr}||1$ for some addr, i.e., if my right neighbor is my sibling, then write down $u'_i = (\mathsf{addr}, (\mathsf{pos}_i, \mathsf{pos}_{i+1}))$, i.e., both siblings' positions need to be updated.
– If $\mathsf{addr}_{i-1}^{\langle d \rangle} = \mathsf{addr}||0$ and $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||1$ for some addr, i.e., if my left neighbor is my sibling, then write down $u'_i = \perp$.
– Else if i does not have a neighboring sibling, parse $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||b$ for some $b \in \{0, 1\}$, then write down $u'_i = (\mathsf{addr}, (\mathsf{pos}_i, *))$ if $b = 0$ or write down $u'_i = (\mathsf{addr}, (*, \mathsf{pos}_i))$ if $b = 1$. In these cases, only the position of one of the siblings needs to be updated in OPRAM_{d-1}.
– Let $U^{\langle d-1 \rangle} := \{u'_i : i \in [|U|]\}$. Note here that each entry of $U^{\langle d-1 \rangle}$ contains a depth-$(d-1)$ address of the form addr, as well as the update instructions for two position labels of the depth-d addresses $\mathsf{addr}||0$ and $\mathsf{addr}||1$ respectively. We emphasize that when $*$ appears, this means that the position of the corresponding depth-d address does not need to be updated in OPRAM_{d-1}.
– Output $U^{\langle d-1 \rangle}$.

B Basic OPRAM Scheme: Analysis and Extensions

We now give detailed analysis and proofs for our basic OPRAM scheme in Section 5.

B.1 Correctness and Obliviousness

Fact 14 *The above construction maintains correctness. More specifically, at every recursion depth d, the correct position labels will be input to the* Lookup *operations of* OPRAM_d*; and every batch of requests will return the correct answers.*

Proof. Straightforward by construction.

In our OPRAM construction, for every OPRAM_d at recursion depth d, the following invariants are respected by construction as stated in the following facts.

Fact 15 *For every* OPRAM_d*, every* OTM_i *instance at level $i \leq d$ that is created needs to answer at most 2^i batches of m requests before* OTM_i *instance is destroyed.*

Proof. For every OPRAM_d, the following is true: imagine that there is a $(d+1)$-bit binary counter initialized to 0 that increments whenever a batch of m requests come in. Now, for $0 \leq \ell < d$, whenever the ℓ-th bit flips from 1 to 0, the ℓ-th level of OPRAM_d is destroyed; whenever the ℓ-th bit flips from 0 to 1, the ℓ-th level of OPRAM_d is reconstructed. For the largest level d of OPRAM_d, whenever the d-th (most significant) bit of this binary counter flips from 0 to 1 or from 1 to 0, the $(d+1)$-th level is destroyed and reconstructed. The fact follows in a straightforward manner by observing this binary-counter argument.

Fact 16 *For every OPRAM_d and every OTM_ℓ instance at level $\ell \leq d$, during the lifetime of the OTM_ℓ instance: (a) no two real requests will ask for the same depth-d address; and (b) for every request that asks for a real depth-d address, the address must exist in OTM_i.*

Proof. We first prove claim (a). Observe that for any OPRAM_d, if some depth-d address $\mathsf{addr}^{\langle d\rangle}$ is fetched from some level $\ell \leq d$, at this moment, $\mathsf{addr}^{\langle d\rangle}$ will either enter a smaller level $\ell' < \ell$; or some level $\ell'' \geq \ell$ will be rebuilt and $\mathsf{addr}^{\langle d\rangle}$ will go into level ℓ'' — in the latter case, level ℓ will be destroyed prior to the rebuilding of level ℓ''. In either of the above cases, due to correctness of the construction, if $\mathsf{addr}^{\langle d\rangle}$ is needed again from OPRAM_d, a correct position label will be provided for $\mathsf{addr}^{\langle d\rangle}$ such that the request will not go to level ℓ (until the level is reconstructed). Moreover, two real requests will not appear in the same request due to the conflict resolution procedure. Finally, claim (b) follows from correctness of the position labels.

Given the above facts, our construction maintains perfect obliviousness.

Lemma 6 (Obliviousness). *The above OPRAM construction satisfies perfect obliviousness.*

Proof. For every parallel one-time memory instance constructed during the lifetime of the OPRAM, Facts 15 and 16 are satisfied, and thus every one-time memory instance receives a valid request sequence. The lemma then follows in a straightforward fashion by the perfect obliviousness of the parallel one-time memory scheme, and by observing that all other access patterns of the OPRAM construction are deterministic and independent of the input requests.

B.2 Asymptotic Complexity

We now analyze the asymptotic efficiency of our OPRAM construction. First, observe that the asymptotic performance of the fetch phase as stated in the following fact.

Fact 17 *The fetch phase can be completed in $O(m \log^2 N)$ total work, and in $O((\log m + \log\log N) \cdot \log N)$ depth (assuming an unbounded number of CPUs).*

Proof. For total work, it is not difficult to see that one $\log N$ factor arises from the recursion depths, and within each recursion depth it takes $O(m \log N + m \log m)$ work to perform the fetch. Here, $m \log m$ is the total work incurred by the oblivious routing in between recursion depths and $m \log N$ is the work incurred within a single position-based OPRAM.

For depth, one $\log N$ factor comes from the $\log N$ recursion depths, the other $(\log m + \log\log N)$ factor is due to the depth incurred by each recursion depth as well as due to the routing in between depths: 1) Within each recursion depth, it takes $O(1)$ depth to look up each of the up to $O(\log N)$ hierarchical levels, and then select the correct result in another $O(\log\log N)$ depth; and 2) the routing between adjacent depths can be implemented with the AKS sorting network [1] that takes $O(\log m)$ depth.

We now proceed to analyze the efficiency of the maintain phase.

Fact 18 *Averaging over a sequence of batch accesses, the maintain phase costs $O(m \log^3 N)$ amortized total work (except with negligible in N probability). Further, for each batch of accesses, the maintain phase can always be completed in $O(\log^2 N)$ depth assuming an unbounded number of CPUs.*

Proof. For each OPRAM_d, every level $\ell \leq d+1$ must be rebuilt after every 2^ℓ batch of m requests. Due to Fact 9, each rebuilding operation will take $O(2^\ell \cdot m \log(2^\ell \cdot m))$ total work, and has depth $O(\log(2^\ell \cdot m))$, which is at most $O(\log N)$. After the rebuilding, the $\mathsf{Convert}$ algorithm also has the same asymptotic performance. Thus, for each recursion depth, the amortized total work is $O(m \log^2 N)$. Counting all $O(\log N)$ recursion depths, we have the desired result for total work.

For depth, observe that for each recursion depth, the depth incurred by the rebuilding is dominated by the depth of the AKS sorting network which is $O(\log N)$. We then have the depth result by observing that the maintain phase is performed sequentially over $O(\log N)$ recursion depths.

Lemma 7. *In the above OPRAM construction, the total work blowup is $O(\log^3 N)$, and the depth blowup is $O((\log m + \log\log N)\log N)$.*

Proof. Straightforward from Facts 17 and 18.

Corollary 1. *The above OPRAM construction incurs $O(\log^3 N)$ simulation overhead when consuming the same number of CPUs as the original PRAM.*

Proof. This corollary is implied directly by Lemma 7. The difference is that Lemma 7 would require more than m CPUs such that the depth of the algorithm may be smaller than the total work blowup, but if we are constrained to exactly m CPUs, the amortized parallel runtime per batch of accesses would be exactly $O(\log^3 N)$.

B.3 Extension: Results for Large Block Sizes

Observe that if the block size is large, then each block in OPRAM_d can store more position identifiers for blocks in OPRAM_{d+1}. Hence, the number D of recursive OPRAMs can be reduced. This can lead to the following improvement.

Corollary 2 (Large Block Size). *Suppose the block size is $\Theta(N^\epsilon)$ bits. Then, the above OPRAM construction can be modified to have $O(\frac{1}{\epsilon}\log^2 N)$ total work blowup and simulation overhead, and $O(\frac{1}{\epsilon}(\log m + \log\log N))$ depth blowup.*

Proof. When the block size is $B := \Theta(N^\epsilon)$ bits, the number of depths of recursive OPRAMs becomes $D := \frac{\log N}{\log \frac{B}{\log N}} = O(\frac{1}{\epsilon})$.

Hence, in every performance metric stated in Lemma 7 and Corollary 1, one factor of $\log N$ is replaced with $O(\frac{1}{\epsilon})$.