Perfectly Secure Oblivious Parallel RAM with $O(\log^3 N/\log\log N)$ Overhead

T-H. Hubert Chan
HKU

Wei-Kai Lin
Cornell

Kartik Nayak
Duke

Elaine Shi
Cornell

hubert@cs.hku.hk, wklin@cs.cornell.edu, kartik@cs.duke.edu, runting@gmail.com

Abstract

Oblivious RAM (ORAM) is a technique for compiling any program to an oblivious counterpart, i.e., one whose access patterns do not leak information about the secret inputs. Similarly, Oblivious Parallel RAM (OPRAM) compiles a *parallel* program to an oblivious counterpart. In this paper, we care about ORAM/OPRAM with *perfectsecurity*, i.e., the access patterns must *identically distributed* no matter what the program's memory request sequence is. We show two novel results.

The first result is a new perfectly secure OPRAM scheme with $O(\log^3 N/\log\log N)$ *expected* overhead. In comparison, the prior literature has been stuck at $O(\log^3 N)$ for more than a decade.

The second result is a new perfectly secure OPRAM scheme with $O(\log^4 N/\log\log N)$ *worst-case* overhead. To the best of our knowledge, this is the first perfectly secure OPRAM scheme with polylogarithmic *worst-case* overhead. Prior to our work, the state of the art is a perfectly secure ORAM scheme with more than $\sqrt{N}$ worst-case overhead, and the result does not generalize to a parallel setting. Our work advances the theoretical understanding of the asymptotic complexity of perfectly secure OPRAMs.

1 Introduction

Oblivious RAM (ORAM) is an algorithmic construction that provably obfuscates a (parallel) program's access patterns. It was first proposed in the ground-breaking work by Goldreich and Ostrovsky [GO96, Gol87], and its parallel counterpart Oblivious Parallel ORAM (OPRAM) was proposed by Boyle et al. [BCP16]. ORAM and OPRAM are fundamental building blocks for enabling various forms of secure computation on sensitive data, e.g., either through trusted-hardware [RYF+13, FRY+14, MLS+13, LHM+15] or relying on cryptographic multi-party computation [GKK+12, LWN+15]. Since their proposal, ORAM and OPRAM have attracted much interest from various communities, and there has been a line of work dedicated to understanding their asymptotic and concrete efficiencies. It is well-known [GO96, Gol87, LN18] that any ORAM/OPRAM scheme must incur at least a logarithmic *blowup* (also known as *overhead* or *simulation overhead*) in total work relative to the insecure counterpart. On the other hand, ORAM/OPRAM schemes with poly-logarithmic overhead have been known [GO96, Gol87, GM11, KLO12, SCSL11, SvDS+13, WCS15, PPRY18], and the very recent exciting work of Asharov et al. [AKL+20a] showed how to match the logarithmic lower bound in the sequential ORAM setting, assuming the existence of one-way functions and a computationally bounded adversary.[1]

[1] For the parallel setting, how to achieve optimality remains open.

Motivation for perfectly secure ORAMs/OPRAMs. With the exception of very few works, most of the literature has focused on either *computationally* secure [GO96, Gol87, AKL+20a, KLO12, GM11, PPRY18, CGLS17] or *statistically* secure [SCSL11, SvDS+13, WCS15, Ajt10, CLP14] ORAMs. Recall that a computationally secure (or statistically secure, resp.) ORAM guarantees that for any two request sequences of the same length, the access patterns incurred are computationally (or statistically resp.) indistinguishable. Most known computationally secure or statistically secure schemes [GO96, Gol87, SCSL11, SvDS+13, WCS15, BCP16, CS17] suffer from a small failure probability that is *negligible in the ORAM's size henceforth denoted* N (assuming the schemes are parametrized to achieve $\mathsf{poly}\log N$ overhead). If the ORAM/OPRAM's size is large, say, $N \geq \lambda$ for some desired security parameter λ, then the failure probability would also be negligible in the security parameter. Unfortunately, for small choices of N (e.g., $N = \mathsf{poly}\log\lambda$), these schemes actually do not give the commonly believed polylogarithmic performance overhead (assuming that $\mathsf{negl}(\lambda)$ failure probability is desired). But we do care about small ORAMs/OPRAMs with perfect security, since they frequently serve as an essential building block in many application settings, such as in the construction of searchable encryption schemes [DPP18], oblivious algorithms [SCSL11, ACN+19, AKL+20a, PPRY18] including notably, the recent OptORAMa work [AKL+20a] that constructed an optimal ORAM.

The study of *perfectly* secure ORAMs/OPRAMs is partly motivated by the aforementioned mismatch, besides the fact that *perfect security* has long been a topic of interest in the multi-party computation and zero-knowledge proof literature [IKO+11, GIW16], and its theoretical importance widely-accepted. Historically, perfect security is viewed as attractive since 1) the security holds in any computational model even if quantum computers or other forms of computers can be built; and 2) perfectly secure schemes often have clean compositional properties.

State of affairs for perfectly secure ORAMs/OPRAMs. Despite the sustained and lively progress in understanding the asymptotic overhead of computationally and statistically secure ORAMs/OPRAMs, our understanding of perfectly secure ORAMs/OPRAMs has been somewhat stuck. In general, few results are known in the perfect security regime: in 2011, Damgård et al. [DMN11] first showed a perfectly secure ORAM scheme with $O(\log^3 N)$ amortized bandwidth overhead and $O(\log N)$ server storage blowup. Recently, Chan et al. [CNS18] show an improved and simplified construction that removed the $\log N$ server storage blowup; and moreover, they showed how to extend the approach to the parallel setting resulting in a perfectly secure OPRAM scheme with $O(\log^3 N)$ blowup. There is no known super-logarithmic lower bound for perfect security, and thus we do not understand yet whether the requirement of perfect security would inherently incur more overhead than computationally secure ORAMs. Therefore, an exciting and extremely challenging open direction is to understand the exact asymptotic complexity of perfectly secure ORAMs and OPRAMs, that is, to seek a matching upper- and lower-bound. This is a very ambitious goal and in this paper, we aim to take the next natural step forward. Since all prior upper bounds seem stuck at $O(\log^3 N)$, we ask the following natural question:

Does there exist an ORAM/OPRAM with $o(\log^3 N)$ *asymptotic overhead?*

To achieve perfect security, the prior perfect ORAM/OPRAM constructions of Damgård et al. [DMN11] and Chan et al. [CNS18] pay a price: their stated $O(\log^3 N)$ overhead is in an *expected* sense, and the ORAM scheme can occasionally run for a long time if certain unlucky events happen. More specifically, the smaller the choice of N, the more likely that the ORAM can run much longer than the expectation. In other words, such ORAM schemes are Las Vegas algorithm. The very recent work of Raskin et al. [RS19] (Asiacrypt'19) was the first to explicitly discuss this issue, and they ask how to construct perfectly secure ORAMs with *deterministic* (i.e., worst-case) performance

bounds. The were the first to show a perfectly secure ORAM with $O(\sqrt{N}\frac{\log N}{\log\log N})$ *worst-case* overhead (assuming $O(1)$ client-side storage)[2]. While conceptually interesting, in comparison with the $O(\log^3 N)$ scheme by Damgård et al. [DMN11] and Chan et al. [CNS18], the price to pay for the deterministic performance bounds seems somewhat unreasonably high. We therefore ask another natural question:

Does there exist perfectly secure ORAMs/OPRAMs with worst-case polylogarithmic overhead?

1.1 Our Results and Contributions

We answer both of the above questions affirmatively. First, we show a novel perfect ORAM/OPRAM scheme whose overhead is upper bounded by $O(\log^3 N/\log\log N)$ with high probability as stated in the following theorem.

Theorem 1.1 (Informal: perfect OPRAM with high-probability performance bounds)**.** *There exists a perfectly secure OPRAM scheme that consumes only $O(1)$ blocks of client private cache and $O(N)$ blocks of server-space; moreover the scheme achieves $O(\log^3 N/\log\log N)$ simulation overhead with $1-\exp(-\Omega(\log^2 N))$ probability (note that here the failure probability pertains to the performance bound and security and correctness can never be violated). Moreover, the above stated simulation overhead also holds in expectation.*

Second, we show a perfectly secure OPRAM scheme with $O(\log^4 N/\log\log N)$ *worst-case* overhead. Not only is our asymptotic overhead significantly better than the prior work of Raskin et al. [RS19], our scheme also works in the parallel setting, whereas the scheme by Raskin et al. [RS19] is inherently sequential. To the best of our knowledge, our scheme is the first perfectly secure OPRAM scheme with *any non-trivial worst-case* performance overhead. We state our second result in the following theorem.

Theorem 1.2 (Informal: perfect OPRAM with deterministic performance bounds)**.** *There exists a perfectly secure OPRAM scheme that achieves $O(\log^4 N/\log\log N)$ simulation overhead (with probability 1). Further, the scheme consumes only $O(1)$ blocks of client private cache and $O(N)$ blocks of server-space.*

1.2 Technical Highlight

We briefly describe the novel techniques needed to achieve the $O(\log^3 N/\log\log N)$ expected-overhead result, and our result with worst-case overhead can be attained by changing one of the underlying building blocks to an algorithm with deterministic running time.

To improve the overhead of perfectly secure ORAMs/OPRAMs to $O(\log^3 N/\log\log N)$, our techniques are inspired by the rebalancing trick of Kushilevitz et al. [KLO12] (SODA'12), and yet departs significantly from Kushilevitz et al. [KLO12]. We observe that existing perfect ORAM/OPRAM constructions suffer from an imbalance of costs in the "offline maintain phase" and the "online fetch phase"; specifically, in Chan et al. [CNS18], the offline maintain phase costs $O(\log^3 N)$ per request whereas the online fetch phase costs only $O(\log^2 N)$. A natural idea is to modify the scheme and rebalance the costs of the offline maintain phase and the online fetch phase, such that both phases would cost only $O(\log^3 N/\log\log N)$. Unfortunately, existing techniques such as Kushilevitz et al. [KLO12] completely fail for rebalancing perfect ORAMs/OPRAMs — we describe the technical reasons in detail in Appendix A.

[2]Their overhead can be improved to $O(\sqrt{N})$ if we allowed a linear amount of client-side storage.

We devise a combination of novel techniques and design a new ORAM/OPRAM scheme whose offline maintenance phase and online fetch phase cost asymptotically the same, that is, $O(\log^3 N / \log\log N)$. To achieve this, we rely on a combination of several novel techniques.

Our starting point is the recent perfectly secure ORAM construction by Chan et al. [CNS18] in which the maintain phase costs $O(\log^3 N)$ and the fetch phase costs only $O(\log^2 N)$. Specifically, their construction consists of $D = O(\log N)$ number of ORAMs such that except for the last ORAM which stores the actual data blocks, every other ORAM serves as a (recursive) index structure into the next ORAM — for this reason, these D ORAMs are also called D recursion depths; and all of the recursion depths jointly realize an implicit logical index structure that is in fact isomorphic to a binary tree (which has a branching factor of 2).

First, we show how to use a *fat-block* trick to increase the branching factor and hence reduce the number of recursion depths by a $\log\log N$ factor. In Chan et al. [CNS18]'s construction, the implicit index structure's branching factor is 2, since each block can store the pointers (later called position labels in our construction) for two blocks in the next recursion depth. A fat-block is defined as a bundle of logarithmically many normal blocks and hence each fat-block can store logarithmically many pointers. In this way, the logical index structure implemented by the recursion has a branching factor of $\log N$ and thus its depth is reduced by a $\log\log N$ factor. The price, however, is that the fetch phase now costs a logarithmic factor more per recursion depth (since obliviously accessing a fat-block is a logarithmic factor more costly than accessing a normal block).

The primary challenge is how to realize the maintain phase such that the amortized *per-depth* maintain-phase cost preserves the same asymptotics as Chan et al. [CNS18], despite the fat-block now being logarithmically fatter than normal blocks. To accomplish this we rely on the following two key insights:

1. *Exploit residual randomness.* First, we rely on an elegant observation first made in the PanORAMa work [PPRY18] in the context of computationally secure ORAMs. Here we make the same observation for perfectly secure ORAMs. At the core of Chan et al. [CNS18]'s ORAM construction is a data structure called an oblivious "one-time-memory" (OTM). When an OTM is initialized, all elements in it are randomly permuted (and the randomness concealed from the adversary) — note that in our setting, each element is a fat-block. The critical observation is that after accessing a subset of the elements in this OTM data structure, the remaining unvisited elements still appear in a random order. By exploiting such residual randomness, when we would like to build a new OTM consisting of the remaining unvisited elements, we can avoid performing expensive oblivious sorting (which would take time $O(n \log n)$ to sort n elements) and instead rely on linear-time operations.

2. *Exploit sparsity.* In Chan et al. [CNS18]'s construction, the D ORAMs at all recursion depths must perform a "coordinated shuffle" operation during the maintain phase. An important step in this coordinated shuffle is for each recursion depth to inform the parent depth the locations of its fat-blocks after the reshuffle. In Chan et al. [CNS18], two adjacent recursion depths perform such "communication" through oblivious sorting, and thus incurring $O(n \log n)$ cost per-depth to rebuild a data structure of size n.

   Our key observation is that the fat-blocks contained in each OTM data structure in each recursion depth are sparsely populated. In fact, most entries in the fat-blocks are empty and only a $1/\log N$ fraction of them are populated. Thus, at this point, we employ oblivious tight compaction to compress away the wasted space — note that the recent work OptORAMa [AKL+20a] showed how to achieve such compaction in linear time. After this compression, the OTM becomes logarithmically smaller and at this point, we can apply oblivious sorting.

New building block: a perfectly oblivious parallel Intersperse procedure. Last but not the least, to extend the above techniques to the parallel setting, we devise a novel, perfectly oblivious algorithmic building block called **Intersperse**. **Intersperse** was first proposed in the very recent OptORAMa [AKL+20a] work. Given two input arrays of equal length each of which has been randomly permuted, **Intersperse** merges them into a single randomly permuted array without leaking any information through its access patterns. The **Intersperse** primitive proposed in OptORAMa [AKL+20a] is inherently sequential, and takes $O(n)$ work and parallel runtime to merge two input arrays of length n. Our new scheme achieves $O(n)$ work and only logarithmic parallel runtime; therefore, in a parallel setting, our speedup w.r.t. the construction in OptORAMa is exponential. To achieve this we devise new algorithmic techniques. This new parallel **Intersperse** building block might be of independent interest and useful in the construction of other parallel oblivious algorithms.

Non-goals. Our paper focuses on the *theoretical* understanding of the asymptotic complexity of perfectly secure ORAMs/OPRAMs. We do not discuss concrete performance in our paper, but we note that some earlier works have made perfectly secure ORAMs concretely efficient in 3-server settings [CKN+18].

1.3 Additional Related Work

Oblivious RAM (ORAM) was first proposed by Goldreich and Ostrovsky in a seminal work [GO96, Gol87]. The parallel counterpart, called Oblivious Parallel RAM (OPRAM), was first proposed by Boyle, Chung, and Pass [BCP16]. As Boyle et al. argue, OPRAM is important due to the parallelism that is prevalent in modern computing architectures such as multi-core processors and large-scale compute clusters. As mentioned, in this paper, we focus on ORAM/OPRAM constructions that satisfy perfect security. Perfectly secure ORAMs, first studied by Damgård et al. [DMN11], 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.

Importance of perfect security. The recent work by Chan et al. [CNS18] explains the importance of studying perfect security: first, when computationally or statistically secure ORAMs are applied to problems of small size (relative to the desired security parameter), we often need security failures to happen with probability that is not just negligible, but sub-exponentially small in the problem's size. Thus for problems of small size, perfectly secure ORAMs can perform asymptotically better than known computationally or statistically secure ORAMs. Exactly for this reason, existing works on searchable encryption [DPP18] and oblivious algorithms [AKL+20a, SCSL11] adopt perfectly secure ORAMs as a building block for solving small-sized sub-problems that are necessary for solving the bigger problem. Finally, as Chan et al. [CNS18] point out, perfectly secure ORAM can also have theoretical applications in perfectly secure multi-party computation (MPC) and other contexts.

Prior results on perfectly secure ORAMs and OPRAMs. Prior works have considered two types of perfect ORAM/OPRAM constructions:

1. *Constructions whose performance bounds hold in expectation [DMN11, CNS18]*: typically these constructions are Las Vegas algorithms that might occasionally runs longer than the desired performance bounds. Specifically, the original perfect ORAM construction by Damgård et al. [DMN11] achieves $O(\log^3 N)$ *expected* simulation overhead and $O(\log N)$ server space blowup. Subsequently, Chan et al. [CNS18] improved Damgård et al. [DMN11]'s result and

achieved $O(\log^3 N)$ *expected* performance overhead and only $O(1)$ server space blowup. Furthermore, Chan et al. [CNS18] also extended their construction to the parallel setting resulting in an OPRAM scheme with the same asymptotics. For both Damgård et al. [DMN11] and Chan et al. [CNS18], the stated performance bounds hold not just in expectation but in fact, with $1 - \exp(-\Omega(\log^2 N))$ probability. However, if N is small, say, polylogarithmic in some security parameter, this failure probability can be non-negligible.

2. *Constructions with deterministic performance bounds.* It would obviously be nice to have perfect ORAM/OPRAM constructions whose performance bounds hold not just in expectation or with high probability, but with probability 1. With this goal in mind, Raskin et al. [RS19][3] show a perfect ORAM construction achieving a worst-case simulation overhead of $O(\sqrt{N}\frac{\log N}{\log \log N})$.

As mentioned earlier, our new techniques enable asymptotical improvements for both of these above categories.

2 Technical Overview

We start with an informal and intuitive exposition of our main technical ideas before formalizing the definitions, constructions, and proofs in subsequent sections. For simplicity, in the roadmap we first focus on describing the sequential ORAM version. We briefly comment on the technicalities that lie in the way of parallelizing the scheme in Section 2.4, deferring the full details of the parallel construction to the formal technical sections later.

2.1 Background on Perfect ORAM

In a recent work, Chan et al. [CNS18] propose a perfectly secure ORAM with $O(\log^3 N)$ simulation overhead. At a high level, their scheme is inspired by the hierarchical ORAM paradigm by Goldreich and Ostrovsky [GO96,Gol87], but relies on a non-blackbox "recursive position map" trick to remove the pseudo-random function (PRF) in Goldreich and Ostrovsky's construction [GO96, Gol87].

2.1.1 Position-based Hierarchical ORAM

First, imagine that the client can store per-block metadata and we will later remove this strong assumption through a non-blackbox recursion technique. Specifically, imagine that the client remembers where exactly each block is residing on the server. In this case, we can construct a perfectly secure ORAM as follows — henceforth this building block is called "position-based ORAM" since we assume that the correct position label for every requested block is known to the client.

Hierarchical levels. The server-side data structure is organized into $\log N + 1$ levels numbered $0, 1, \ldots, \log N$ where level i is

- either *empty*, in which case it stores no blocks;
- or *full*, in which case the level stores 2^i real blocks plus 2^i dummy blocks in a randomly permuted fashion (we also say that the level has *capacity* 2^i).

Each block, whose logical addresses range from 0 to $N - 1$, resides in exactly one of the levels at a random position within the level.

[3] In different settings presented by Raskin et al., we focus on the setting of $O(1)$ client storage [RS19, Fig. 1].

Fetch phase. Every time a block with address addr is requested, the client looks up the block's position. Suppose that the block resides in the i-th position of level ℓ. The client now visits for one block per full level from the server — note that the levels are visited in a fixed order from 0 to $\log N$:

- for level ℓ (i.e., where the desired block resides), the client reads precisely the i-th position to fetch the real block; it then marks the visited position as dummy;
- for every other level $\ell' \neq \ell$, the client reads a random unvisited dummy block (and marks the corresponding block on the server as dummy for obliviousness).

Maintain phase. Every time a block B has been fetched by the client, it updates the block B's contents if this is a write request. Now, imagine that levels $0, 1, \ldots, \ell^*$ are all full and either level $\ell^* + 1$ is empty or $\ell^* = \log N$. The client will now merge the newly fetched (and possibly updated) block B and levels $0, 1, \ldots, \ell^*$ into the "target" level $\ell_{\text{tgt}} := \min(\ell^* + 1, \log N)$ — this procedure is often called "rebuilding" the level $\ell_{\text{tgt}} := \min(\ell^* + 1, \log N)$. At the end of the rebuild, it marks level ℓ_{tgt} as full and every smaller level as empty.

To merge consecutively full levels into the next empty level (or the largest level), the goal is to implement the following ideal functionality *obliviously*:

1. extract all *real* blocks to be merged and place them in an array called A;
2. pad A with dummy blocks to a total length of $2 \cdot 2^{\ell_{\text{tgt}}}$ and randomly permute the resulting array.

Chan et al. [CNS18] shows how to achieve the above obliviously — even though the client has only $O(1)$ blocks of private cache — through oblivious sorting (which can be instantiated using the AKS sorting network [AKS83]). The cost of rebuilding a level of capacity n is dominated by the oblivious sorting on $O(n)$ blocks, which hast a cost of $O(n \log n)$.

Note that the above construction guarantees that whenever a real block is accessed, it is moved into a smaller level. Thus, in every level, each real or dummy block is accessed at most once before the level is rebuilt; and this is important for obliviousness. For this reason, later in our technical sections, we name each level in this hierarchy an oblivious "one-time memory". Note also that *the number of dummies in a level must match the total number of accesses the level can support before it is rebuilt again.*

Additional details about dummy positions. The above description implicitly assumed that for a level the desired block does not reside in, the client is informed of the position of a random unvisited dummy block. If the client does not store this information locally, it can construct a (per-level) metadata array M on the server every time a level is rebuilt. When a block is being requested, the client can sequentially scan the metadata array at *every* level (including the level where the desired block resides) to discover the location of the next unvisited dummy (residing at a random unvisited location in the level).

As Chan et al. [CNS18] show, such a dummy metadata array can be constructed with $O(n \log n)$ overhead using oblivious sorting too, at the time a level of capacity n is rebuilt.

Overhead. Summarizing, in the position-based ORAM, after every 2^ℓ requests, the level ℓ will be rebuilt, paying $O(2^\ell \cdot \log(2^\ell))$ cost. Amortizing the total cost over the sequence of requests, it is not difficult to see that the average cost per request is $O(\log^2 N)$.

2.1.2 Recursive Position Map

So far we have assumed that whenever the client wants to fetch a block, it can *magically* find out the block's position on the server. To remove this assumption, Chan et al. [CNS18] propose to recursively store the blocks' position labels in smaller ORAMs until the ORAM's size becomes constant, resulting in $O(\log N)$ ORAMs henceforth denoted $\mathsf{ORAM}_0, \mathsf{ORAM}_1, \ldots, \mathsf{ORAM}_D$ respectively, where ORAM_i stores the position labels of all blocks in ORAM_{i+1} for $i \in \{0, 1, \ldots, D\}$. We often call ORAM_D the "data ORAM" and every other ORAM a "metadata ORAM"; we also refer to the index i as the *depth* of ORAM_i. Now, suppose that each block can store $\Omega(\log N)$ bits of information, such that we can pack the position labels of at least 2 blocks into a single block. In this case, each ORAM_i is twice smaller in capacity than ORAM_{i+1} and thus ORAM_0 would be of $O(1)$ size — thus operations to ORAM_0 can be supported trivially by scanning through the whole ORAM_0 consuming only constant cost.

As Chan et al. [CNS18] show, in the hierarchical ORAM context such a recursion idea does not work in a straightforward blackbox manner, but needs a special "coordinated rebuild" technique which we now explain. Henceforth, suppose that each block's logical address addr is $\log N$ bits long, and we use the notation $\mathsf{addr}^{\langle d \rangle}$ to denote the address addr, written in binary format, truncated to the first d bits.

- *Fetch phase (straightforward):* To fetch a block at some logical address addr, the client looks up logical address $\mathsf{addr}^{\langle d \rangle}$ in each ORAM_d for $d = 0, 1, \ldots D$ sequentially. Since the block at logical address $\mathsf{addr}^{\langle d \rangle}$ in ORAM_d stores the position labels for the two blocks at logical addresses $\mathsf{addr}^{\langle d \rangle}||0$ $\mathsf{addr}^{\langle d \rangle}||1$ in ORAM_{d+1}, the client is always able to find out the position of the block in the next recursion depth before it performs a lookup there.

- *Maintain phase (coordinated rebuild):* The maintain phase needs special treatment such that the rebuilds at all recursion depths are coordinated. Specifically, whenever the data ORAM_D is rebuilding the level ℓ, each other recursion depth ORAM_d would be rebuilding level $\min(\ell, d)$ in a coordinated fashion — note that each ORAM_d has only d levels.

  The main goal of the coordination is for each ORAM_d to pass the blocks' updated position labels back to the parent depth ORAM_{d-1}. More specifically, recall that when ORAM_d rebuilds a level ℓ, all real blocks in the level would now be placed at a new random position. When these new positions have been decided, ORAM_d must inform the corresponding metadata blocks in ORAM_{d-1} the new position labels. The coordinated rebuild is possible due to the following invariant which is not hard to observe (recall that $\mathsf{addr}^{\langle d \rangle}$ is the block that stores the position labels for the block $\mathsf{addr}^{\langle d+1 \rangle}$ in ORAM_{d+1}):

  > For every addr, the block at address $\mathsf{addr}^{\langle d \rangle}$ in ORAM_d is always stored at a smaller or equal level relative to the block at address $\mathsf{addr}^{\langle d+1 \rangle}$ in ORAM_{d+1}.

  Chan et al. [CNS18] show how to rely on oblivious sorting to accomplish this coordinated rebuild, paying $O(n \log n)$ to pass the new position labels of level-ℓ in ORAM_d to the parent ORAM_{d-1} where $n = 2^\ell$ is the level's capacity.

2.1.3 Analysis

It is not hard to see that the entire fetch phase consumes $O(\log^2 N)$ overhead where one $\log N$ factor comes from the number of levels within each recursion depth, and another comes from the number of recursion depths. The maintain phase, on the other hand, consumes $O(\log^3 N)$ *amortized*

cost where one logarithmic factor arises from the number of depths, one arises from the number of levels within each depth, and the last one stems from the additional logarithmic factor in oblivious sorting.

To asymptotically improve the overhead, one promising idea is to somehow balance the fetch and maintain phases. This idea has been explored in computationally secure ORAMs first by Kushilevitz et al. [KLO12] and later improved in subsequent works [CGLS17]. Unfortunately as we explain in Appendix A, Kushilevitz et al.'s rebalancing trick is not compatible with known perfectly secure ORAMs. Thus we need fundamentally new techniques for realizing such a rebalancing idea.

2.2 Building Blocks

Before we introduce our new algorithms, we describe two important oblivious algorithm building blocks that were discovered in very recent works [AKL+20a, Pes18].

Tight compaction. Tight compaction is the following task: given an input array containing m balls where each ball is tagged with a bit indicating whether it is real or dummy, produce an output array containing also m balls such that all real balls in the input appear in the front and all dummies appear at the end.

In a very recent work called OptORAMa [AKL+20a], the authors show how to accomplish tight compaction obliviously in $O(1)$ overhead. Their algorithm can be expressed as a linear-sized circuit (of constant fan-in and fan-out), consisting only of boolean gates and swap gates, where a boolean gate can perform boolean computations on two input bits; and a swap gate takes in a bit and two balls, and decides to either swap or not swap the two balls.

Intersperse. The same work OptORAMa [AKL+20a] described another linear-time, randomized oblivious algorithm called "intersperse", which can be used to accomplish the following task: given two randomly shuffled input arrays $\mathbf{I}$ and $\mathbf{I}'$ (where the permutations used in the shuffles are hidden from the adversary), create an output array of length $|\mathbf{I}| + |\mathbf{I}'|$ that contains all elements from the two input arrays, and moreover, all elements in the output array are randomly shuffled in the view of the adversary.

2.3 A New Rebalancing Trick for Perfectly Secure ORAMs

We propose new techniques for instantiating such a rebalancing trick. Our idea is to introduce a notion called a fat-block. A fat-block is a bundle of $\chi := \log N$ normal blocks; thus to access a fat-block requires paying $\chi = \log N$ cost.

Imagine that in each metadata ORAM, the atomic unit of storage is a fat-block (rather than a normal block). Since each fat-block can pack $\chi = \log N$ position labels, the depth of the recursion is now $\log_\chi N = \log N / \log\log N$, i.e., a $\log\log N$ factor smaller than before (see Section 2.1.2). More concretely, a metadata ORAM ORAM_d at depth d stores a total of χ^d metadata fat-blocks — for the time being we assume that N is a power of χ for simplicity, and let $D := \log_\chi N + 1$ be the number of recursion depths such that the total storage is still $O(N)$ blocks (but our idea can easily be generalized to the case when N is not a power of χ). Within each ORAM_d, as before, we have a total of $d \log \chi + 1$ levels where each level ℓ can store 2^ℓ fat-blocks.

It is not hard to see that the fetch phase would now incur $O(\log^3 N / \log\log N)$ cost across all recursion depths — in comparison with before, the extra $\log N$ factor arises from the cost of reading a fat-block, and the $\log\log N$ factor saving comes from the $\log\log N$ saving in depth.

Our hope is that now with the smaller recursion depth, we can accomplish the maintain phase in amortized $O(\log^3 N / \log\log N)$ cost. Recall that each level ℓ in a metadata ORAM_d now contains

2^ℓ fat-blocks. The crux is to be able to rebuild a level containing 2^ℓ fat-blocks in cost that is linear in the level's total size, that is, $2^\ell \cdot \chi$. Note that if we naïvely used oblivious sorting on fat-blocks (like in Section 2.1.1) to accomplish this, the cost would have been $2^\ell \cdot \chi \cdot \log(2^\ell)$.

To resolve this challenge, the following two insights are critical:

- *Sparsity:* First, observe that each level in a metadata ORAM is *sparsely populated*: although the entire level, say, level ℓ, has the capacity to store $2^\ell \cdot \chi$ position labels, the level is rebuilt after every 2^ℓ requests. Thus in fact only 2^ℓ of these position label entries are populated.

- *Residual randomness:* The second important observation is that the unvisited fat-blocks contained in any level appear in a random order where the randomness of the permutation is hidden from the adversary — note that a similar observation was first made in the recent PanORAMa work [PPRY18] by Patel et al.

  More specifically, suppose that to start with, a level contains n fat-blocks including some reals and some dummies, and all of these n fat-blocks have been randomly permuted (where the randomness of the permutation is hidden from the adversary). As the client visits fat-blocks in a level, the adversary learns which blocks are visited. Now, among all the unvisited blocks, there are both real and dummy blocks and all these blocks are equally likely to appear in any order w.r.t. the adversary's view.

We now explain how to rely on the above insights to rebuild a level containing $n = 2^\ell$ fat-blocks in $O(n \cdot \chi)$ overhead — note that at most half of these fat-blocks are real, and the remaining are dummy. From Section 2.1.2, we learned that to rebuild a level containing n fat-blocks, it suffices to realize the following functionality obliviously:

a) *Merge.* The first step of the rebuild is to merge consecutively full levels plus the newly accessed fat-block into the next empty level (or the largest level). After this merge step, this new level is marked full and every smaller level is marked empty.

b) *Permute.* After the above merge step, the resulting array containing n fat-blocks must be randomly permuted (and their positions after the permutation will then be passed to the parent depth).

c) *Update.* After the permutation step, each real fat-block in the level whose logical address is addr must receive up to χ updated positions from the child recursion depth, i.e., the fat-block at logical address addr wants to learn where the fat-blocks at logical addresses $\mathsf{addr}||0$, $\mathsf{addr}||1$, ..., $\mathsf{addr}||(\chi - 1)$ newly reside in the child depth.

d) *Create dummy metadata.* Finally, create a dummy metadata array to accompany this level: the dummy metadata array containing n entries where each entry is $O(\log N)$ bits (note that an entry is a normal block, not a fat-block). This array should store the positions of all *dummy* fat-blocks contained in the level in a randomly permuted order, and padded with $\perp$ to a length of n.

Realizing "merge + permute". We first explain how to accomplish the "merge + permute" steps. For simplicity we focus on explaining the case where consecutive full levels are merged into the next empty level (since it would be fine if the merging into the largest level *alone* is done naïvely using oblivious sort on all fat-blocks). Here it is important to rely on the residual randomness property mentioned earlier. Suppose the levels to be merged contain $1, 2, 4, \ldots, n/2$ fat-blocks respectively; besides these, the most recently accessed fat-block also wants to be merged. Recall that in all of these levels to be merged, the unvisited blocks appear in a random order w.r.t.

the adversary's view. Thus, we can simply do $O(\log n)$ cascading merges using **Intersperse**, every time merging two arrays each containing 2^i fat-blocks into an array containing 2^{i+1} fat-blocks.

Realizing "update". At this moment, let us not view the level as an array of n fat-blocks any more, but as an array of $O(n \cdot \chi)$ position entries. For realizing the "update" step in $O(n \cdot \chi)$ overhead, the key insight is to exploit the sparsity.

Recall that the problem we need to solve boils down to the following. Imagine there is a destination array D consisting $O(n \cdot \chi)$ position entries among which $O(n)$ entries are populated (i.e., real), and all remaining entries are dummy. Additionally, there is a source array S consisting of $O(n)$ entries. In both the source S and the destination D, each real entry is of the form (k, v) where k denotes a key and v denotes a payload value; further, in the destination D, every real entry must have a distinct key. Now, we would like to route each real entry $(k, v) \in S$ to the corresponding entry with the same key in the destination array D.

Exploiting the sparsity in the problem definition, we devise the following algorithm where an important building block is linear-time *oblivious tight compaction* (see Section 2.2).

First, we rely on oblivious tight compaction to compact the destination array D, resulting in a compacted array $\widetilde{D}$ consisting of only $O(n)$ entries. Moreover, recall that oblivious tight compaction can be viewed as a *circuit* consisting of boolean gates and swap gates. When we compact the destination array D, each swap gate remembers the routing decision since later it will be useful to run this circuit in the reverse direction. After the compaction, we can now afford to pay the cost of oblivious sorting. Through $O(1)$ oblivious sort operations, each entry in the source S can route itself to each entry in the compacted destination $\widetilde{D}$ — this can be accomplished through a standard technique called oblivious routing [CS17, BCP16], which has a cost of $O(n \log n)$. Now, by running the aforementioned tight compaction circuit in the reverse direction, we can route each element of the compacted destination $\widetilde{D}$ back into the original destination array D.

It is not difficult to see that the above steps require only $O(n \cdot (\chi + \log n))$ cost.

Obliviously create dummy metadata array. Finally, obliviously creating the dummy metadata array is easy: this can be accomplished by writing down $O(\log N)$ bits of metadata per fat-block, and then by performing a combination of oblivious random permutation and oblivious sort on the resulting metadata array.

2.4 Parallelizing the Scheme

So far, for simplicity we have focused on the sequential case. To obtain our OPRAM result, we need to make the above scheme parallel. To this end, we will rely on the OPRAM techniques by Chan et al. [CNS18]. However, we are faced with the new challenge of how to make the **Intersperse** algorithm parallel. Recall that given two randomly shuffled arrays $\mathbf{I}_0$ and $\mathbf{I}_1$, **Intersperse** produces a randomly shuffled output array combining the two input arrays. The linear-time **Intersperse** algorithm described by Asharov et al. [AKL$^+$20a] consists of two steps: 1) sample a random auxiliary array consists of a $|\mathbf{I}_0|$ number of 0s and $|\mathbf{I}_1|$ number of 1s; and 2) run tight compaction on the auxiliary array, and then apply the reverse routing to the elements of $\mathbf{I}_0||\mathbf{I}_1$. While the second step can be parallelized, the first step is inherently sequential in Asharov et al. [AKL$^+$20a]. Our task therefore is to parallelize the first step, and importantly, we want that the output array to be *perfectly* randomly permuted — however we allow the algorithm to be Las Vegas and this will get us the $O(\log^3 N / \log\log N)$ result where the performance bound holds with high probability. We defer the algorithmic details on how to achieve this to Section 4. Note that since the runtime of the Las Vegas algorithm leaks information about the lengths of the two input arrays $\mathbf{I}_0$ and $\mathbf{I}_1$, our **Intersperse** abstraction is in fact slightly weaker than that of Asharov et al. [AKL$^+$20a] — we

leak the lengths of both input arrays and not just the sum of the two lengths. However, it turns out that the weaker version is enough to get our result.

2.5 Roadmap of Subsequent Formal Sections

In the subsequent technical sections, we formalize the blueprint described in this section. Our formal description is modularized which will facilitate formal analysis and proofs. Moreover, in our formal section we will directly present the OPRAM result (since the sequential ORAM is a special case of the more general OPRAM result).

3 Preliminaries

3.1 PRAMs and Oblivious Simulation

PRAMs and OPRAMs. We consider how to obliviously simulate parallel algorithms in the standard Parallel Random-Access Machine (PRAM) model. Both the original (insecure) algorithm and the obliviously simulated algorithm run on a PRAM machine with m CPUs; and it is required that the oblivious counterpart must always give the same output distribution as the original PRAM on any input. Let $[N]$ be the address space of the shared memory in the PRAM. We assume without loss of generality that $N \geq m$ throughout this paper[4].

More concretely, the original PRAM algorithm will generate a batch of m memory requests in each parallel step, where each memory request wants to either read a logical address or write a logical address. In essence we need to devise an oblivious method to serve every batch of m memory requests and always ensure correctness. *Obliviousness requires that every request sequence of the same length results in identical access pattern distribution.* Throughout the paper, the memory access pattern of a PRAM means the ordered vector of the physical memory locations accessed by each CPU in all steps (and within each step, the physical locations are ordered by the identifiers of the CPUs making the access).

For write conflict resolution, we allow the original (insecure) PRAM to support 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 parallel step. For the oblivious-simulation PRAM, we assume a "concurrent read, exclusive write" PRAM (CREW). In other words, our OPRAM algorithm must ensure that there are no concurrent writes at any time. Note that allowing the original PRAM to be CRCW but restricting the compile oblivious PRAM to be CREW makes it a stronger result.

In our paper, we will assume that each CPU can perform word-level operations including addition, subtraction, and bit-wise boolean operations in unit time. Further, we make the following assumptions related to sampling the randomness required by the algorithm: 1) let w denote the number of bits in a word, we shall assume that each CPU can sample a uniform random number from $[m]$ for any $m \leq 2^w$ in unit time; 2) we assume that for any $a < b \leq 2^w$, each CPU can sample a Bernoulli random variable with probability a/b in unit time (which follows by the first assumption). Specifically, the first assumption above is needed by the Alonso and Schott [AS96] oblivious random permutation and the second assumption is needed by our parallel **Intersperse** algorithm in Section 4.

[4]If $N < m$, the oblivious simulation can be achieved by assigning at most one address to each CPU and then performing oblivious routing [BCP16], which takes only $O(\log m)$ overhead.

Appendix B provides more detailed definitions on PRAMs and OPRAMs.

Metrics. We will use the standard notion of *simulation overhead* to characterize an OPRAM's performance. 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, then we say that the simulation overhead is γ.

More generally, suppose that an ample (i.e., unbounded) number of CPUs are available: in this case if algorithm can be completed in T parallel steps consuming $m_1, m_2, \ldots, m_T$ CPUs in each step respectively, then we say that the algorithm can be completed in T *depth* and $W := \sum_{t \in [T]} m_t$ *total work*. Therefore, for an OPRAM, if each parallel step of the original PRAM (i.e., a batch m memory requests) can be completed in W total work and $T = O(W/m)$ depth then the OPRAM has simulation overhead W/m.

Oblivious simulation of a non-reactive functionality. For defining the security of intermediate building blocks, we now define what it means to *obliviously realize* a non-reactive functionality. Let $\mathcal{F} : \{0,1\}^* \to \{0,1\}^*$ be a possibly randomized functionality. We say that $M_{\mathcal{F}}$ is a perfect oblivious simulation (or oblivious simulation for short) of $\mathcal{F}$ with leakage $\mathcal{L}$, iff there exists a simulator Sim, such that for every input $x \in \{0,1\}^*$, the following real-world and ideal-world distributions are identical:

- *Real world*: execute $M_{\mathcal{F}}(x)$ and let y be the output and Addr be the memory access patterns; output (y, Addr);
- *Ideal world*: output $(\mathcal{F}(x), \mathsf{Sim}(\mathcal{L}(x)))$.

For simplicity, if the leakage function $\mathcal{L}(x) = |x|$, we often say that $M_{\mathcal{F}}$ is a perfect oblivious simulation of $\mathcal{F}$ (omitting the leakage function) for short.

Modeling input assumptions. Some of our building blocks provide perfect obliviousness only if the input array is randomly shuffled and the corresponding randomness concealed. More formally, suppose that a machine $M(A, x)$ and a functionality $\mathcal{F}(A, x)$ both take in an array $A \in D^n$ where $D \in \{0,1\}^\ell$ as input and possibly an additional input $x \in \{0,1\}^*$. Formally, we say that "the machine M is a perfect oblivious simulation of the functionality $\mathcal{F}$ with leakage $\mathcal{L}$ *assuming that the input array A is randomly shuffled*", iff for every $A \in D^n$ and every $x \in \{0,1\}^*$, the following real-world and ideal-world distributions are identical:

- *Real world*: randomly shuffle the array A and obtain A', execute $M_{\mathcal{F}}(A', x)$ and let y be the output and Addr be the memory access patterns; output (y, Addr);
- *Ideal world*: output $(\mathcal{F}(A, x), \mathsf{Sim}(\ell, \mathcal{L}(A, x)))$.

Note that the above definition considers only a single input array A, but there is a natural generalization for algorithms that take two or more input arrays — in this case we may require that some or all of these input arrays be randomly shuffled to achieve perfect obliviousness.

3.2 Oblivious Algorithm Building Blocks

We describe some algorithmic building blocks. Unless otherwise noted, for algorithms that operate on arrays of n elements, we always assume that a single memory word is wide enough to store the index of each element within the array, i.e., $w \geq \log n$ where w is the bit-width of each PRAM word. We typically use the following notation: let B denote the bit-width of each element, and let $\beta := \lceil B/w \rceil$ denote the number of memory words it takes to store each element.

3.2.1 Oblivious Sort

Oblivious sorting can be accomplished through a sorting network such as the famous construction by Ajtai, Komlós, and Szemerédi [AKS83]. We restate this result in the context of PRAM algorithms:

Theorem 3.1 (Oblivious sorting [AKS83]). *There exists a deterministic, oblivious algorithm that sorts an array of n elements consuming $O(\beta \cdot n \log n)$ total work and $O(\log n)$ depth where $\beta \geq 1$ denotes the number of memory words it takes to represent each element.*

3.2.2 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 it is a perfect oblivious simulation of the functionality $\mathcal{F}_{\text{perm}}$.

Alonso and Schott [AS96] construct a parallel random permutation algorithm that takes $O(n \log^2 n)$ total work and $\log^2 n$ depth to randomly permute n elements. Although achieving obliviousness was not a goal of their paper, it turns out that their algorithm is also perfectly oblivious, giving rise to the following theorem:

Theorem 3.2 (Alonso-Schott ORP). *Suppose that for any integer $m \in [n]$, each CPU of the PRAM can sample an integer uniformly at random from $[m]$ in unit time. Then, there is a perfectly oblivious algorithm that permutes an array of n elements in $O(\beta \cdot n \log n + n \log^2 n)$ total work and $O(\log^2 n)$ depth where $\beta \geq 1$ denotes the number of memory words for representing each element.*

A few recent works [CCS17, AKL+20a] describe another perfectly oblivious random permutation algorithm which is asymptotically more efficient but the algorithm is Las Vegas, i.e., the algorithm satisfies perfect obliviousness and correctness, but with a small probability the algorithm may run longer than the stated bound.[5] Below, we restate this result in the form that we desire in this paper — the specific theorem stated below arises from the improved analysis of Asharov et al. [AKL+20a]; for the performance bounds, we state an *expected* version and a *high-probability* version.

Theorem 3.3 (A Las Vegas ORP algorithm). *Let $\beta \geq 1$ denote the number of memory words it takes to represent each element. There exists a Las Vegas perfectly oblivious random permutation construction that completes in expected $O(\beta \cdot n \log n)$ total work and expected $O(\log n)$ depth. Furthermore, except with $n^{-\Omega(\sqrt{n})}$ probability, the algorithm completes in $O(\beta \cdot n \log n)$ total work and $O(\log n)$ depth.*

Note that the above theorem gives a high-probability performance bound for sufficiently large n. Later in our OPRAM construction, we will adopt ORP for problems of different sizes — we will use Theorem 3.3 for sufficiently large instances and use Theorem 3.2 for small instances.

3.2.3 Oblivious Routing

Oblivious routing [BCP16] is the following primitive where n source CPUs wish to route data to n' destination CPUs based on the key.

[5] Using more depth but only unbiased random bits, Czumaj [Czu15] shows a Las Vegas switching network to achieve the same abstraction.

- *Inputs:* The inputs contain two arrays: 1) a source array $\mathsf{src} := \{(k_i, v_i)\}_{i\in[n]}$ where each element is a (key, value) pair or a dummy element denoted $(\perp, \perp)$; and 2) a destination array $\mathsf{dst} := \{k'_i\}_{i\in[n']}$ containing a list of (possibly dummy) keys.

  We assume that each (non-dummy) key appears no more than C times in the src array where $C = O(1)$ is a known constant; however, each (non-dummy) key can appear any number of times in dst.

- *Outputs:* We would like to output an array $\mathsf{Out} := \{v'_{i,j}\}_{i\in[n'],j\in[C]}$ where $(v'_{i,1}, \ldots, v'_{i,C})$ contains all the values contained in src whose keys match k'_i (padded with $\perp$).

Theorem 3.4 (Oblivious routing [BCP16,CS17,CCS17])**.** *There exists a perfectly oblivious routing algorithm that accomplishes the above task in $O(\log(n + n'))$ depth and $O(\beta \cdot (n + n') \log(n + n'))$ total work where $\beta \geq 1$ denotes the number of words it takes to represent each element.*

3.2.4 Tight Compaction

As mentioned in Section 2.2, tight compaction is the following task: given an input array containing n elements where each element is tagged with a bit indicating whether it is real or dummy, produce an output array containing also n elements such that all real elements in the input appear in the front and all dummies appear at the end. We will use the parallel oblivious tight compaction of Asharov et al. [AKL+20b] running in linear work and logarithmic depth.

Theorem 3.5 (Oblivious tight compaction [AKL+20b])**.** *There exists a deterministic, oblivious tight compaction algorithm that compacts an array of n elements in total work $O(\beta \cdot n)$ and depth $O(\log n)$ where $\beta \geq 1$ denotes the number of words it takes to represent each element. Moreover, all elements are moved in the black-box manner (except for the real or dummy tags).*

4 Parallel Intersperse

4.1 Definition

Inspired by recent works [PPRY18,AKL+20a], we define a building block called **Intersperse** which can be used to mix two input arrays. Here we adopt a definition that differs slightly from Asharov et al. [AKL+20a] — in our definition the algorithm receives the two input arrays separately and the lengths of both input arrays are *publicly known*. In comparison, in Asharov et al. [AKL+20a]'s definition, the **Intersperse** algorithm receives the concatenation of the two input arrays and only the sum of their lengths is public but not each array's individual length. More specifically, **Intersperse** has the following syntax.

- **Input.** Two arrays $(\mathbf{I}_0, \mathbf{I}_1)$ of size n_0 and n_1, where $n_0 + n_1 = n$.
- **Output.** An array $\mathbf{B}$ of size n that contains all elements of $\mathbf{I}_0$ and $\mathbf{I}_1$. Each position in $\mathbf{B}$ will hold an element from either $\mathbf{I}_0$ or $\mathbf{I}_1$, chosen uniformly at random and the choices are concealed from the adversary.

We now define the security notion required for **Intersperse**. We require that when we run **Intersperse** on two input arrays $\mathbf{I}_0$ and $\mathbf{I}_1$ that are both randomly shuffled (based on a secret permutation), the resulting array will be randomly shuffled (based on a secret permutation) too. More formally stated, we require that **Intersperse** is a perfect oblivious simulation of the following $\mathcal{F}_{\text{shuffle}}(\mathbf{I}_0, \mathbf{I}_1)$ functionality with leakage $(|\mathbf{I}_0|, |\mathbf{I}_1|)$, provided that the two input arrays are randomly

shuffled. Henceforth we assume that the bit-width of each element in the input arrays is a publicly known parameter that the scheme is implicitly parametrized with.

$\mathcal{F}_{\text{shuffle}}(\mathbf{I}_0, \mathbf{I}_1)$:

1. Choose a permutation $\pi : [n] \to [n]$ uniformly at random where $n := |\mathbf{I}_0| + |\mathbf{I}_1|$.
2. Let $\mathbf{I}$ be the concatenation of $\mathbf{I}_0$ and $\mathbf{I}_1$.
3. Initialize an array $\mathbf{B}$ of size n. Assign $\mathbf{B}[i] := \mathbf{I}[\pi(i)]$ for every $i = 1, \ldots, n$.
4. **Output:** The array $\mathbf{B}$.

The recent work OptORAMa by Asharov et al. [AKL+20a] showed how to construct an **Intersperse** algorithm in linear time, i.e., $O(n)$; however, their algorithm is inherently sequential. A manuscript by Asharov et al. [AKL+] considered how to devise a parallel version of **Intersperse** in an attempt to make OptORAMa [AKL+20a] parallel; but their parallel **Intersperse** algorithm achieves only statistical security. Below we describe a variant of their algorithm that is perfectly secure.

4.2 A Parallel Intersperse Algorithm

4.2.1 Warmup

Asharov et al. [AKL+20a] used the following method to construct an **Intersperse** algorithm:

1. First, initialize an array Aux of size n that has n_0 zeros and n_1 ones, where the zeros' positions are chosen uniformly at random (and the remaining positions are ones). More formally, the algorithm must obliviously simulate the following $\mathcal{F}_{\text{SampleAux}}(n, n_0)$ functionality with leakage (n, n_0).

   $\mathcal{F}_{\text{SampleAux}}(n, n_0)$ – **Sample Auxiliary Array**

   - **Input:** Two numbers $n, n_0 \in \mathbb{N}$ such that $n_0 \leq n$.
   - **The functionality:** Sample an array Aux of n bits uniformly at random conditioned on having n_0 zeros and $n - n_0$ ones. Output Aux.

2. Next, we route elements 1-to-1 from $\mathbf{I}_0$ to zeros in Aux and 1-to-1 route elements from $\mathbf{I}_1$ to ones in Aux. This can be accomplished by running oblivious tight compaction circuit (Theorem 3.5) to pack all the 0s in Aux in the front. During the process, all swap gates remember their routing decisions. Now, we can run the oblivious tight compaction circuit in reverse and on the input array $\mathbf{I}_0||\mathbf{I}_1$. It is not hard to see that in the outcome, every 0 position in Aux would receive an element from $\mathbf{I}_0$ and every 1 position in Aux would receive an element from $\mathbf{I}_1$.

Asharov et al. [AKL+20a] proved that the above algorithm indeed realizes the **Intersperse** abstraction as defined above.[6] Moreover, their oblivious tight compaction algorithm is parallel in nature (see Section 3.2.4); unfortunately they adopt a highly sequential procedure for generating the Aux array.

Therefore, it suffices to devise a parallel procedure for generating such an Aux array. More formally, we would like to devise an algorithm that obliviously simulates the functionality $\mathcal{F}_{\text{SampleAux}}(n, n_0)$ allowing leakage (n, n_0).

[6] In fact, as mentioned, they prove a slightly stronger version where only the total length of the two input arrays is revealed but not each individual array's length.

4.2.2 A Naive Algorithm

A naïve algorithm is the following: simply write down exactly n_0 number of 0s and $n - n_0$ number of 1s, apply an oblivious random permutation to permute the array, and output the result. If we use Theorem 3.2 to instantiate this naïve algorithm, we obtain the following theorem:

Theorem 4.1 (Naïve parallel algorithm for sampling Aux)**.** *For any $n_0 \le n$, there exists an algorithm that perfectly obliviously simulates $\mathcal{F}_{\text{SampleAux}}(n, n_0)$; moreover, for sampling an* Aux *array of length n, the algorithm completes in $O(n \log^2 n)$ total work and $O(\log^2 n)$ depth.*

This immediately gives rise to the following corollary for **Intersperse** due to the result of Asharov et al. [AKL+20a]:

Corollary 4.2 (Naïve parallel **Intersperse**)**.** *There exists an algorithm that perfectly obliviously simulates $\mathcal{F}_{\text{shuffle}}$ for two randomly shuffled input arrays. Moreover, the algorithm completes in $O(\beta n + n \log^2 n)$ total work and $O(\log^2 n)$ depth where n denotes the sum of the lengths of the two input arrays, and $\beta \ge 1$ denotes the number of memory words required to represent each element.*

4.2.3 A More Efficient Las Vegas Algorithm

Below we describe a more efficient Las Vegas Algorithm 4.3 that obliviously simulates the functionality $\mathcal{F}_{\text{SampleAux}}(n, n_0)$ with leakage (n, n_0).

Algorithm 4.3: PSampleAuxArray – **Sample Auxiliary Array with Low Depth**

- **Input:** Two numbers $n, n_0 \in \mathbb{N}$ such that $n_0 \le n$.
- **The Algorithm:**
  Let $n_1 = n - n_0$. In the following, without loss of generality, we assume $n_0 \le n_1$ (otherwise, simply swap the roles of 0s and 1s and run a symmetric algorithm). Let λ be the largest integer that such that $n \ge \log^6 \lambda$.

  1. *Approximate initialization.* If $n_0 < \log^4 \lambda$, write down an initial array of size n containing all 1s. Else if $n_0 \ge \log^4 \lambda$, write down an initial array where each element is set to 0 with probability $\frac{n_0}{n}$ and set to 1 otherwise. Let X denote the outcome array of this step. Let n_0' be the number of 0s in X and n_1' be the number of 1s.
  2. *Number of bits to flip.* If $n_0' > n_0$, let $b^* = 0$ and if $n_1' > n_1$, let $b^* = 1$. Let $F^* = n_{b^*}' - n_{b^*}$. (i.e., F^* is the number of 0s or 1s in the array X that are needed to be flipped to reach our target of having exactly n_0 number of 0s.) We can obliviously compute b^* and F^* in $O(n)$ total work and $O(\log n)$ depth by summing up the array in a tree-like manner.
  3. *Subsampling by $\frac{1}{\log \lambda}$ factor.* Make a copy of X and call it Y. For each coordinate $i \in [n]$ in Y, sample a random indicator bit that is 1 with probability $\frac{1}{\log \lambda}$ and attach it to the entry.

     Run the oblivious tight compaction (Section 3.2.4) on Y to get all the elements that are tagged with a 1 in the front. During this process, each swap gate in the circuit remembers its routing decision such that later we could perform reverse routing to route a fine-tuned version of Y back into X. If the number of elements tagged with a 1 is more than $\frac{1.5n}{\log \lambda}$, then Retry from Step 1. Let $Y' := Y[1..\frac{1.5n}{\log \lambda}]$.

4. *Fine-tuning.* Obliviously sort the array Y' such that all the b^* bits appear in the front and flip the first F^* bits of the outcome array. If there are less than F^* such b^* bits, Retry from Step 1. Perform an oblivious random permutation algorithm (Theorem 3.3) on Y', and then overwrite the front Y with Y' by $Y[i] := Y'[i]$ for each $i \in [\frac{1.5n}{\log \lambda}]$.
5. *Reverse-routing.* Reverse route the array Y back to the input array, overwriting the corresponding positions in the input. This is performed using the information we recorded in Step 3: each swap gate in the tight compaction circuit remembered its routing decision so we can reverse-route the array Y back to the array X. Output X.

Theorem 4.4 (PSampleAuxArray)**.** *For any parameters $n_0 \le n$, the algorithm* PSampleAuxArray(n, n_0) *is a perfect oblivious simulation of $\mathcal{F}_{\text{SampleAux}}(n, n_0)$ with leakage (n, n_0). Except with probability $e^{-\Omega(n^{1/3})}$, the algorithm completes in $O(n)$ total work and $O(\log n)$ depth. Furthermore, the above stated performance bounds also apply in expectation.*

Proof. We first analyze the performance of PSampleAuxArray, assuming that there is no Retry. With no Retry, the performance of the algorithm is dominated by tight compaction on (bit-)arrays of size $O(n)$, oblivious sort and oblivious random permutation on arrays of size $\Theta(\frac{n}{\log \lambda}) = \Omega(n^{\frac{5}{6}})$. By Theorem 3.3, it runs in $O(n + \frac{n \log n}{\log \lambda})$ total work and $O(\log n)$ depth, either in expectation or except with probability $n^{-\Theta(n^{5/12})} \le e^{-\Omega(n^{1/3})}$.

To show the performance with Retry, we bound the probability of Retry, which readily adds to the final failure probability for the high probability statement. With the high probability statement, observing the number of times of Retry is a geometric distribution, the expected number of times of Retry is $O(1)$, which gives the desired expected performance.

To give an upper bound of the probability of Retry, we consider the following bad events. Let Z be the set of subsamples chosen at Step 3 (i.e., an element $a \in Y$ is in Z iff a is tagged with 1). Let E_1 be the event $|Z| \notin [\frac{0.5n}{\log \lambda}, \frac{1.5n}{\log \lambda}]$ (so that E_1 is a superset of the Retry event at Step 3). By Chernoff bound, we have $\Pr[E_1] = \Pr\left[\left||Z| - \frac{n}{\log \lambda}\right| > \frac{0.5n}{\log \lambda}\right] \le e^{-\Omega(\frac{n}{\log \lambda})}$. Then, at Step 4, let Z_{b^*} be the number of b^* bits chosen in Z, and let E_2 be the (superset of the) Retry event $Z_{b^*} < F^*$. In both cases of n_0 at Step 1, the total probability of Retry is at most $\Pr[E_1 \cup E_2]$, and we bound it by considering two cases separately.

If $n_0 < \log^4 \lambda$, then $\Pr[E_1 \cup E_2] \le \Pr[E_1] + \Pr[E_2|\neg E_1]$[7]. As $F^* \le \log^4 \lambda, Z_{b^*} = |Z|$, we have

$$\Pr[E_2|\neg E_1] = \Pr[Z_{b^*} < F^*|\neg E_1] \le \Pr\left[|Z| < \log^4 \lambda \middle| |Z| \in [\frac{0.5n}{\log \lambda}, \frac{1.5n}{\log \lambda}]\right] = 0$$

for all $n \ge \log^6 \lambda$, which implies that $\Pr[E_1 \cup E_2] \le e^{-\Omega(\frac{n}{\log \lambda})} \le e^{-\Omega(n^{1/3})}$.

Otherwise, $n_0 \ge \log^4 \lambda$. To bound $\Pr[E_2]$, let E_3 be the event $F^* > \frac{1}{2}\sqrt{n_{b^*}} \log \lambda$. Then, we have

$$\Pr[E2] \le \Pr[(E_2 \cap \neg E_3) \cup E_3] \le \Pr[E_2 \cap \neg E_3] + \Pr[E_3].$$

The probability of E_3 is implied by Chernoff bound: $\Pr[E_3] = \Pr[F^* > \frac{1}{2}\sqrt{n_{b^*}} \cdot \log \lambda] \le \Pr[|F^* - n_{b^*}| > \frac{1}{2}\sqrt{n_{b^*}} \cdot \log \lambda] \le e^{-\Omega(\log^2 \lambda)}$. Also, we have

$$\Pr[E_2 \cap \neg E_3] = \Pr[(Z_{b^*} < F^*) \cap (F^* \le \frac{1}{2}\sqrt{n_{b^*}} \log \lambda)] \le \Pr[Z_{b^*} < \frac{1}{2}\sqrt{n_{b^*}} \log \lambda].$$

[7]For all events A, B, if $\Pr[\neg A] > 0$, then we have $\Pr[A \cup B] = \Pr[A \cup (B \cap \neg A)] \le \Pr[A] + \Pr[B \cap \neg A] = \Pr[A] + \Pr[B|\neg A] \Pr[\neg A] \le \Pr[A] + \Pr[B|\neg A]$.

Let S be the set of the first n_{b^*} elements of b^* (among total n'_{b^*}) in X, and $\bar{Z}_{b^*}$ be the number of elements in S that is also tagged in Y. Then, $\bar{Z}_{b^*} \le Z_{b^*}$, and $\mathbb{E}[\bar{Z}_{b^*}] = \frac{n_{b^*}}{\log \lambda}$. Using the assumption $n \ge \log^6 \lambda$ and $n_0 \ge \log^4 \lambda$, we have $n_{b^*} \ge \log^4 \lambda$ and $\mathbb{E}[\bar{Z}_{b^*}] > \frac{1}{2}\sqrt{n_{b^*}} \log \lambda$. Hence,

$$\Pr\left[Z_{b^*} < \frac{1}{2}\sqrt{n_{b^*}} \log \lambda\right] \le \Pr\left[\bar{Z}_{b^*} < \frac{1}{2}\sqrt{n_{b^*}} \log \lambda\right] \le \Pr\left[|\bar{Z}_{b^*} - \mathbb{E}[\bar{Z}_{b^*}]| > \mathbb{E}[\bar{Z}_{b^*}] - \frac{1}{2}\sqrt{n_{b^*}} \log \lambda\right].$$

Using Chernoff bound and each b^* bit in S is sampled independently into Y, and then by $\mathbb{E}[\bar{Z}_{b^*}] - \frac{1}{2}\sqrt{n_{b^*}} \log \lambda \ge \frac{1}{2}\mathbb{E}[\bar{Z}_{b^*}]$, the RHS is at most $e^{-\Omega(\mathbb{E}[\bar{Z}_{b^*}])} = e^{-\Omega(\log^3 \lambda)}$. Plugging in $n \ge \log^6 \lambda$ and by union bound, the total probability of Retry is at most $e^{-\Omega(\log^2 \lambda)} = e^{-\Omega(n^{1/3})}$ in both cases.

To prove perfect obliviousness, given n and $n_0 \le n$, observe that the distribution of the access pattern depends only on n and n_0. Hence, it suffices to check every step of PSampleAuxArray.

1. *Approximate initialization.* This step samples a $\{0, 1\}$-Bernoulli random variable for every element, and the access pattern is deterministic.

2. *Number of bits to flip.* This step performs oblivious addition using a tree-like structure, whose access pattern is fixed and deterministic.

3. *Subsampling by $\frac{1}{\log \lambda}$ factor.* This step further samples another Bernoulli random variable for each element, and performs tight compaction. So far, the access pattern is still fixed and deterministic.

   Then, the element with index $\frac{1.5n}{\log \lambda}$ is examined to determine whether Retry is needed. Observe the Retry probability depends only on n_0 and n, and the event of Retry is independent of the element contents and whether there are any previous Retry attempts.[8]

4. *Fine-tuning.* This step uses oblivious sort, which has fixed and deterministic access pattern. Again, the probability of the next Retry event depends only on n_0 and n, and is independent of whether there are previous Retry attempts.

   Moreover, a perfectly oblivious random permutation algorithm (Theorem 3.3) on Y' is performed, whose access pattern is guaranteed to be independent of its input contents.

5. *Reverse-routing.* This step carries out oblivious tight compaction in reverse, whose access pattern is fixed and deterministic.

Finally, from the construction, every $\binom{n}{n_0}$ configuration of choosing n_0 out of n positions to be 0 is equally likely. Moreover, the access pattern is independent of the output configuration. Hence, it follows that **Intersperse** is perfectly oblivious, with the desired performance. □

Now due to the work of Asharov et al. [AKL+20a], we can combine PSampleAuxArray and oblivious tight compaction (Theorem 3.5) to achieve **Intersperse**. This gives rise to the following corollary.

Corollary 4.5 (Parallel **Intersperse**)**.** *Let $\beta \ge 1$ be the number of words used to represent an element. There is an* **Intersperse** *algorithm that is a perfectly oblivious simulation of $\mathcal{F}_{\text{shuffle}}$ on two randomly shuffled input arrays; moreover, except with $\exp(-\Omega(n^{1/3}))$ probability, the algorithm completes in $O(\beta n)$ total work and $O(\log n)$ depth. Moreover, the stated performance bounds also apply in expectation.*

[8] This dependence of Retry event on n_0 and n is exactly the reason why are n_0 and n both revealed. Also notice that the Retry probability is only negligible, so it reveals only a negligible amount of information about n_0, which is acceptable for statistical security, e.g., the manuscript by Asharov et al. [AKL+].

5 One-Time Memory

We describe an abstract data structure called an oblivious one-time memory (OTM) which will serve as a core building block in our OPRAM construction. Roughly speaking, a one-time memory (OTM) is initialized with a set of elements using a procedure called Build. Once initialized, it allows each element stored in it to be looked up *at most once* using a procedure called Lookup. Further, it is assumed that when each lookup request arrives, the request is accompanied by a correct "position label" for the element requested. When the OTM is no longer needed, one can call a Getall operation to extract the set of remaining unvisited elements. A similar notion of oblivious OTM was formulated by Chan et al. [CNS18]. Moreover, assuming that each element can be represented with $\chi \geq 1$ words, Chan et al. [CNS18] show how to construct a perfectly oblivious OTM that consumes $O(\chi n \log n)$ total work to initialize an OTM data structure containing n elements; and where each lookup incurs only $O(\chi)$ overhead.

In this section, we construct an oblivious OTM assuming that the input array of elements provided to the OTM at initialization has already been randomly shuffled (and the randomness hidden from the adversary). Our goal is to allow each lookup to be supported with $O(\chi)$ total work as before (as long as a correct position label accompanies each lookup request); however, we would like the initialization procedure to consume only $O(n \cdot (\chi + \log n))$ total work which is asymptotically better than the OTM of Chan et al. [CNS18] when χ dominates $\log n$. In other words, in our construction, the initialization procedure is allowed to perform only linear work moving and/or copying the fat elements (i.e., a bundle of χ words), but is additionally allowed $O(n \log n)$ amount of computation on metadata.

5.1 Definition

A parallel 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 randomly permuted real elements (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 element identified by its logical address or a dummy address denoted $\perp$ — if the requested element has a real address, we assume that the correct position label is supplied to indicate where in the data structure the requested element is. Finally, when the data structure is no longer needed, one may call a Getall operation to obtain a list of real elements (tagged with their logical addresses) that have not been looked up yet, mixed with an appropriate number of dummies, and permuted according to a secret random permutation.

We require that our oblivious one-time memory data structure retain obliviousness as long as 1) the sequence of real addresses looked up all exist in the data structure (i.e., it appeared as part of the input to Build), and 2) each real address is looked up at most once.

5.1.1 Formal Definition

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 number of batch lookups supported.

The scheme $\mathsf{OTM}^{[n,m,t]}$ is comprised of the following possibly randomized, stateful algorithms (Build, Lookup, Getall), 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 refers to its content.

- $U \leftarrow \mathsf{Build}(S)$: The algorithm takes as input an array S of n elements, where each element is either a *real* key-value pair of the form (k_i, v_i), or *dummy* denoted $(\perp, \perp)$; moreover any two real elements in S must have distinct keys. The algorithm then creates an in-memory data structure to facilitate subsequent lookup requests (not included in the output); moreover it outputs a position-label array U containing exactly n key-position pairs each of the form (k, pos). Further, every real key in the input S will appear exactly once in the list U; and the list U is padded with $\perp$ to a length n.

  Recall that each value v_i in the input S can be "fatter" than its position label pos that is included in the output U. Later in our OPRAM scheme (Section 6), this key-position list U will be propagated back to the parent recursion depth during a coordinated rebuild[9].

- $(v_i : i \in [m]) \leftarrow \mathsf{Lookup}\big((k_i, \mathsf{pos}_i) : i \in [m]\big)$: there are m concurrent Lookup requests in a single batch, where we allow each key k_i requested to be either real or $\perp$. If 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.[10] Moreover, each real (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.

- $R \leftarrow \mathsf{Getall}$: 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 elements inserted during Build but have not been looked up yet, mixed with $\perp$ to a length of n.

Valid request sequence. Our oblivious one-time memory ensures correctness and obliviousness only if the sequence of requests is valid, defined as below. Roughly speaking, a request sequence is valid only if lookups are non-recurrent (i.e., never look for the same real key twice); and moreover the number of batch requests must be exactly the predetermined parameter t. More formally, a sequence of operations is valid, iff:

- The sequence begins with a single call to Build upfront; followed by a sequence of 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.

- Also, in all Lookup operations in the sequence, no two real keys requested (either within the same batch or across 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, if $k_i = \perp$, the i-th answer returned must be $\perp$; 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.

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 elements in R must all be $\perp$.

[9] 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 outputted since U will later on be (externally) needed by the parent recursion depth in our OPRAM construction.

[10] We emphasize this is a major difference between this one-time memory scheme and the oblivious hashing abstraction of Chan et al. [CGLS17]); Chan et al.'s abstraction [CGLS17] allows lookup queries to ask for keys that do not exist in the data structure.

Perfect obliviousness. For obliviousness, we require that there exists a simulator $\mathsf{Sim}(1^n, 1^m, 1^t)$ that takes in only the length of the input array provided to Build, the number of requests in a concurrent batch, and the total number of batched requests the OTM must support, such that the following holds. For any input array S consisting of n elements, any sequence of batched requests $K := \{k_{i,j}\}_{i\in[t],j\in[m]}$ such that every key queried must appear in S and moreover, every key is looked up at most once in the same batch or across batches, the following real- and ideal-world distributions must be identical:

- *Real-world.* Consider the following real-world experiment.
  1. *Randomly shuffle* the input array S; and run Build on the outcome;
  2. Make a sequence of t batch Lookup operations defined by K, and in every request in any batch, provide the correct position labels as defined by the output U of Build;
  3. Run Getall and let R be the resulting array.
  4. The real-world distribution is defined by the tuple $(\mathsf{Addresses}, R)$ where Addresses is the access patterns incurred by the OTM in the above experiment.
- *Ideal world.* The ideal-world experiment outputs the following joint distribution:

$$\left(\mathsf{Sim}(1^n, 1^m, 1^t), \mathcal{F}_{\text{getall}}(S, K)\right),$$

  where $\mathcal{F}_{\text{getall}}(S, K)$ is the ideal functionality that performs the following: mark every entry in S whose key is contained in K as dummy, randomly shuffle the resulting array and output it.

5.2 Construction

5.2.1 Intuition

We would like achieve the following obliviously. When Build receives an input array of elements, we want to create 1) an array A containing all real elements in the input and an appropriate number of dummies such that all elements are randomly shuffled; and 2) a dummy metadata array denoted dummy that contains a randomly permuted list of the locations of dummy elements in A. When a batch of m Lookup requests arrive, each of the m requests is either a real request tagged with the desired element's correct position in A; or it is a dummy request denoted as $\perp$. Imagine that there are m CPUs, i.e., one for serving each of the m requests. The m CPUs find the next m unvisited dummy positions from the array dummy, denoted $\mathsf{dpos}_1, \ldots, \mathsf{dpos}_m$. For each CPU $i \in [m]$, if it received a real request, it fetches the element from the specified position (that accompanies the request); otherwise it fetches a dummy element from position dpos_i. Finally, Getall simply removes all the visited locations from A and returns the remaining unvisited elements — it is not hard to see that in the array returned by Getall, all elements are randomly shuffled. We stress that the number of dummy elements in the array A must be sufficient to support the number of lookup queries to the OTM.

The main challenge is how to realize the Build procedure obliviously consuming only linear total work on the (possibly fat) elements but allowing $O(n \log n)$ total work on metadata. Here we exploit the fact that the input array has already been randomly shuffled and the randomness hidden from the adversary. Therefore, we only have to pad the input array with an appropriate number of dummies and intersperse this concatenated array. For building the dummy metadata array dummy, we only have to deal with metadata, thus we can rely on standard oblivious sorting techniques.

5.2.2 Detailed Construction

Build aims to create 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 added dummies and n denotes the number of real elements. Each entry of the array A (real or dummy alike) contains a key-value pair $(\mathsf{key}, \mathsf{val})$ (where val can be of large size).

2. An array dummy of $\widetilde{n}$ indices that indicate the positions of the added dummies within A, and a counter count that keeps track of how many elements have been looked up so far.

These in-memory data structures, $(A, \mathsf{dummy}, \mathsf{count})$, will then be updated during Lookup.

Build Algorithm $\mathsf{Build}\,((k_i, v_i) : i \in [n])$ proceeds as follows.

1. *Initialize.* In parallel, construct an array A_1 of length n are copied from the input, an array A_0 of length $\widetilde{n}$ with entries set to $(\perp, \perp)$.

2. *Permute real and dummy elements.* Perform Parallel **Intersperse** (Section 4) on the arrays A_0, A_1 by interleaving the n elements from A_1 with the $\widetilde{n}$ elements from A_0. The resulting permuted array is the A in the data structure.

3. *Construct the key-position map U.* The map U is constructed in the following steps.
   a) Let M be a metadata array of length $n+\widetilde{n}$, where the entries of M are of the form $(\mathsf{key}, \mathsf{pos})$, and $\mathsf{pos} \in [1..n+\widetilde{n}]$ will index a position within the array A. For each $i \in [n+\widetilde{n}]$ in parallel, set $M[i].\mathsf{key} := A[i].\mathsf{key}$ and $M[i].\mathsf{pos} = i$.
   b) Obliviously sort the array M on the keys to produce an array $\widehat{M}$; we use the convention that the extra dummy keys $\perp$'s are at the end.
   c) We construct the key-position map U from the first n entries of $\widehat{M}$ — recall that each entry of U is of a key-position pair (k, pos).

4. *Construct the* dummy *indices.* For each $i \in [1..\widetilde{n}]$, we denote $\widehat{M}_n[i] := \widehat{M}[n + i]$. Perform a perfectly oblivious random permutation (ORP, Section 3.2.2) on $\widehat{M}_n[1..\widetilde{n}]$ (which contain only metadata). We then construct the array of dummy indices: for $i \in [1..\widetilde{n}]$ in parallel, we set $\mathsf{dummy}[i] := \widehat{M}_n[i].\mathsf{pos}$.

   We initialize the counter $\mathsf{count} := 0$.

At this moment, the data structure $(A, \mathsf{dummy}, \mathsf{count})$ 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.

If we instantiate **Intersperse** using the algorithm corresponding to Corollary 4.5, and instantiate the oblivious random permutation using the algorithm corresponding to Theorem 3.3, we obtain the following fact — for simplicity, throughout Sections 5 and 6, we will focus on the *expected* performance. Later in Section 7, we will describe how to obtain high-probability performance bounds (where we will need to instantiate small instances with non-Las-Vegas algorithms with deterministic performance bounds).

Fact 1. *The* Build *algorithm completes in* $O\big((n+\widetilde{n}) \cdot (\chi + \log(n+\widetilde{n}))\big)$ *total work and* $O(\log(n+\widetilde{n}))$ *depth in expectation.*

As mentioned before, when the elements can be "fat" and the metadata is "thin", our **Build** is asymptotically more efficient than that of Chan et al. [CNS18]. We now prove the above fact.

Proof. The depth is dominated by **Intersperse**, which follows by Corollary 4.5. The total work on elements is dominated by the **Intersperse** procedure on (A_0, A_1), which is $O\big((n+\widetilde{n})\chi\big)$ by Corollary 4.5. On metadata, the dominating subroutines are the oblivious sort (realized with the AKS sorting network [AKS83]) on M and the oblivious random permutation on $\widehat{M}$ (Theorem 3.3). Hence, the summed total work is $O\left((n+\widetilde{n})\cdot(\chi+\log(n+\widetilde{n}))\right)$. □

Lookup We implement a batch of m concurrent lookup operations $\mathsf{Lookup}\left((k_i, \mathsf{pos}_i) : i \in [m]\right)$ 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 $\mathsf{pos} := \mathsf{dummy}[\mathsf{count}+i]$, i.e., the position to fetch from is the next indexed dummy. (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 value from $A[\mathsf{pos}]$, mark $A[\mathsf{pos}] := \perp$.

3. *Update counter.* The counter is only updated once per batch request: $\mathsf{count} := \mathsf{count} + m$.

4. *Return.* Return the value read in the above Step 2.

The following fact is straightforward from the algorithm.

Fact 2. *The* Lookup *algorithm runs in $O(m\chi)$ total work and $O(1)$ depth.*

Getall By always having exactly t batch requests, there are exactly $\widetilde{n}$ entries in A have been accessed during previous Lookup operations. Our goal is to remove these accessed entries and output a list of remaining unvisited entries. Note that the algorithm *need not hide* which entries have been accessed since this information has already been observed by the adversary.

It is not hard to see that we can accomplish this removal in $O\big((n+\widetilde{n})\cdot\chi\big)$ total work and $O(\log(n+\widetilde{n}))$ depth. Basically, the algorithm boils down to an all-prefix-sum calculation: suppose we write down $b_i := 0$ if the i-th element has been visited and write down $b_i := 1$ otherwise. Let $s_i := \sum_{j=[i]} b_i$ denote the prefix sum up to index i. We will then assign i-th CPU to grab the i-th element and if it is unvisited, the CPU places it at index s_i in the final output array.

To compute all prefix sums in parallel, we can rely on a binary tree — without loss of generality, assume that $n+\widetilde{n}$ is a power of 2.

1. Consider a binary tree with $n+\widetilde{n}$ leaves where the i-th leaf is tagged with the bit b_i. Every node in the tree wants to compute two sums: 1) a *subtree sum* that sums up all leaves in its own subtree; and 2) a *prefix sum* defined as the sum of the entire prefix upto the rightmost child in its subtree.
2. First, compute the subtree sums of all nodes in $O(n+\widetilde{n})$ total work and $O(\log(n+\widetilde{n}))$ depth using the most natural algorithm: from the leaf level to the root, every node sums up the subtree sums of its two children.
3. Next, compute all nodes' prefix sums in the following fashion. First, the prefix sum of the root is the same as its subtree sum. Now, a node in the tree can calculate its own prefix sum as long as its parent has calculated its prefix sum:

- If the node is the *left* child of some parent, its prefix sum is its parent's prefix sum minus its sibling's subtree sum;
- If the node is the *right* child of some parent, simply copy the parent's prefix sum.

It is not hard to see that when executed in parallel, the above algorithm completes in $O(n+\widetilde{n})$ total work and $O(\log(n+\widetilde{n}))$ depth. We thus have the following fact.

Fact 3. *The* Getall *algorithm runs in* $O\big((n+\widetilde{n})\cdot\chi\big)$ *total work and* $O(\log(n+\widetilde{n}))$ *depth.*

Lemma 5.1 (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 $S=((k_i,v_i):i\in[n])$ and $K\subseteq\{k_i\}_{i\in[n]}$, the real-world distribution of $(\mathsf{Accesses},R)$ is identical to the ideal-world $\big(\mathsf{Sim}(1^n,1^m,1^t),\mathcal{F}_{\text{getall}}(S,K)\big)$. We proceed by defining Sim, then we show the real-world Accesses is identical to Sim and that the marginal distribution of R is identical to $\mathcal{F}_{\text{getall}}(S,K)$.

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.

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

Third, simulating the access patterns of Getall is done in the most natural manner since the access pattern of Getall is a deterministic function of the access pattern of the second step, Lookup.

Observe the list S is randomly permuted upfront (before Build) in the real-world and the added dummies $(\perp,\perp)$ are also randomly permuted (as dummies differ only in the metadata $\widehat{M}$). Then, in the array A generated by Build, every real and dummy element will be in a random location by **Intersperse** (Corollary 4.5). With a valid request sequence, the real-world algorithm Lookup accesses each real or dummy element at most once, and thus every real-world access visits a random position of the array A (besides reading and writing dummy and count). Hence, the marginal distribution of Accesses is identical to the output of Sim.

For the marginal variable R of real-world experiment, we use again that, in the array A generated by Build, every real and dummy element is in a random location. Conditioning on any fixed access pattern in the real world, the unvisited locations holds still a random unvisited real element or a random unvisited dummy $(\perp,\perp)$. As R consists of all the unvisited real and dummy elements in the sequential ordering, it is identical to the ideal output $\mathcal{F}_{\text{getall}}(S,K)$. □

Summarizing the above Fact 1, 2, 3, and Lemma 5.1, we conclude with the following theorem.

Theorem 5.2. *The above scheme* (Build, Lookup, Getall) *is a perfectly oblivious (parallel) one-time memory. Assume that each element can be represented as* χ *words, the performance is:*

- Build*:* $O\big((n+\widetilde{n})\cdot(\chi+\log(n+\widetilde{n}))\big)$ *total work and* $O(\log(n+\widetilde{n}))$ *depth in expectation,*
- Lookup*:* $O(m\chi)$ *total work and* $O(1)$ *depth, and*
- Getall*:* $O\big((n+\widetilde{n})\cdot\chi\big)$ *total work and* $O(\log(n+\widetilde{n}))$ *depth.*

6 OPRAM

In this section we will put together the building blocks constructed earlier and obtain our final OPRAM scheme.

Terminology. Adopting the terminology of earlier works on ORAMs [SvDS+13, WCS15, SCSL11] and OPRAMs [CS17, BCP16], we use the term *block* to refer to a word of the PRAM. Recall that the PRAM makes batches of requests where each batch is of size m. Our construction uses $\chi = \Theta(\log N)$ to denote a "branching factor" which we assume is a power of 2 without loss of generality.

6.1 Overview

Recursive OPRAMs. Let $D := \log_\chi \frac{N}{m}$. Our OPRAM construction consists of $D+1$ position-based OPRAMs (defined and constructed in Section 6.2) henceforth denoted OPRAM_0, OPRAM_1, OPRAM_2, ..., OPRAM_D — we also refer to them as $D+1$ recursion depths. Each position-based OPRAM denoted OPRAM_d consists of $d \log_2 \chi + 1$ levels geometrically growing (with factor 2) in size, where each level is a one-time oblivious memory scheme as defined and described in Section 5.

For $d < D$, OPRAM_d stores $\Theta(\chi^d \cdot m)$ fat-blocks where each fat-block is a bundle of χ normal blocks (i.e., χ words). The last recursion depth OPRAM_D stores the actual *data blocks*. Henceforth every OPRAM_d where $d < D$ is said to be a *metadata OPRAM*; since these OPRAMs jointly store a logical index structure for discovering the position labels of the desired (fat-)blocks in the next recursion depth. The last OPRAM_D is called the *data OPRAM* since it stores the actual data blocks.

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-$(\log_2 m + d \log_2 \chi)$ 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 fat-blocks stored at OPRAM_0. Since each batch consists of m concurrent lookups, one of them will be responsible for this fat-block in OPRAM_0.

For $d < D$, a fat-block with the address $\mathsf{addr}^{\langle d \rangle}$ in OPRAM_d stores the position labels for χ (fat-)blocks in OPRAM_{d+1}, at addresses $\{\mathsf{addr}^{\langle d \rangle} || s : s \in \{0,1\}^{\log_2 \chi}\}$. Henceforth, we say that these χ addresses are *siblings* to one another.

6.2 Position-Based OPRAM

A position-based OPRAM is almost a fully functioning OPRAM except that whenever a batch of memory requests come in, we assume that *each request must be tagged with a correct position label indicating exactly where the requested block is in the OPRAM.* In our subsequent full OPRAM construction, to fetch a data block in OPRAM_D, we recursively request the block's position label from OPRAM_{D-1} first — once a correct position label is obtained, we may begin accessing OPRAM_D for the desired block. In other words, every OPRAM_d stores the position labels for the next OPRAM_{d+1}.

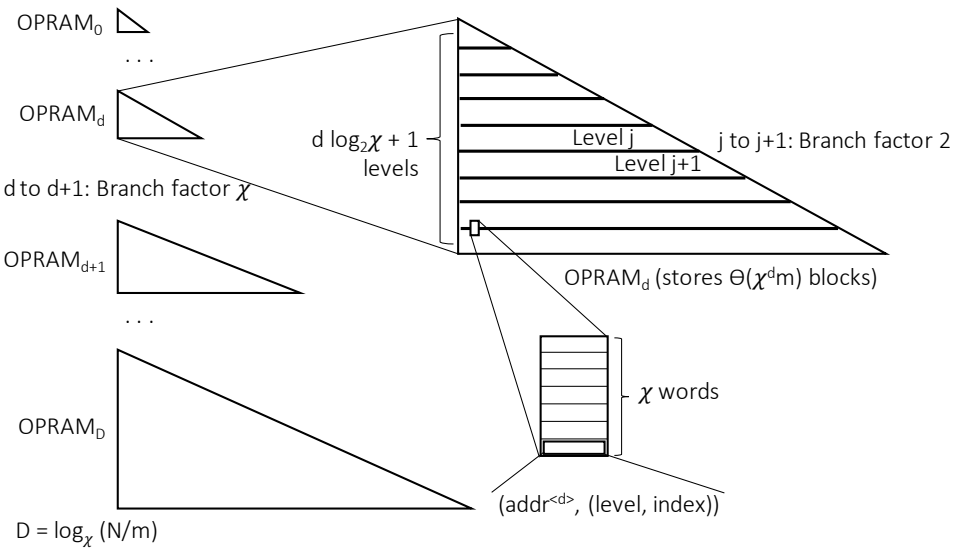

Figure 1: OPRAM data structure.

6.2.1 Data Structure

We next describe OPRAM_d for some $1 \le d \le D = \log_\chi \frac{N}{m}$. As we shall see, the case OPRAM_0 is trivial and is treated specially.

Hierarchical levels. The position-based OPRAM_d consists of $d \log_2 \chi + 1$ levels henceforth denoted as $(\mathsf{OTM}_j : j = 0, \ldots, d \log_2 \chi)$ 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 (fat- or data) blocks, m concurrent lookups in each batch (which can all be real), and 2^j batch requests. This means that for every OPRAM_d, the smallest level is capable of storing up to m real fat-blocks. Every subsequent level can store twice as many real (fat- or data) blocks as the previous level. For the largest OPRAM_D, its largest level is capable of storing N real data blocks given that $D = \log_\chi \frac{N}{m}$ — this means that the total space consumed is $O(N)$. Plugging in the hierarchical levels, the full OPRAM data structure is illustrated in Figure 1.

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 index within the level the block resides at.

Augmented block. We assume that each fat block or data block is of the form (logical address, payload), i.e., each block carries its own logical address. This will simplify the procedure MergeLevels later.

6.2.2 Operations

Each position-based OPRAM supports two operations, Lookup and Shuffle. In the following, we describe the algorithms Lookup and Shuffle for every OPRAM_d where $d \ge 1$, and then we will describe the Lookup and Shuffle for the trivial case, OPRAM_0.

Lookup. Every batch lookup operation, denoted $\mathsf{Lookup}\big((\mathsf{addr}_i, \mathsf{pos}_i) : i \in [m]\big)$ 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 $j = 0, \ldots, d \log_2 \chi$ in parallel, perform the following:
   - For each $i \in [m]$ in parallel, perform the following:
     If pos_i indicates that the block should be stored in level j, then set $\mathsf{addr}'_i := \mathsf{addr}_i$ and let $\mathsf{pos}'_i := \mathsf{pos}_i$ (and specifically the part of the position label denoting the offset within level j); otherwise, set $\mathsf{addr}'_i := \perp$ and $\mathsf{pos}'_i := \perp$.
   - $(v_{ij} : i \in [m]) \leftarrow \mathsf{OTM}_j.\mathsf{Lookup}((\mathsf{addr}'_i, \mathsf{pos}'_i) : i \in [m])$.
2. For each $i \in [m]$ in parallel, perform the following:
   set val_i to be the only non-dummy element in $(v_{ij} : j = 0, \ldots, d \log_2 \chi)$, if it exists; otherwise set $\mathsf{val}_i := \perp$. This step can be accomplished using an oblivious select operation in $O(\log d + \log\log \chi)$ depth consuming $d \log_2 \chi$ CPUs.
3. Return $(\mathsf{val}_i : i \in [m])$.

The following follows by Fact 2, the total work of $\mathsf{OTM.Lookup}$.

Fact 4. *For* OPRAM_d*, for each* Lookup *containing a batch of* m *requests, the total work is* $O(m \cdot \chi \cdot d \cdot \log_2 \chi)$*, and the depth is* $O(\log d + \log\log \chi)$*.*

Shuffle. A shuffle operation, denoted $\mathsf{Shuffle}(U, \ell, A_0)$, receives as input an update array U (we will define constraints on U subsequently), the level ℓ to be rebuilt, and an array A_0 of m fat- or data blocks. For each OPRAM_d it must be guaranteed that $\ell \le d \log_2 \chi$; moreover, the operation is called only when level ℓ is empty or $\ell = d \log_2 \chi$. The $\mathsf{Shuffle}$ algorithm is triggered by a previous Lookup instance, which fetches the m fat- or data blocks and then passes these m blocks in the array A_0. For the case $d = D$, the contents of these data blocks might possibly be updated.

The $\mathsf{Shuffle}$ algorithm then combines levels $0, 1, \ldots, \ell - 1$ into level ℓ: 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.

For $d = D$, the update array $U = \emptyset$; for $d < D$, the update array U must satisfy the following validity requirement. Let $A := A_0 \cup (\bigcup_{i=0}^{\ell} \mathsf{OTM}_i.\mathsf{Getall})$, where the operator $\cup$ denotes union. We shall see that each entry of the update array U contains a pair of depth-$(d+1)$ address and the corresponding updated position label in OPRAM_{d+1}; moreover, if a real depth-$(d+1)$ address appears in U, then its depth-d prefix address must appear in A, whose fat-block will need update.

In our full OPRAM scheme later, the update array U will be passed from the immediate larger OPRAM_{d+1}, and contains the new position labels that OPRAM_{d+1} has chosen for recently accessed logical addresses.

As we later see, $\mathsf{Shuffle}$ actually needs to be performed in parallel across all recursion depths. Hence, $\mathsf{Shuffle}(U, \ell, A_0)$, is actually broken into two phases as follows.

The first phase is $\mathsf{MergeLevels}(\ell, A_0)$:

1. **Randomly Interspersing Adjacent Levels.** Apply oblivious random permutation (Theorem 3.3) to the m elements in A_0. Then, for i from 1 to ℓ, perform the following:
   $A_i := \textbf{Intersperse}(A_{i-1}, \mathsf{OTM}_{i-1}.\mathsf{Getall})$.
   At this moment, only A_ℓ needs to be kept; the old OTM_0, …, OTM_ℓ instances and intermediate A_i's (for $i < \ell$) may be destroyed.

2. Let $(\mathsf{OTM}', U') \leftarrow \mathsf{OTM}_\ell.\mathsf{Build}(A_\ell)$ (recall that each block in A_ℓ is augmented to carry its own logical address).

3. 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.

4. Finally, output U' (in our full OPRAM construction later, U' will be passed to the the next (i.e., immediately smaller) position-based OPRAM as the update array for performing its shuffle.

The second phase is $\mathsf{UpdateLevel}(U, \ell)$:

1. **Updating Positions.** For $d = D$, U should be empty and this phase is skipped; for $d < D$, we perform the following.

   Recall that in OTM_ℓ, internally there is an array A of length $2 \cdot m \cdot 2^\ell$, where each non-dummy entry is a fat-block containing χ positions in OPRAM_{d+1}, where each position contains the level number and the index within that level. In parallel, convert each fat-block into χ sibling entries, each of which is a pair of depth-$(d+1)$ address together with the corresponding position. We use M to denote the resulting array that contains $2\chi \cdot m \cdot 2^\ell$ such entries.

2. Next, observe that each entry of U is also a pair of depth-$(d+1)$ address together with its updated position. However, since U contains at most $m \cdot 2^\ell$ entries, we want to avoid performing oblivious sort on the whole of M and U.

   The key insight is that the position of an entry in M contains the level number (in OPRAM_{d+1}). This position needs to be updated *iff* its level is at most ℓ. Hence, each such entry can be marked as *special*.

3. Using oblivious tight compaction (Section 3.5), we can produce an array M' of size $m \cdot 2^\ell$ that contains all these special entries. While performing oblivious tight compaction, we also record the information (on the server) that later allows us to obliviously reverse the movements.

4. The entries of M' are updated using the positions in U via oblivious routing (Theorem 3.4), which can be implemented by oblivious sorts.

5. Then, using the recorded information (on server), the movements of tight compaction are reversed such that the updated positions in M' get obliviously routed back to the special entries in M. Finally, from the updated array M, we get back the corresponding updated array A, which also forms the updated OTM_ℓ.

Fact 5. *For* OPRAM_d*, let* $\ell \leq d \log_2 \chi$*, then the above two phases of* $\mathsf{Shuffle}(U, \ell, A)$ *runs in* $O(m \cdot 2^\ell \cdot (\chi + \ell + \log m))$ *total work and* $O(\ell \cdot (\log m + \ell))$ *depth in expectation*[11]*.*

Proof. From the description of the algorithm, we analyze the two phases.

- The first phase $\mathsf{MergeLevels}$ is dominated by **Intersperse** and $\mathsf{OTM}_\ell.\mathsf{Build}$. The ℓ instances of **Intersperse** run on fat-blocks of size χ, which takes $O(m\chi \cdot 2^\ell)$ total work and $O(\ell \cdot (\log m + \ell))$ depth. By Fact 1, $\mathsf{OTM}_\ell.\mathsf{Build}$ runs in $O(m \cdot 2^\ell(\chi + \ell + \log m))$ total work and $O(\log m + \ell)$ depth in expectation. Note that if $d = D$, the elements are normal data blocks of size $O(1)$, which is strictly bounded by χ, and hence the total work holds as well.

[11] Later in Section 7, we will describe how to obtain high-probability performance bounds.

- The second phase UpdateLevel is not used if $d = D$. For $d < D$, updating the positions from U takes $O(m \cdot 2^\ell \cdot \chi)$ total work on address-position pairs due to tight compaction and $O(m \cdot 2^\ell \cdot (\ell + \log m))$ operations due to oblivious sorting, which sums up to the claimed total work. The depth is dominated by tight compaction, $O(\log m + \ell)$.

Combining the above gives the result. □

Trivial Case: OPRAM_0. 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 $O(\chi m \log m)$ 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), $\mathsf{Shuffle}(U, 0, A_0)$ can be implemented by oblivious sorting, where U contains the updated fat-block contents and A_0 is empty for OPRAM_0. To elaborate, $\mathsf{OPRAM}_0.\mathsf{MergeLevels}$ shuffles A and outputs $\emptyset$, and $\mathsf{OPRAM}_0.\mathsf{UpdateLevel}$ takes U as input and updates the contents of A. It takes $O(\chi m \log m)$ total work and $O(\log m)$ depth.

6.2.3 Analysis of Position-Based OPRAM

Fact 6. *The construction of position-based OPRAM 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. Given as input the correct position labels, the Lookup of position-based OPRAM passes the labels to OTM.Lookup, and hence the correctness follows. □

In our position-based OPRAM construction, for every OPRAM_d at recursion depth d, the following invariants are respected by construction as stated in the following facts. For any recursion depth d, denote $L(d) := d \log_2 \chi$ as the largest level in OPRAM_d.

Fact 7. *For every* OPRAM_d*, every* OTM_ℓ *instance at level $\ell \le L(d)$ that is created needs to answer at most 2^ℓ batches of m requests before* OTM_ℓ *instance is destroyed.*

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

Fact 8. *For every* OPRAM_d *and every* OTM_ℓ *instance at level $\ell \le L(d)$, 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_ℓ*.*

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 $\ell \leq L(d)$, 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). Finally, claim (b) follows from correctness of the position labels (Fact 6). □

6.3 Detailed OPRAM Scheme

6.3.1 Operations

Upon receiving a batch of m requests denoted as $((\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i) : i \in [m])$, 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 χ-bit flags_i that indicates whether each of the χ sibling addresses $\{\mathsf{addr}_i^{\langle d \rangle} || s : s \in \{0,1\}^{\log_2 \chi}\}$ is requested in OPRAM_{d+1}.

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

   (a) 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, 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.

   (b) Using $\widehat{A}^{\langle d+1 \rangle}$, for each $i \in [1..m]$, we produce an entry $(\mathsf{addr}_i^{\langle d \rangle}, \mathsf{flags}_i)$ according to the following rules:

      i. If $\mathsf{addr}_i^{\langle d+1 \rangle}$ is a dummy, then $\mathsf{addr}_i^{\langle d \rangle} := \perp$ and $\mathsf{flags}_i := \perp$ are also dummy.

      ii. Observe that all addresses with the same depth-d prefix are grouped together. Hence, within such a group, we only need to keep the last one (truncated to its depth-d prefix) and add a χ-bit flag to indicate which of the sibling addresses are present. All other addresses are set to dummy.

   (c) The batch access for OPRAM_d is $((\mathsf{addr}_i^{\langle d \rangle}, \mathsf{flags}_i) : i \in [m])$.

   As noted by prior works [BCP16, CLT16, CS17], conflict resolution can be completed by employing oblivious sorting.

2. **Fetch.** For $d = 0$ to D sequentially, perform the following:

   - For each $i \in [m]$ in parallel: let $(\mathsf{addr}_i^{\langle d \rangle}, \mathsf{flags}_i)$ be the depth-d result of conflict resolution.

- 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 $P^{\langle d \rangle}$ 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. We use A_d to denote the m (fat- or data) blocks returned from the lookup of OPRAM_d, and proceed with the following two cases of d.
- If $d < D$, each lookup from a non-dummy $(\mathsf{addr}_i^{\langle d \rangle}, \mathsf{flags}_i)$ will return positions for the χ sibling addresses $\{\mathsf{addr}_i^{\langle d \rangle} || s : s \in \{0,1\}^{\log_2 \chi}\}$. The χ bits in flags_i will determine whether each of these χ position labels will be "needed" later in the lookup of OPRAM_{d+1}.

  At recursion depth $d+1$, there are m CPUs waiting for the position labels corresponding to $\{\mathsf{addr}_i^{\langle d+1 \rangle} : i \in [m]\}$. At depth d, there are χ (real or dummy) labels per CPU. To get the m labels needed at depth $d+1$, run tight compaction on the $\chi \cdot m$ labels such that moves the m needed positions to the front. Now, using oblivious routing (see Theorem 3.4), the m position labels $P^{\langle d+1 \rangle}$ can be delivered to the m CPUs at recursion depth $d+1$,
- If $d = D$, A_D will contain the data blocks fetched. Recall that conflict resolution was used to suppress duplicates. Hence, oblivious routing can be used to deliver each data block to the corresponding CPUs that request it.

3. **Maintain.** We first consider depth D. For every $i \in [m]$ in parallel: set $u_i := (\mathsf{addr}_i^{\langle D \rangle}, \mathsf{data}_i)$, where data_i is the updated data block for the address $\mathsf{addr}_i^{\langle D \rangle}$ (or just the original data block if it is not modified). Set the array $A_D := \{u_i : i \in [m]\}$ and $U^{\langle D \rangle} := \emptyset$. Suppose that $\ell^{\langle D \rangle}$ is the smallest empty level in OPRAM_D.

   For $1 \leq d < D$, recall A_d denote the m fat-blocks that are returned from the lookup of OPRAM_d; for the trivial case $d = 0$, $A_0 := \emptyset$. By the construction of OPRAM_d, we have the invariant that for all $0 \leq d < D$, if $\ell^{\langle D \rangle} < d \log_2 \chi$ (recall that OPRAM_d consists of $d \log_2 \chi$ levels), then $\ell^{\langle D \rangle}$ is also the smallest empty level in OPRAM_d.

   For $d := D$ down to 0, each of the following steps is done **in parallel** across different d's:

   - If $d \log_2 \chi < \ell^{\langle D \rangle}$, set $\ell_d := d \log_2 \chi$; otherwise, set $\ell_d := \ell^{\langle D \rangle}$.
   - Call $U^{\langle d-1 \rangle} \leftarrow \mathsf{OPRAM}_d.\mathsf{MergeLevels}(\ell_d, A_d)$.
   - After the previous step, all $U^{\langle d \rangle}$'s are ready. Hence, we call $\mathsf{OPRAM}_d.\mathsf{UpdateLevel}(U^{\langle d \rangle}, \ell_d)$.

6.3.2 Analysis of OPRAM Scheme

Obliviousness. Given Fact 7 and 8, our OPRAM construction maintains perfect obliviousness.

Lemma 6.1 (Obliviousness)**.** *The above OPRAM construction satisfies perfect obliviousness.*

Proof. For every parallel one-time memory instance constructed during the lifetime of the OPRAM, Facts 7 and 8 are satisfied, and thus every one-time memory instance receives a valid request sequence. Putting together the perfect obliviousness of the parallel one-time memory scheme (Lemma 5.1) and the **Intersperse** (Corollary 4.5), the position-based OPRAM is perfectly oblivious — the output of $\mathsf{OTM}.\mathsf{Getall}$ is uniformly random shuffled, then the result of **Intersperse** is uniformly random shuffled, and then the input to the subsequent $\mathsf{OTM}.\mathsf{Build}$ is uniformly random shuffled, which implies such OTM is perfectly oblivious. The perfect obliviousness of the position-based OPRAM and then the full OPRAM follows by observing that all other access patterns of the construction are identically distributed and independent of the input requests. □

Efficiency. We now analyze the asymptotical efficiency of our OPRAM construction. First, observe that the asymptotical performance of the conflict resolution and fetch phases as stated in the following fact.

Fact 9. *The fetch phase can be completed using* $O(Dm\chi \log N) = O(\frac{m \log^3 N}{\log \log N})$ *total work and* $O(D \cdot (\log \log N + \log m)) = O(\log N \cdot (\log \log N + \log m))$ *depth.*

Proof. The factor D comes from the number of recursion depths. For each recursion depth, the following costs are incurred: 1) Within each recursion depth, there are $O(\log \frac{N}{m})$ hierarchical levels. Each of the m requests accesses and computes on one fat- or data block per level (Fact 4). 2) The routing between adjacent depths can be implemented with the AKS sorting network [AKS83] that moves $O(m \log m)$ fat-blocks, which takes $O(\chi m \log m)$ total work and $O(\log m)$ depth. Hence, the total work is

$$O(Dm\chi \log \frac{N}{m}) + O(D\chi m \log m) = O(Dm\chi \log N) = O\left(\frac{m \log^3 N}{\log \log N}\right),$$

because $\chi = \Theta(\log N)$. The depth is $O(D \cdot (\log \log N + \log m))$, where $O(\log \log N)$ is the depth of each $\mathsf{OPRAM}_d.\mathsf{Lookup}$ (Fact 4). □

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

Fact 10. *Let* T *be the total steps of the original PRAM (where each step contains* m *memory concurrent requests). Using* m *CPUs, the maintain phase of OPRAM takes* $T \cdot O\left(\frac{\log^3 N}{\log \log N}\right)$ *parallel steps in expectation.*

Proof. After every 2^ℓ batch of m requests, for each OPRAM_d, the level ℓ is reconstructed. Due to Fact 5, each such reconstruction will take $O(m \cdot 2^\ell \cdot (\chi + \ell + \log m))$ total work in expectation for each d. Summing from $d = 0$ to D, the total work is $O(D \cdot m \cdot 2^\ell \cdot (\chi + \ell + \log m))$. The depth of all D reconstructions is still $O(\ell \cdot (\log m + \ell))$ from Fact 5 because both $\mathsf{OPRAM}_d.\mathsf{MergeLevels}$ and $\mathsf{OPRAM}_d.\mathsf{UpdateLevel}(U^{\langle d \rangle}, \ell_d)$ are performed **in parallel** across different d's. Because the depth is less than the total work divided by m, the number of parallel steps is just $O(D \cdot 2^\ell \cdot (\chi + \ell + \log m))$ for each ℓ.

Then, during T batch of requests, for each ℓ such that $2^\ell \le T$, it takes $\lceil \frac{T}{2^\ell} \rceil \cdot O(D \cdot 2^\ell \cdot (\chi + \ell + \log m)) = T \cdot O(D \cdot (\chi + \ell + \log m))$ parallel steps. Summing over all levels $\ell = 0$ to $D \log_2 \chi$, the total number of parallel steps is

$$T \cdot O\left(D \cdot \left((\chi + \log m) D \log_2 \chi + D^2 \log^2 \chi\right)\right) = T \cdot O\left(\frac{\log^3 N}{\log \log N}\right)$$

in expectation, because $\chi = \Theta(\log N)$ and $D = O(\frac{\log N}{\log \chi})$. □

Theorem 6.2. *The OPRAM construction achieves* $O\left(\frac{\log^3 N}{\log \log N}\right)$ *simulation overhead in expectation for any sequence of operations.*

Proof. Straightforward from Lemma 6.1, Facts 9 and 10. □

7 High-Probability Performance Bounds

So far, we have focused on the expected performance of our OPRAM construction. We can in fact upgrade our performance guarantees to hold with high probability. Specifically, let λ denote a desired security parameter, and suppose that we would like our performance bounds to hold with $1 - \mathsf{negl}(\lambda)$ probability for some negligible function $\mathsf{negl}(\cdot)$. To achieve this, it suffices to replace all small instances of oblivious random permutation and **Intersperse** with non-Las-Vegas algorithm whose performance bounds hold deterministically. Recall that the Las Vegas building blocks we use, on problems of size n, can fail to achieve the stated performance bounds with $\exp(-\Omega(n^\epsilon))$ probability for some appropriate constant $\frac{1}{3} \le \epsilon < 1$. Thus it suffices to use non-Las-Vegas counterparts (Theorem 3.2 and Corollary 4.2) for instances of size $n < \log^6 \lambda$ to obtain a performance failure probability that is negligibly small in λ.

Below we restate the key facts and theorems assuming that

1. for $n < \log^6 \lambda$: oblivious random permutation is instantiated with Theorem 3.2 and **Intersperse** is instantiated with Corollary 4.2;
2. large instances where $n \ge \log^6 \lambda$ are still instantiated using Theorem 3.3 and Corollary 4.5.

Building blocks. We first state the high-probability performance bounds for one-time memory and and position-based OPRAM.

Fact 11 (High probability performance bounds for one-time memory)**.** *There exists a perfectly oblivious one-time memory scheme with the following performance. For $n \ge \log^6 \lambda$, the* Build *algorithm completes in $O\big((n+\widetilde{n}) \cdot (\chi + \log(n+\widetilde{n}))\big)$ total work and $O(\log(n+\widetilde{n}))$ depth, except with probability $e^{-\Omega(\log^2 \lambda)}$; for $n < \log^6 \lambda$, it completes, with probability 1, in $O\big((n+\widetilde{n}) \cdot (\chi + \log^2(n+\widetilde{n}))\big)$ total work and $O\left(\log^2(n+\widetilde{n})\right)$ depth.*

Proof. The $n \ge \log^6 \lambda$ case is identical to the proof of Fact 1, and the $n < \log^6 \lambda$ case follows by Theorem 3.2 and Corollary 4.2. □

Fact 12 (High probability performance bounds for position-based OPRAM)**.** *For any* OPRAM_d*, let $\ell \le d \log_2 \chi$, then the above two phases of* $\mathsf{Shuffle}(U, \ell, A)$ *have the following performance:*

- *For $m \cdot 2^\ell < \log^6 \lambda$, with probability 1, it completes in $O\big(m \cdot 2^\ell \cdot (\chi + (\ell + \log m)^2) + \chi m \log m\big)$ total work and $O(\ell \cdot (\ell + \log m)^2)$ depth.*
- *For $m \cdot 2^\ell \ge \log^6 \lambda$, except with probability $O(\ell \cdot e^{-\log^2 \lambda})$, it completes with*

$$\begin{cases} O\big(m \cdot 2^\ell \cdot (\chi + \ell + \log m) + \log^6 \lambda \cdot (\chi + \log^2 \log \lambda) + \chi m \log m\big) & \text{total work, and} \\ O\big(\ell \cdot (\log m + \ell) + \log^3 \log \lambda\big) & \text{depth.} \end{cases}$$

Proof. We first analyze MergeLevels. For small instances, $m \cdot 2^\ell < \log^6 \lambda$, the total cost of MergeLevels is dominated by the non-Las-Vegas variants of **Intersperse**, ORP (only in the case OPRAM_0), and OTM_ℓ.Build , while the depth is dominated by performing **Intersperse** for ℓ times; Hence, by Corollary 4.2, Theorem 3.2, and Fact 11, it takes $O\big(m \cdot 2^\ell \cdot (\chi + (\ell + \log m)^2) + \chi m \log m\big)$ total work and $O\left(\ell \cdot (\ell + \log m)^2\right)$ depth. Otherwise, $m \cdot 2^\ell \ge \log^6 \lambda$, in MergeLevels, the first $6 \log \log \lambda$ levels of non-Las-Vegas **Intersperse** take $O\big(\log^6 \lambda \cdot (\chi + \log^2 \log \lambda) + \chi m \log m\big)$ total work and $O(\log^3 \log \lambda)$ depth; The remaining $O(\ell)$ levels of **Intersperse** and one OTM_ℓ.Build are

both Las Vegas, and thus the analysis is similar to the analysis of Fact 5, where the only difference is taking union bound over the probability that any of such subroutines run longer; Adding up, except with probability $O(\ell \cdot e^{-\log^2 \lambda})$, it takes the claimed total work and depth.

Then, in the second phase, UpdateLevel is constructed identical to the construction of Fact 5, and hence the total work and depth is the same and absorbed by MergeLevels in both cases. □

OPRAM scheme. We first give the high probability statement for the maintain phase.

Fact 13. *Let T be the total steps of the original PRAM (where each step contains m memory concurrent requests). For any λ, except with $O(T \cdot e^{-\log^2 \lambda})$ probability, the maintain phase incurs $T \cdot O\Big(\frac{\log^3 N}{\log\log N} + \frac{\log^2 N \cdot \log^2 \log \lambda + \log N \cdot \log^3 \log \lambda}{\log\log N}\Big)$ parallel steps consuming m CPUs if $N \geq \log^6 \lambda$; otherwise, it incurs $T \cdot O\Big(\frac{\log^4 N}{\log\log N}\Big)$ parallel steps with probability 1.*

Proof. We follow the analysis of Fact 10 but consider N in two cases. If $N < \log^6 \lambda$, then in every OPRAM_d, every level ℓ consist of $m \cdot 2^\ell < \log^6 \lambda$ blocks. By Fact 12, after every 2^ℓ batch of m requests, the reconstruction of all OPRAM_d for $d = 0$ to D takes $O\big(D \cdot m \cdot 2^\ell \cdot (\chi + (\ell + \log m)^2) + D \cdot \chi m \log m\big)$ total work. The depth is $O(\ell \cdot (\ell + \log m)^2)$ as all OPRAM_d are performed **in parallel**. Because the depth is less than the total work divided by m for every ℓ, using m CPUs, it takes $O\big(D \cdot 2^\ell \cdot (\chi + (\ell + \log m)^2) + D \cdot \chi \cdot \log m\big)$ parallel steps for every 2^ℓ batches. During T batch of requests, the above number of parallel steps is counted over every 2^ℓ batches for every ℓ from 0 to $D \log_2 \chi = O(\log N)$. Hence, the total number of parallel steps is

$$T \cdot O\big(D \cdot \big(\log N \cdot (\log^2 m + \chi \log m) + \log^2 N \cdot \log m + \log^3 N\big)\big) = T \cdot O\left(\frac{\log^4 N}{\log\log N}\right)$$

This holds with probability 1.

If $N \geq \log^6 \lambda$, we have levels ℓ from 0 to $6 \log\log \lambda$ falls to the small case and the remaining $O(D \log_2 \chi)$ levels falls to the large case in Fact 12.. During T batch requests, the maintenance on the small levels takes

$$\begin{aligned} &T \cdot O\big(D \cdot \big(\log\log \lambda \cdot (\log^2 m + \chi \log m) + \log^2 \log \lambda \cdot \log m + \log^3 \log \lambda\big)\big) \\ =&T \cdot O\left(\frac{\log N \cdot \log^3 \log \lambda + \log^2 N \cdot \log^2 \log \lambda}{\log\log N}\right) \end{aligned}$$

parallel steps by $m < \log^6 \lambda$ (recall that in Fact 12, a level ℓ is small iff $m \cdot 2^\ell < \log^6 \lambda$, and then we have no small level if $m \geq \log^6 \lambda$). On the $O(D \log_2 \chi)$ large levels, the calculation is calculated similar to the proof of Fact 10, and the only difference is the probability of taking more than claimed parallel steps. Hence, the total number of parallel steps is

$$T \cdot O\left(\frac{\log^3 N}{\log\log N} + \frac{\log^2 N \cdot \log^2 \log \lambda + \log N \cdot \log^3 \log \lambda}{\log\log N}\right).$$

During any sequence of T requests, the Las Vegas subroutines **Intersperse** or ORP are called by the maintenance of OPRAM (indirectly from the position-based OPRAM or one-time memory) for $O(T)$ times and the probability bound holds by taking union bound over Corollary 4.5) and Theorem 3.3, which yields $O(T \cdot e^{-\log^2 \lambda})$.

□

Theorem 7.1. *There exists a perfect OPRAM scheme whose simulation overhead is upper bounded by* $O\left(\frac{\log^3 N}{\log \log N}\right)$ *with probability* $1 - O(T \cdot \exp(-\Omega(\log^2 N)))$ *where* T *denotes the parallel runtime of the original PRAM to be compiled.*

Proof. Choosing the parameter $\lambda = N$, it follows from Lemma 6.1, Facts 9 and 13. □

For any N, by choosing a sufficiently large λ such that $\log^6 \lambda > N$, Fact 13 immediately implies the following theorem. This improves the best known deterministic performance bound of OPRAM, which was previously $O(\log^4 N)$ and constructed by replacing the oblivious random permutation in Chan et al. [CNS18]'s OPRAM with a non-Las-Vegas, parallel algorithm by Alonso and Schott [AS96].

Theorem 7.2. *For any* N*, there exists a perfect OPRAM scheme whose simulation overhead is upper bounded by* $O\left(\frac{\log^4 N}{\log \log N}\right)$ *with probability* 1*.*

References

[ACN+19] Gilad Asharov, Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Locality-preserving oblivious ram. In *Eurocrypt*, 2019.

[Ajt10] Miklós Ajtai. Oblivious rams without cryptographic assumptions. In *STOC*, 2010.

[AKL+] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Oblivious parallel RAM with $O(\log N)$-overhead and depth. Manuscript.

[AKL+20a] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 403–432, Cham, 2020. Springer International Publishing.

[AKL+20b] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Oblivious parallel tight compaction. Cryptology ePrint Archive, Report 2020/125, 2020. `https://eprint.iacr.org/2020/125`.

[AKS83] 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.

[AS96] Laurent Alonso and René Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 159(1):15–28, May 1996.

[BCP16] 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.

[CCS17] T-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In *Asiacrypt*, 2017.

[CGLS17] 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.

[CKN+18] T.-H. Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, pages 158–188, 2018.

[CLP14] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *Asiacrypt*, 2014.

[CLT16] 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.

[CNS18] T-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In *TCC*, 2018.

[CS17] T-H. Hubert Chan and Elaine Shi. Circuit OPRAM: A unifying framework for computationally and statistically secure ORAMs and OPRAMs. In *TCC*, 2017.

[Czu15] Artur Czumaj. Random Permutations Using Switching Networks. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 703–712, New York, NY, USA, 2015. ACM.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.

[DPP18] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *Advances in Cryptology – CRYPTO 2018*, 2018.

[FRY+14] 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.

[GIW16] Daniel Genkin, Yuval Ishai, and Mor Weiss. Binary AMD circuits from secure multiparty computation. In *Theory of Cryptography Conference*, pages 336–366. Springer, 2016.

[GKK+12] 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.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587, 2011.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[Gol87] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[IKO+11] 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.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[LHM+15] 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.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, pages 523–542, 2018.

[LWN+15] 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.

[MLS+13] 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.

[Pes18] Enoch Peserico. Deterministic oblivious distribution (and tight compaction) in linear time. *CoRR*, abs/1807.06719, 2018.

[PPRY18] S. Patel, G. Persiano, M. Raykova, and K. Yeo. Panorama: Oblivious ram with logarithmic overhead. In *IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882, 2018.

[RS19] Michael Raskin and Mark Simkin. Perfectly oblivious ram with small storage overhead. In *Asiacrypt*, 2019.

[RYF+13] 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.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[SvDS+13] 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 *CCS*, 2013.

[WCS15] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS*, 2015.

A Why the KLO Rebalancing Trick Fails for Perfect ORAMs

To asymptotically improve the overhead, one promising idea is to somehow balance the fetch and maintain phases. This idea has been explored in computationally secure ORAMs first by Kushilevitz et al. [KLO12] and later improved in subsequent works [CGLS17]. Kushilevitz et al.'s idea is essentially a reparametrization trick that works for a (computationally secure) hierarchical ORAM.

We will explain Kushilevitz et al.'s trick pretending that the underlying hierarchical ORAM construction is the position-based ORAM scheme in Section 2.1.1 — we then explain why this particular trick is in fact, incompatible with our perfectly secure (position-based) ORAM (even though it worked in the context of earlier computationally secure schemes).

Basically, instead of having $\log_2 N$ many levels of doubling sizes, we now have $L := \log_\chi N + 1$ super-levels numbered $0, 1, \ldots, L-1$ where $\chi > 2$ is called the branching factor. Except for the largest super-level $L-1$, each super-level $\ell < L-1$ contains χ copies of normal levels where each level stores at most χ^ℓ real blocks (and an appropriate number of dummies). The largest level can store N real blocks (and an appropriate number of dummy blocks).

A super-level is *full* iff all levels within it are full. Now for the maintain phase, suppose that super-levels $0, 1, \ldots, \ell^*$ are all full (and suppose that $\ell^* + 1$ is not the largest level), we will merge all super-levels $0, \ldots, \ell^*$ (as well as the most recently fetched block) into an empty level contained within the super-level $\ell^* + 1$. If all super-levels are full, then we merge all super-levels into the largest super-level. For the fetch phase, we need to read a block from every level residing in every super-level.

Assuming that this reparametrization had worked, then the fetch phase would incur $O(\chi \log_\chi N)$ where χ stems from the number of levels within each super-level, and $\log_\chi N$ comes from the number of levels. For the maintain phase, every χ^ℓ accesses we would need to rebuild a level of capacity χ^ℓ. Thus (pretending for the time being that the number of dummies in a level equals the number of real blocks) the amortized cost incurred by the maintain phase would be $\log_\chi N \log_2 N$ where the $\log_2 N$ factor comes from the oblivious sorting and the $\log_\chi N$ factor can be thought of as coming from the number of super-levels. Now observe if we set $\chi = \log N$, both fetch- and maintain-phases would incur $O(\log^2 N / \log\log N)$; thus overall we achieve $\log\log N$ factor improvement over Section 2.1.1.

There is, however, a flaw in the above argument. In this reparametrized scheme, it is not hard to see that a level of capacity χ^ℓ (contained inside a super-level) is accessed $\chi^{\ell+1}$ times before it is rebuilt. To ensure that each location is accessed at most once, the level needs to contain $\chi^{\ell+1}$ number of dummies. Since the number of dummies is χ times larger than the level's capacity (i.e., the number of real blocks the level stores), the actual (amortized) cost of the maintain phase is χ factor greater than our analysis above.

In comparison, in earlier computationally secure ORAM constructions [KLO12, CGLS17], each level is a data structure called an "oblivious hash table" where dummies need not be over-provisioned relative to the number of reals (contrary to our perfectly secure construction in Section 2.1.1); and yet such an "oblivious hash table" can support *unbounded* number of accesses, as long as each real block is requested at most once [KLO12, CGLS17]. This explains why this rebalancing trick worked in the context of earlier computationally secure ORAMs.

B Preliminaries

B.1 Definitions

B.1.1 Parallel Random-Access Machines

We review the concepts of a parallel random-access machine (PRAM) and an oblivious parallel random-access machine (OPRAM). The definitions in this section are borrowed from Chan et al. [CNS18].

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.

We consider a PRAM model where the number of CPUs is fixed to be some parameter m. 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 $\vec{I}^{(t)} := (I_i^{(t)} : i \in [m])$. 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.

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.

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

$$\mathsf{PRAM} := (\Pi, N, T, m),$$

where Π 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 denotes the number of CPUs.

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.

B.1.2 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 $A_t := (\mathsf{addr}_1^t, \mathsf{addr}_2^t, \ldots, \mathsf{addr}_m^t)$ be the list of addresses such that the i-th 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 [CGLS17, CS17] 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 $\mathsf{OPRAM}(\mathsf{inp})$ is identically distributed as $\mathsf{PRAM}(\mathsf{inp})$ for any input inp.

Simulation overhead. We adopt *simulation overhead* as the metric to characterize the overhead of (parallel) oblivious simulation of a 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, 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.