On the Depth of Oblivious Parallel RAM

T.-H. Hubert Chan[1](✉), Kai-Min Chung[2], and Elaine Shi[3]

[1] The University of Hong Kong, Pokfulam, Hong Kong
hubert@cs.hku.hk
[2] Academia Sinica, Taipei, Taiwan
[3] Cornell University, Ithaca, USA

Abstract. Oblivious Parallel RAM (OPRAM), first proposed by Boyle, Chung, and Pass, is the natural parallel extension of Oblivious RAM (ORAM). OPRAM provides a powerful cryptographic building block for hiding the access patterns of programs to sensitive data, while preserving the paralellism inherent in the original program. All prior OPRAM schemes adopt a single metric of "simulation overhead" that characterizes the blowup in parallel runtime, assuming that oblivious simulation is constrained to using the *same* number of CPUs as the original PRAM.

In this paper, we ask whether oblivious simulation of PRAM programs can be further sped up if the OPRAM is allowed to have *more* CPUs than the original PRAM. We thus initiate a study to understand the true depth of OPRAM schemes (i.e., when the OPRAM may have access to unbounded number of CPUs). On the upper bound front, we construct a new OPRAM scheme that gains a logarithmic factor in depth and without incurring extra blowup in total work in comparison with the state-of-the-art OPRAM scheme. On the lower bound side, we demonstrate fundamental limits on the depth any OPRAM scheme—even when the OPRAM is allowed to have an unbounded number of CPUs and blow up total work arbitrarily. We further show that our upper bound result is optimal in depth for a reasonably large parameter regime that is of particular interest in practice.

Keywords: Oblivious parallel RAM · Oblivious RAM · Depth complexity

1 Introduction

Oblivious RAM (ORAM), originally proposed in the seminal works of Goldreich and Ostrovsky [8,9], is a powerful cryptographic building block that allows a program to hide access patterns to sensitive data. Since Goldreich and Ostrovsky's ground-breaking results, numerous subsequent works showed improved ORAM constructions [10,13,18,20,21] with better asymptotics and/or practical performance. ORAM has also been used in various practical and theoretical applications such as multi-party computation [11,22], secure processor design [14,17], and secure storage outsourcing [19,23].

The full version of this paper is available online [3].

T. Takagi and T. Peyrin (Eds.): ASIACRYPT 2017, Part I, LNCS 10624, pp. 567–597, 2017.
https://doi.org/10.1007/978-3-319-70694-8_20

Since most modern computing architectures inherently support parallelism (e.g., cloud compute clusters and modern CPU designs), a natural problem is how to hide sensitive access patterns in such a parallel computing environment. In a recent seminal work, Boyle et al. [1] first propose the notion of Oblivious Parallel RAM (OPRAM), which is a natural extension of ORAM to the parallel setting. Since then, several subsequent works have constructed efficient OPRAM schemes [5,6,15]. One central question in this line of research is whether there is an OPRAM scheme whose *simulation overhead* matches that of the best known ORAM scheme. Specifically, an OPRAM scheme with simulation overhead X means that if the original PRAM consumes m CPUs and runs in parallel time T, then we can obliviously simulate this PRAM also with m CPUs, and in parallel runtime $X \cdot T$. In a recent companion paper called Circuit OPRAM [5], we answered this question in the affirmative. In particular, if N is the number of distinct blocks that the CPUs can request, then Circuit OPRAM proposed a unifying framework where we can obtain statistically secure OPRAMs with $O(\log^2 N)$ simulation overhead, and computationally secure OPRAMs with $(\log^2 N/ \log\log N)$ simulation overhead—thus matching the best known ORAM schemes in both settings [13,21].

All previous OPRAM schemes consider a single performance metric referred to as simulation overhead as mentioned above. It is immediate that an OPRAM scheme with X simulation overhead also immediately implies an ORAM construction with X simulation overhead. Thus, the recent Circuit OPRAM [5] also suggests that we have hit some road-block for constructing more efficient OPRAM schemes—unless we knew how to asymptotically improve the efficiency of sequential ORAM. Note also that in the regime of sufficiently large block sizes, Circuit OPRAM achieves $O(\alpha \log N)$ simulation overhead for any super-constant function α, and this is (almost) tight in light of Goldreich and Ostrovsky's logarithmic ORAM lower bound [8,9].

1.1 Our Results and Contributions

In this paper, we rethink the performance metrics for an OPRAM scheme. We argue that while adopting a single simulation overhead metric is intuitive, this single metric fails to capture the true "work-depth" of the oblivious simulation. In particular, we ask the questions:

1. *If the OPRAM is allowed to access more CPUs than the original PRAM, can we have oblivious simulations with smaller parallel runtime blowup than existing OPRAM schemes?*
2. *Are there any fundamental limits to an OPRAM's work-depth, assuming that the OPRAM can have access to an unbounded number of CPUs?*

To answer the above questions, we turn to the parallel algorithms literature, and adopt two classical metrics, that is, *total work* and *parallel runtime* in the study of OPRAMs. Like the parallel algorithms literature, we also refer to a(n) PRAM/OPRAM's parallel runtime as its *work-depth* (or *depth*). The depth metric represents the runtime of a PRAM when given ample CPUs—thus the depth is the inherently sequential part of a PRAM that cannot be further parallelized even with an arbitrarily large number of CPUs. The depth metric is commonly used in conjunction with total work—since we would like to design low-depth parallel algorithms that do not blow up total work by too much in comparison with the sequential setting (e.g., by repeating computations too many times). Using these classical metrics from the parallel algorithms literature, we can reinterpret the single "simulation overhead" metric adopted by previous OPRAM works as follows: an OPRAM with simulation overhead X has both total work blowup and parallel runtime blowup X in comparison with the original PRAM.

Note that when the OPRAM is constrained to using the same number of CPUs as the original PRAM, its parallel runtime blowup must be at least as large as the total work blowup. In this paper, however, we show that this need not be the case when the OPRAM can access more CPUs than the original PRAM. We design a new OPRAM scheme that gains a logarithmic factor in speed (i.e., depth) in comparison with the state-of-the-art [5] when given logarithmically many more CPUs than the original PRAM. In some sense, our new OPRAM scheme shows that the blowup in total work incurred due to obliviousness can be parallelized further (albeit through non-trivial techniques). Additionally, we prove new lower bounds that shed light on the inherent limits on any OPRAM scheme's depth. In light of our lower bounds, our new OPRAM scheme is optimal in depth for a wide range of parameters. We now present an informal overview of our results and contributions.

Upper Bounds. First, we show that for any PRAM running in time T and consuming W amount of total work, there exists a statistically secure oblivious simulation that consumes logarithmically many more CPUs than the original PRAM, and runs in parallel runtime $O(T \log N \log\log N)$ and total work $O(W \log^2 N)$.

In comparison, the best known (statistically secure) OPRAM scheme incurs both $O(\log^2 N)$ blowup in both total work and parallel runtime (i.e., $O(\log^2 N)$ simulation overhead). In this sense, while preserving the total work blowup, we improve existing OPRAMs' depth by a logarithmic factor.

We then extend our construction to the computationally secure setting by adapting an elegant trick originally proposed by Fletcher et al. [7], and show how to shave another $\log\log N$ factor off both the total work and parallel runtime, assuming that one-way functions exist. Our results are summarized in the following informal theorem.

Theorem 1 (Small-depth OPRAMs: Informal). *The following results are possible for small-depth OPRAMs where N denotes the original PRAM's total memory size, m denotes the original PRAM's number of CPUs, and the security failure must be negligible in N.*

- **Statistically secure, general block size.** *There exists a statistically secure OPRAM that achieves $O(\log^2 N)$ blowup in total work and $O(\log N \log\log N)$ blowup in parallel runtime for general block sizes of $\Omega(\log N)$ bits.*

- **Computationally secure, general block size.** *Assume the existence of one-way functions, then there exists a computationally secure OPRAM that achieves* $O(\frac{\log^2 N}{\log\log N})$ *total work blowup and* $O(\log N)$ *parallel runtime blowup for general block sizes of* $\Omega(\log N)$ *bits.*
- **Statistically secure, large block size.** *For any super-constant function* $\alpha(N) = \omega(1)$*, for any constant* $\epsilon > 0$*, there exists a statistically secure OPRAM that achieves* $O(\alpha \log N \log\log N)$ *total work blowup and* $O(\log m + \log\log N)$ *parallel runtime blowup for blocks of* N^ϵ *bits or larger.*

Lower Bounds. Next, we consider if there are any fundamental limits to an OPRAM scheme's work-depth. We prove a non-trivial lower bound showing that any *online* OPRAM scheme (i.e., with no a-priori knowledge of future requests) that does not perform encoding of data blocks and does not duplicate data blocks too extensively must suffer from at least $\Omega(\log m)$ depth blowup where m is the number of CPUs—and this lower bound holds even when the OPRAM scheme may access arbitrarily many CPUs and have arbitrarily large total work blowup. We stress that our lower bound employs techniques that are different in nature from those of Goldreich and Ostrovsky's classical ORAM lower bound [8,9]—in particular, theirs bounds total work rather than depth. Furthermore, our lower bound holds even for computational security.

Theorem 2 (Lower bound for an OPRAM's depth). *Any computationally or statistically secure online OPRAM scheme must incur at least* $\Omega(\log m)$ *blowup in parallel runtime, as long as the OPRAM (1) does not perform encoding of data blocks (i.e., in the "balls-and-bins" model); and (2) does not make more than* $m^{0.1}$ *copies of each data block.*

We note that the conditions our lower bound assumes (online, balls-and-bins, and bounded duplication) hold for all ORAM and OPRAM constructions.

On the Tightness of Our Upper and Lower Bounds. In light of our lower bound, our OPRAM constructions are optimal in depth in a reasonably large parameter regime. Specifically, our (computationally secure) OPRAM scheme is depth-optimal when $m = N^\epsilon$ for any constant $\epsilon > 0$ for general block sizes. For larger block sizes, our OPRAM scheme is depth-optimal for a larger range of m—in particular, when the block size is sufficiently large, our (statistically secure) OPRAM scheme is tight for m as small as $m = \mathsf{poly}\log N$.

Technical Highlights. Both our lower bounds and upper bounds introduce non-trivial new techniques. Since our lower bound studies the depth of parallel algorithms, it is of a very different nature than Goldreich and Ostrovsky's ORAM lower bounds for total work [8,9]. To prove the depth lower bound, we also depart significantly in technique from Goldreich and Ostrovsky [8,9]. In particular, our lower bound is of an *online* nature and considers the possible batches of requests that a low-depth access pattern can support in a single PRAM step; whereas in comparison, Goldreich and Ostrovksy's lower bound applies even to offline ORAM/OPRAM algorithms, and they perform a counting argument over many steps of the ORAM/OPRAM. The most difficult challenge in proving our lower bound is how to offset the large number of possibilities introduced by "preprocessing", i.e., the number of possible memory configurations before the PRAM step of concern starts. To deal with this challenge, our core idea is to devise a new method of counting that is *agnostic to preprocessing*.

For our new small-depth OPRAM, the main challenge we cope with is of a very different nature from known ORAM and OPRAM works. In particular, all previous ORAMs and OPRAMs that follow the tree-based paradigm [18] adopt a standard recursion technique such that the CPU need not store a large amount of metadata (referred to as the position map). Known schemes treat this recursion as a blackbox technique. Unfortunately, in our work, it turns out that this recursion becomes the main limiting factor to an OPRAM's depth. Thus, we open up the recursion, and our core technique for achieving small-depth OPRAM is to devise a novel offline/online paradigm, such that the online phase that is inherently sequential across recursion levels has small (i.e., $O(\log\log N)$) depth per recursion level; whereas all work that incurs logarithmic depth is performed in an offline phase in parallel across all recursion levels. Designing such an offline/online algorithm incurs several challenges which we explain in Sect. 5.2. We hope that these new techniques can also lend to the design of oblivious parallel algorithms in general.

Another way to view our small-depth OPRAM's contributions is the following. In our setting, we must address two challenges: (1) concurrency, i.e., how to coordinate a batch of m requests such that they can be served simultaneously without causing write conflicts; and (2) parallelism, i.e., how to make each request parallel by using more CPUs. Note that the concurrency aspect is applicable only to OPRAMs where multiple concurrent requests are involved, whereas the parallelism aspect is applicable even for parallelizing the operations of a sequential ORAM. Previous OPRAM constructions [1,6] are concerned only about the former concurrency aspect, but we need to take both into account—in this sense, we are in fact the *first* to investigate the "parallelism" aspect of ORAMs/OPRAMs.[1] In particular, in our fetch phase algorithm, the two aspects are intertwined for the case of general m, in the sense that we cannot separate our techniques into two phases involving one "concurrent compilation" and one "parallel compilation"—such intertwining allows us to construct more efficient algorithms. In the maintain phase, our divide-and-conquer strategy for eviction indeed can be used to parallelize a sequential ORAM.

Related work. Boyle, Chung, and Pass recently initiated the study of Oblivious Parallel RAM (OPRAM) [1]. They were also the first to phrase the simulation overhead metric for OPRAMs, i.e., the parallel runtime blowup of the OPRAM in comparison with the original PRAM, assuming that the OPRAM consumes the same number of CPUs as the original PRAM. Several subsequent

[1] We gratefully acknowledge the Asiacrypt reviewers for pointing out this aspect of our contribution.

works [1,5,6,15] have improved Boyle et al. [1]'s OPRAM construction. Most recently, Chan and Shi [5] show that we can construct statistically secure and computationally secure OPRAMs whose asymptotical performance match the best known sequential ORAM; and their approach is based on the tree-based paradigm [18]. A similar asymptotical result (but for the computationally secure setting only) was also shown by Chan et al. [4] using the hierarchical framework originally proposed by Goldreich and Ostrovsky [8,9]. In the OPRAM context, Goldreich and Ostrovsky's logarithmic lower bound [8,9] immediately implies that any OPRAM with constant blocks of CPU cache must suffer from at least logarithmic *total work* blowup. Thus far there is no other known OPRAM lower bound (and our depth lower bound departs significantly in techniques from Goldreich and Ostrovksy's lower bound).

In the interest of space, we refer the reader to our online full version [3] for additional discussions about the related work.

2 Definitions

2.1 Parallel Random-Access Machines

A *parallel random-access machine* (PRAM) consists of a set of CPUs and a shared memory denoted mem indexed by the address space $[N] := \{1, 2, \ldots, N\}$. In this paper, we refer to each memory word also as a *block*, and we use B to denote the bit-length of each block.

We use m to denote the number of CPUs. In each step t, each CPU executes a next instruction circuit denoted Π, updates its CPU state; and further, CPUs interact with memory through request instructions $\boldsymbol{I}^{(t)} := (I_i^{(t)} : i \in [m])$. Specifically, at time step t, CPU i's instruction is of the form $I_i^{(t)} := (\mathsf{op}, \mathsf{addr}, \mathsf{data})$, where the operation is $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$ performed on the virtual memory block with address addr and block value $\mathsf{data} \in \{0,1\}^B \cup \{\perp\}$. If $\mathsf{op} = \mathtt{read}$, then we have $\mathsf{data} = \perp$ and the CPU issuing the instruction should receive the content of block $\mathsf{mem}[\mathsf{addr}]$ at the initial state of step t. If $\mathsf{op} = \mathtt{write}$, then we have $\mathsf{data} \neq \perp$; in this case, the CPU still receives the initial state of $\mathsf{mem}[\mathsf{addr}]$ in this step, and at the end of step t, the content of virtual memory $\mathsf{mem}[\mathsf{addr}]$ 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 be well-defined. In this paper, we assume the following:

- The original PRAM supports concurrent reads and concurrent writes (CRCW) with an arbitary, 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.

- The compiled, oblivious PRAM (defined below) is a “concurrent read, exclusive write” PRAM (CREW). In other words, the design of our OPRAM construction must ensure that there are no concurrent writes at any time.

We note that a CRCW-PRAM with a parametrizable conflict resolution rule is among the most powerful CRCW-PRAM model, whereas CREW is a much weaker model. Our results are stronger if we allow the underlying PRAM to be more powerful but the compiled OPRAM uses a weaker PRAM model. For a detailed explanation on how stronger PRAM models can emulate weaker ones, we refer the reader to the work by Hagerup [12].

CPU-to-CPU communication. In the remainder of the paper, we sometimes describe our algorithms using CPU-to-CPU communication. For our OPRAM algorithm to be oblivious, the inter-CPU communication pattern must be oblivious too. We stress that such inter-CPU communication can be emulated using shared memory reads and writes. Therefore, when we express our performance metrics, we assume that all inter-CPU communication is implemented with shared memory reads and writes. In this sense, our performance metrics already account for any inter-CPU communication, and there is no need to have separate metrics that characterize inter-CPU communication. In contrast, Chen et al. [6] defines separate metrics for inter-CPU communication.

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

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

where Π denotes the next instruction circuit, N denotes the total memory size (in terms of number of blocks), m denotes the number of CPUs, and T denotes the PRAM's parallel time steps. *Without loss of generality, we assume that* $N \geq m$. We stress that henceforth in the paper, the notations N and m denote the number of memory blocks and the number of CPUs for the original PRAM—our OPRAM construction will consume $O(1)$ factor more memory and possibly more than m CPUs.

2.2 Oblivious Parallel Random-Access Machines

Randomized PRAM. A *randomized PRAM* is a special 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.

Oblivious PRAM (OPRAM). A randomized PRAM parametrized with total memory size N is said to be *statistically oblivious*, iff there exists a negligible function $\epsilon(\cdot)$ such that for any inputs $x_0, x_1 \in \{0,1\}^*$,

$$\mathsf{Addresses}(\mathsf{PRAM}, x_0) \overset{\epsilon(N)}{\equiv} \mathsf{Addresses}(\mathsf{PRAM}, x_1),$$

where $\mathsf{Addresses}(\mathsf{PRAM}, x)$ denotes the joint distribution of memory accesses made by PRAM upon input x and the notation $\overset{\epsilon(N)}{\equiv}$ means the statistical distance is bounded by $\epsilon(N)$. More specifically, for each time step $t \in [T]$, $\mathsf{Addresses}(\mathsf{PRAM}, x)$ includes the memory addresses requested by the CPUs in time step t, as well as whether each memory request is a read or write operation. Henceforth we often use the notation OPRAM to denote a PRAM that satisfies statistical obliviousness.

Similarly, a randomized PRAM parametrized with total memory size N is said to be *computationally oblivious*, iff there exists a negligible function $\epsilon(\cdot)$ such that for any inputs $x_0, x_1 \in \{0,1\}^*$,

$$\mathsf{Addresses}(\mathsf{PRAM}, x_0) \overset{\epsilon(N)}{\equiv_{\mathsf{c}}} \mathsf{Addresses}(\mathsf{PRAM}, x_1)$$

Note the only difference from statistical security is that here the access patterns only need to be indistinguishable to computationally bounded adversaries, denoted by the notaiton $\overset{\epsilon(N)}{\equiv_{\mathsf{c}}}$.

Following the convention of most existing ORAM and OPRAM works [8,9,13,20,21], we will require that the security failure probability to be negligible in the N, i.e., the PRAM's total memory size.

Oblivious simulation. We say that a given OPRAM *simulates* a PRAM if for every input $x \in \{0,1\}^*$, $\Pr[\mathsf{OPRAM}(x) = \mathsf{PRAM}(x)] = 1 - \mu(N)$ where the completeness error μ is a negligible function and the probability is taken over the randomness consumed by the OPRAM—in other words, we require that the OPRAM and PRAM output the same outcome on any input x.

Online OPRAM. In this paper we focus on *online* OPRAM that simulates a PRAM by processing memory request of each PRAM step in an online fashion. Namely, each PRAM memory request is processed by the OPRAM without knowing the future requests. Note that all known ORAM and OPRAM constructions satisfy the online property.

Performance measures. For an online OPRAM simulates a certain PRAM, we measure its performance by its *work-depth* and *total work* overhead. The work-depth overhead is defined to be the number of time steps d for OPRAM to simulate each PRAM step. Let W denote the total number of blocks accessed by OPRAM to simulate a PRAM step. The total work overhead is defined to be W/m, which captures the overhead to simulate a batch of memory request in a PRAM step. Note that both d and W are random variables.

3 Lower Bound on Work-Depth

We show a lower bound on the work-depth in terms of the number m of CPUs. We establish a $\Omega(\log m)$ depth lower bound for OPRAMs satisfying the following properties. We remark that our construction in Sect. 5 as well as all existing ORAM and OPRAM constructions satisfy these properties.

1. **Balls-and-bins storage.** As coined in the ORAM lower bound of Goldreich and Ostrovsky [2], data blocks are modeled as "balls," while shared memory locations and CPU registers are modeled as "bins". In particular, this means that every memory location stores at most one data block and the content of the data block can be retrieved from that location independent of other storage.
2. **Online OPRAM.** As defined in Sect. 2.2, we consider online OPRAM that only learns the logic memory request at the beginning of a PRAM step.
3. ***s*-bounded duplication.** We also need a technical condition on the bound of data duplication. Namely, there is a bound s such that every data block has at most s copies stored on the memory. All known ORAM and OPRAM constructions do not store duplications on the memory[2], i.e., $s = 1$.

It is worth comparing our depth lower bound for OPRAM with the ORAM lower bound of [2]. Both lower bounds assume the balls-and-bins model, but establish lower bound for different metrics and rely on very different arguments (in particular, as we discussed below, counting arguments do not work in our setting). We additionally require online and bounded duplication properties, which are not needed in [2]. On the other hand, our lower bound holds even for OPRAM with *computational security*. In contrast, the lower bound of [2] only holds for statistical security.

The setting for the lower bound. For simplicity, we consider the following setting for establishing the lower bound. First, we consider OPRAM with initialization, where n logical data blocks of the original PRAM are initialized with certain distinct content. This is not essential as we can view the initialization as the first n steps of the PRAM program. We also assume that the logical data size n is sufficiently larger than the total CPUs register size. Specifically, let α be a constant in $(0, 1/3)$ and r be the register size of a CPU. We assume $n \geq \Omega(r \cdot m^{1+(\alpha/4)})$. For any OPRAM satisfying the above three properties with $s \leq m^{(1/3)-\alpha}$, we show that the work-depth is at least $(\alpha/3) \cdot \log m$ with probability at least $1 - m^{-\alpha/4}$ for every PRAM step. In particular, the expected work-depth per step is at least $\Omega(\log m)$ as long as $s \leq m^{1/3-\Omega(1)}$.

Theorem 3 (Lower Bound on Work-Depth). *Let Π be a computationally-secure online OPRAM that satisfies the balls-and-bins model with s-bounded duplication for $s < m^{(1/3)-\alpha}$ for constant $\alpha \in (0, 1/3)$, where the number N of blocks is at least m. Let r be the register size of each CPU. Assume that $n \geq 4r \cdot m^{1+(\alpha/4)}$ and Π has correctness error $\mu \leq m^{-\alpha/4}/4$. Then for each PRAM step t, let $\mathsf{depth}(\Pi, t)$ denote the work-depth of Π for PRAM step t,*

$$\Pr[\mathsf{depth}(\Pi, t) \leq (\alpha/3) \cdot \log m] \leq m^{-\alpha/4},$$

where the probability is over the randomness of the OPRAM compiler Π.

[2] In some hierarchical ORAMs [10,13], there might be several copies of the same block on the server, but only one copy is regarded as *fresh*, while other copies are *stale* and may contain old contents.

Before proving Theorem 3, we first discuss the intuition behind the lower bound proof in Sect. 3.1, where under simplifying assumptions, we reduce the OPRAM lower bound to solving a "user-movie problem" that captures the main argument of our lower bound proof. We discuss how to remove the simplifying assumptions in the end of the section. We then present the formal proof of Theorem 3 in Sect. 3.2

3.1 Intuition: A User-Movie Problem

As a warmup, we first present an intuitive proof making a few simplifying assumptions: (1) the OPRAM compiler must be perfectly correct and perfectly secure; and (2) there is no data block duplication in memory. Later in our formal proofs in Sect. 3.2, these assumptions will be relaxed.

Let us consider how to prove the depth lower bound for a PRAM step t for an OPRAM. Recall that we consider online OPRAM that learn the logical memory requests at the beginning of the step. We can view what happened before the step t as a preprocessing phase that stores the logical memory blocks in different memory locations, and the step t corresponds to an online phase where the CPUs fetch the requested memory blocks with certain observed access pattern. Since the access pattern should hide the logical memory request, any fixed access pattern should allow the CPUs to complete any possible batch of m requests (assuming perfect correctness and perfect security). We say that an access pattern can support a batch of m requests, if there *exists a pre-processing* (i.e., packing of data blocks into memory), such that each CPU can "reach" its desired data block through this access pattern. Our goal is to show that if the access pattern is low depth, then it is impossible to satisfy every batch of m requests—even when one is allowed to enumerate all possible pre-processings to identify one that best matches the requests (given the fixed access pattern). To show this, our argument involves two main steps.

1. First, we show that for any access pattern of low depth, say, d, each CPU can reach at most 2^d memory locations.
2. Second, we show that if an access pattern can satisfy all possible batches of m requests (with possibly different pre-processing), then it must be that some CPU can reach many physical locations in memory.

The former is relatively easy to show. Informally speaking, consider the balls-and-bins model as mentioned earlier: in every PRAM step, each CPU can only access a single memory location (although each memory location can be accessed by many CPUs). This means that at the end of the PRAM step, the block held by each CPU can only be one of two choices: (1) the block previously held by the CPU; or (2) the block in the memory location the CPU just accessed. This means that the access pattern graph must have a small fan-in of 2 (although the fan-out may be unbounded). It is not difficult to formalize this intuition, and show that given any depth-d access pattern, only 2^d memory locations can "flow into" any given CPU. Henceforth, we focus on arguing why the latter is also true—and this requires a much more involved argument.

For ease of understanding, henceforth we shall refer to CPUs as *users*, and refer to data blocks in physical memory as *movies*. There are n distinct movies stored in a database of size N (without duplications) and m users. Each user wants to watch a movie and can access to certain 2^d locations in the database, but the locations the users access to cannot depend on the movies they want to watch. On the other hand, we can decide which location to store each movie to help the users to fetch their movies from the locations they access to. In other words, we first decide which 2^d locations each user access to, then learn which movie each user wants to watch. Then we decide the location to store each movie to help the users to fetch their movies. Is it possible to find a strategy to satisfy all possible movie requests?

We now discuss how to prove the impossibility for the user-movie problem. We first note that a simple counting argument does not work, since there are n^m possible movie requests but roughly $N^n \gg n^m$ possible ways to store the movies in physical memory. To prove the impossibility, we first observe that since we do not allow duplications, when two users request the same movie, they must have access to the same location that stores the movie. Thus, any pair of users must be able to reach a common movie location—henceforth we say that the two users "share" a movie location. This observation alone is not enough, since the users may all share some (dummy) location. If, however, two sets of users request two different movies, then not only must each set share a movie location, the two sets must share two *distinct* locations. More generally, the m users' movie requests induce a *partition* among users where all users requesting the same movie are in the same *part* (i.e., equivalence class), and users in two different parts request different movies. This observation together with carefully chosen partitions allow us to show the existence of a user that needs to access to a large number of locations, which implies an impossibility for the user-movie problem for sufficiently small depth d. We stress that this idea of "partitioning" captures the essence of what pre-processing *cannot* help with, and this explains why our proof works even when there are a large number of possible pre-processings.

Specifically, let $k = m/2$ and label the m users with the set $M := [2] \times [k]$. We consider the following k partitions that partition the users into k pairs. For each $i \in [k]$, we define partition $P_i = \{\{(1, a), (2, a + i)\} : a \in [k]\}$, where the addition is performed modulo k. Note that all k^2 pairs in the k partitions are distinct. By the above observation, for each partition P_i, there are k *distinct* locations $\ell_{i,1}, \ldots, \ell_{i,k} \in [N]$ such that for each pair $\{(1, a), (2, a+i)\}$ for $a \in [k]$, both users $(1, a), (2, a + i)$ access to the location $\ell_{i,a}$. Now, for each location $\ell \in [N]$, let w_ℓ denote the number of $\ell_{i,a} = \ell$ and d_ℓ denote the number of users access to the location ℓ. Note that $w_\ell \leq k$ since user pairs in a partition access to distinct locations (i.e., $\ell_{i,a} \neq \ell_{i,a'}$ for every $i \in [k]$ and $a \neq a' \in [k]$). Also note that $d_\ell \geq \sqrt{2w_\ell}$ since there are only $\binom{d_\ell}{2}$ distinct pairs of users access to the location ℓ.

To summarize, we have (i) $\sum_\ell w_\ell = k^2$, (ii) $w_\ell \leq k$ for all $\ell \in [N]$, and (iii) $d_\ell \geq \sqrt{2w_\ell}$ for all $\ell \in [N]$, which implies $\sum_\ell d_\ell \geq k \cdot \sqrt{2k} = \sqrt{k/2} \cdot m$. Recall that d_ℓ denote the number of users access to the location ℓ and there are m users.

By averaging, there must exist a user who needs to access to at least $\sqrt{k/2}$ locations. Therefore, the user-movie problem is impossible for $d \leq 0.5 \cdot \log m - 2$. Note that the distinctness of the $\ell_{i,a}$'s induced by the partitions plays a crucial role to drive a non-trivial lower bound on the summation $\sum_\ell d_\ell$.

Removing the simplifying assumptions. In above intuitive proof we make several simplyfing assumptions such as perfect security and perfect correctness. We now briefly discuss how to remove these assumptions. The main non-trivial step is to handle computational security, which requires two additional observations. Following the above argument, let us say that an access pattern is compatible with a CPU/user partition if it can support a logic memory request with corresponding induces CPU/user partition.

- First, the above impossibility argument for the user-movie problem can be refined to show that if an access pattern has depth d, then it can be compatible with at most $2^{2(d+1)}$ partitions in $P_1, \ldots, P_k$ defined above.
- Second, whether an access pattern is compatible with a partition can be verified in polynomial time.

Based on these two observations, we show that if $d \leq 0.5 \cdot \log m - 4$ (with noticeable probability), then we can identify two efficiently distinguishable CPU partitions, which implies a depth lower bound for computationally-secure OPRAM. First, we consider the access pattern of partition P_1. Since $d \leq 0.5 \cdot \log m - 4$, it can only be compatible with at most $k/2$ partitions. By an averaging argument, there exists some partition P_i such that P_i is not compatible with the access pattern of P_1 with probability at least $1/2$. On the other hand, by perfect correctness, the access pattern of P_i is always compatible with P_i. Therefore, the access patterns of P_1 and P_i are efficiently distinguishable by an efficient distinguisher D that simply verifies if the access pattern is compatible with P_i.

We now briefly discuss how to remove the remaining assumptions. First, it is not hard to see that the above argument does not require perfect correctness and can tolerate a small correctness error. Second, we make an implicit assumption that the requested data blocks are not stored in the CPU registers so that the CPUs must fetch the requested data blocks from physical locations on the server. This can be handled by considering logic access requests with random logical address and assuming that the logic memory size n is sufficiently larger than the total CPU register size (as in the theorem statement).

We also implicitly assume that we can observe the beginning and end of the access pattern of a PRAM step t. For this, we note that by the online property, we can without loss of generality consider t as the last step so that we know the end of the access pattern for free. Furthermore, we observe that we do not need to know the beginning of the access pattern since the compatibility property is monotone in the following sense. If a partition P_i is compatible with the access pattern of the last d accesses, it is also compatible with the access pattern of the last $d + 1$ accesses. Thus, we can consider the access pattern of the last d accesses for certain appropriately chosen d.

Finally, to handle s-bounded duplication with $s > 1$, we consider CPU partitions where each part is a set of size $s+1$, instead of a pair. By the pigeonhole principle, each part can still certify a pair of CPUs with a shared memory location. However, some extra care is needed for defining the partitions to make sure that different partitions do not certify the same pair of CPUs, and the depth lower bound degrades when s increases. Nevertheless, the lower bound remains $\Omega(\log m)$ for $s \leq m^{1/3-\Omega(1)}$.

3.2 Proof of Theorem 3

We now proceed with a formal proof. We first note that for proving lower bound of the PRAM step t, we can consider PRAM programs where t is the last step, since the behavior of an online OPRAM does not depend on the future PRAM steps. Thus, we can focus on proving lower bound of the last PRAM step. We prove the theorem by contradiction. Suppose that

$$\Pr[\mathsf{depth}(\Pi, t) \leq (\alpha/3) \cdot \log m] > m^{-\alpha/4}, \tag{1}$$

we show two PRAM programs $\mathsf{P}_1, \mathsf{P}_2$ with identical first $t-1$ steps and different logic access request at step t such that the access pattern of $\Pi(\mathsf{P}_1)$ and $\Pi(\mathsf{P}_2)$, which denote the OPRAM simulation of $\mathsf{P}_1, \mathsf{P}_2$ respectively, are efficiently distinguishable. Towards this, we define the CPU partition of a memory request.

Definition 1 (CPU Partition). *Let* $\mathsf{addr} = (\mathsf{addr}_1, \ldots, \mathsf{addr}_m) \in [n]^m$ *be a memory request.* addr *induces a partition* P *on the CPUs, where two CPUs* c_1, c_2 *are in the same part* iff *they request for the same logical address* $\mathsf{addr}_{c_1} = \mathsf{addr}_{c_2}$*. In other words,* P *partitions the CPUs according to the requested logical addresses.*

Recall that s is the bound on the number of duplication. We assume $m = (s+1) \cdot k$ for some prime k. This is without loss of generality, because any integer has a prime number that is within a multiplicative factor of 2. We label the m CPUs with the set $\mathrm{M} := [s+1] \times [k]$. We consider the following set of partitions $P_1, \ldots, P_k$: For $i \in [k]$, the partition $P_i := \{S_i(a) : a \in [k]\}$ is defined such that each part has the form $S_i(a) := \{(b, a+bi) : b \in [s+1]\}$, where addition is performed modulo k. In other words, the parts in the partitions can be viewed as all possible distinct line segments in the $\mathbb{Z}_k^2$ plane.

We will show two programs where their last memory requests have induced partitions P_1 and P_i for some $i \in [k]$ such that their compiled access patterns are efficiently distinguishable. To show this, we model the view of the adversary with an *access pattern graph* and consider a *compatiability* property between an access pattern graph and a CPU partition, defined as follows.

Access pattern graphs and compatibility. Given the access pattern of $\Pi(\mathsf{P})$ for a PRAM program P and a depth parameter $d \in \mathbb{N}$, we define an access pattern graph G as follows.

(a) **Nodes.** The nodes are partitioned into $d + 1$ *layers*. In layer 0, each node represents a physical location in the memory at the beginning of the last d-th time step of $\mathit{\Pi}(\mathsf{P})$.
For $1 \leq i \leq d$, each node in layer i represents a physical location in the memory or a CPU at the end of the last $(d - i + 1)$-st time step.
Hence, we represent each node with (i, u), where i is the layer number and u is either a CPU or a memory location.

(b) **Edges.** Each edge is directed and points from a node in layer $i - 1$ to one in layer i for some $i \geq 1$. For each CPU or a memory location, there is a directed edge from its copy in layer $i - 1$ to one in layer i.
If a CPU c reads from some physical location ℓ in the last $(d-i)$-th time step, then there is a directed edge from $(i-1, \ell)$ to (i, c). Since we allow concurrent read, the out-degree of a node corresponding to a physical location can be unbounded.
If a CPU c writes to some physical location ℓ in the last $(d-i)$-th time step, then there is a directed edge from $(i - 1, c)$ to (i, ℓ).
Observe that since we consider OPRAM with exclusive write, the in-degree of a node (either corresponding to a CPU or a memory location) is at most 2. In fact, the degree 2 bound holds even with concurrent write models as long as the write conflict resolution can be determined only by the access pattern.

The access pattern graph G captures the potential data flow of the last d time steps of the data access. Specifically, a path from $(0, \ell)$ to (d, c) means CPU c may learn the content of the memory location ℓ at the last d time step. If there is no such path, then CPU c cannot learn the content. This motivates the definition of compatible partitions.

Definition 2 (Compatible Partition). *Let G be an access pattern graph and $P_1, \ldots, P_k$ be the partitions defined above. We say $P_i = \{S_i(a) : a \in [k]\}$ is compatible with G if there exist k distinct physical locations $\ell_{i,1}, \ldots, \ell_{i,k}$ on the server such that for each $a \in [k]$, there are at least two CPUs c_1 and c_2 in $S_i(a)$ such that both nodes (d, c_1) and (d, c_2) are reachable from $(0, \ell_{i,a})$ in G.*

Intuitively, compatibility is a necessary condition for the last d time steps of data access to "serve" an access request with induced partition P_i, assuming the requested data blocks are not stored in the CPU registers at the last d-th time step. Recall that each data block has at most s copies in the server, and each part $S_i(a)$ has size $s + 1$. By the Pigeonhole principle, for each part $S_i(a)$ in the induced partition, there must be at least two CPUs $c_1, c_2 \in S_i(a)$ obtaining the logical block from the same physical location ℓ_a on the server, which means the nodes (d, c_1) and (d, c_2) are reachable from $(0, \ell_a)$ in G. We note that verifying compatibility can be done in polynomial time.

Lemma 1 (Verifying Compatibility Takes Polynomial Time). *Given a CPU partition P and an access pattern graph G, it takes polynomial time to verity whether P is compatible with G.*

Proof. Given P and G as in the hypothesis of the lemma, we construct a bipartite graph H as follows. Each vertex in L is labeled with a memory location ℓ, and each vertex in R is labeled with a part S in P. There is an edge connecting a vertex ℓ in L to a vertex S in R *iff* there are at least two CPUs c_1 and c_2 in S such that both (d, c_1) and (d, c_2) are reachable from $(0, \ell)$ in G. This bipartite graph can be constructed in polynomial time.

Observe that P is compatible with G *iff* there is a matching in H such that all vertices's in R are matched. Hence, a maximum matching algorithm can be applied to H to decide if P is compatible with G.

Now, the following key lemma states that an access pattern graph G with small depth d cannot be compatible with too many partitions. We will use the lemma to show two programs with efficiently distinguishable access patterns.

Lemma 2. *Let G be an access pattern graph with the depth parameter d, and $P_1, \ldots, P_k$ be the partitions defined above. Among $P_1, \ldots, P_k$, there are at most $((s+1) \cdot 2^d)^2$ partitions that are compatible with G.*

Proof. Recall that the in-degree of each node is at most 2. Thus, for each node (d, c) in layer d, there are at most 2^d nodes $(0, \ell)$ in layer 0 that can reach the node (d, c). For the sake of contradiction, we show that if G is compatible with $u > ((s+1) \cdot 2^d)^2$ partitions, then there exists a node (d, c) that is reachable by more that 2^d nodes in layer 0.

For convenience, we define a bipartite graph $H = (L, R, E)$ from G as follows. Each vertex in L is labeled with a CPU c, and each vertex in R is labeled with a physical location ℓ of the memory. There is an edge (c, ℓ) in H iff $(0, \ell)$ reaches (d, c) in G. Our goal can be restated as showing that if G is compatible with $u > ((s+1) \cdot 2^d)^2$ partitions, then there exists $c \in L$ with degree $\deg(c) > 2^d$. We do so by lower bounding the number of edges $|E| > m \cdot 2^d$.

By definition, if P_i is compatible with G, then there exist k distinct physical locations $\ell_{i,1}, \ldots, \ell_{i,k}$ on the server such that for each $a \in [k]$, there are at least two CPUs $c_{i,a}, c'_{i,a} \in S_i(a)$ such that $(d, c_{i,a})$ and $(d, c'_{i,a})$ are reachable from $(0, \ell_{i,a})$ in G, which means there are edges $(c_{i,a}, \ell_{i,a})$ and $(c'_{i,a}, \ell_{i,a})$ in H. Thus, a compatible partition certifies $2k$ edges in H, although two different partitions may certify the same edges.

Let $P_{i_1}, \ldots, P_{i_u}$ be the set of compatible partitions. While they may certify the same edges, the set of CPU pairs $\{(c_{i_j,a}, c'_{i_j,a}) : j \in [u], a \in [k]\}$ are distinct for the following reason: Recall that the parts in partitions correspond to different line segments in $\mathbb{Z}_k^2$. Since two points define a line, the fact that the parts correspond to different lines implies that all CPU pairs are distinct.

For each memory location ℓ, let w_ℓ denote the number of $\ell_{i_j,a} = \ell$. It means that ℓ is connected to w_ℓ distinct pairs of CPUs in H, which implies that $\deg(\ell) \geq \sqrt{2w_\ell}$ since there must be at least $\sqrt{2w_\ell}$ distinct CPUs. Also, note that $\sum_\ell w_\ell = u \cdot k$ and $w_\ell \leq u$ for every ℓ since ℓ can appear in each partition at most once. It is not hard to see that the above conditions imply a lower bound on $|E| = \sum_\ell \deg(\ell) \geq k \cdot \sqrt{2u} > m \cdot 2^d$. This in turn implies the existence of $c \in L$ with degree $\deg(c) > 2^d$, a contradiction.

Let us now consider a PRAM program P_1 that performs dummy access in the first $t-1$ steps and a random access request at the t step with induced partition P_1. Specifically, in the first $t-1$ steps, all CPUs read the first logic data block. For the t-th step, let $(b_1, \ldots, b_k)$ be uniformly random k distinct logic data blocks. For $a \in [k]$, the CPUs in part $S_1(a)$ of P_1 read the block b_a at the t-th step. Let $d = (\alpha/3) \cdot \log m$ and $G(\Pi(\mathsf{P}_1), d)$ denote the access pattern graph of $\Pi(\mathsf{P}_1)$ with depth parameter d. The following lemma follows directly by Lemma 2 and an averaging argument.

Lemma 3. *There exists $i^* \in [k]$ such that*

$$\Pr[P_{i^*} \textit{ is compatible with } G(\Pi(\mathsf{P}_1), d)] \leq ((s+1) \cdot 2^d)^2/k \leq m^{-\alpha/3},$$

where the randomness is over Π and P_1.

Now, consider a PRAM program P_2 that is identical to P_1, except that the access request at the t-th step has induced partition P_{i^*} instead of P_1. Namely, for $a \in [k]$, the CPUs in part $S_{i^*}(a)$ of P_{i^*} read the block b_a at the t-th step, where $(b_1, \ldots, b_k)$ are uniformly random k distinct logic data blocks.

Lemma 4. *Suppose that Π satisfies Eq. (1), then*

$$\Pr[P_{i^*} \textit{ is compatible with } G(\Pi(\mathsf{P}_2), d)] > m^{-\alpha/4}/4,$$

where the randomness is over Π and P_2.

Proof. First note that since each CPU request a random data block at the t-th PRAM step, the probability that the requested data block is stored in the CPU register is at most r/n. By a union bound, with probability at least $1 - m \cdot (r/m) \geq 1 - m^{-\alpha/4}/4$, all data blocks requested at the t-th PRAM step are not in the corresponding CPU registers. In this case, the CPUs need to obtain the data blocks from the server. Furthermore, if the work-depth of the t-th PRAM step is $\leq d$, then the CPUs need to obtain the data blocks in the last d time steps of data access, which as argued above, implies compatibility. Therefore,

$$\Pr[P_{i^*} \text{ is compatible with } G(\Pi(\mathsf{P}_2), d)] > m^{-\alpha/4} - m^{-\alpha/4}/4 - \epsilon_c > m^{-\alpha/4}/4.$$

Recall by Lemma 1 that compatibility can be checked in polynomial time. The above two lemmas imply that assuming Eq. (1), $\Pi(\mathsf{P}_1)$ and $\Pi(\mathsf{P}_2)$ are efficiently distinguishable by a distinguisher D who checks the compatibility of P_{i^*} and the access pattern graph with depth parameter $d = (\alpha/3) \cdot \log m$. This is a contradiction and completes the proof of Theorem 3.

4 Background on Circuit OPRAM and Building Blocks

4.1 Preliminaries: Circuit OPRAM

As a warmup, we first briefly review the recent Circuit OPRAM algorithm [5] that we build on top of. For clarity, we make a few simplifying assumptions in this overview:

- We explain the non-recursive version of the algorithm where we assume that the CPU can store a position map for free that tracks the rough physical location of every block: this CPU-side position map is later removed using a standard recursion technique in Circuit OPRAM [5]—however, as we point out later, to obtain a small depth OPRAM in our paper, we must implement the recursion differently and thus in our paper we can no longer treat the recursion as blackbox technique.
- We assume that m is not too small and is at least polylogarithmic in N; and
- A standard conflict resolution procedure proposed by Boyle et al. [1] has been executed such that the incoming batch of m requests are for distinct real blocks (or dummy requests).

Core data structure: a pool and $2m$ *subtrees.* Circuit ORAM partitions the ORAM data structure in memory into $2m$ *disjoint subtrees.* Given a batch of m memory requests (from m CPUs), each request will be served from a random subtree. On average, each subtree must serve $O(1)$ requests in a batch; and due to a simple balls and bins argument, except with negligible probability, even the worst-case subtree serves only $O(\alpha \log N)$ incoming requests for any super-constant function α.

In addition to the $2m$ subtree, Circuit OPRAM also maintains an overflow pool that stores overflowing data blocks that fail to be evicted back into the $2m$ subtrees at the end of each batch of m requests.

It will help the reader to equivalently think of the $2m$ subtrees and the pool in the following manner: First, think of a single big Circuit ORAM [21] tree (similar to other tree-based ORAMs [18]). Next, identify a height with $2m$ buckets, which naturally gives us $2m$ *disjoint subtrees.* All buckets from smaller heights as well as the Circuit ORAM's stash form the *pool.* As proven in the earlier work [5], at any time, the pool contains at most $O(m + \alpha \log N)$ blocks.

Fetch. Given a batch of m memory requests, henceforth without loss of generality, we assume that the m requests are for distinct addresses. This is because we can adopt the conflict resolution algorithm by Boyle et al. [1] to suppress duplicates, and after data has been fetched, rely on oblivious routing to send fetched data to all request CPUs. Now, look up the requested blocks in two places, both the pool and the subtrees:

- *Subtree lookup:* For a batch of m requests, each request comes with a position label—and all m position labels define m random paths in the $2m$ subtrees. We can now fetch from the m path in parallel. Since each path is $O(\log N)$ in length, each fetch can be completed in $O(\log \log N)$ parallel steps with the help of $\log N$ CPUs.

  All fetched blocks are merged into the pool. Notice that at this moment, the pool size has grown by a constant factor, but later in a cleanup step, we will compress the pool back to its original size. Also, at this moment, we have not removed the requested blocks from the subtrees yet, and we will remove them later in the maintain phase.

- *Pool lookup:* At this moment, all requested blocks must be in the pool. Assuming that m is not too small, we can now rely on oblivious routing to route blocks back to each requesting CPU—and this can be completed in $O(\log m)$ parallel steps with m CPUs.

Maintain. In the maintain phase, perform the following: (1) remove all blocks fetched from the paths read; and (2) perform eviction on each subtree.

- *Efficient simultaneous removals.* After reading each subtree, we need to remove up to $\mu := O(\alpha \log N)$ blocks that are fetched. Such removal operations can lead to write contention when done in parallel: since the paths read by different CPUs overlap, up to $\mu := O(\alpha \log N)$ CPUs may try to write to the same location in the subtree. Circuit OPRAM employs a novel *simultaneous removal* algorithm to perform such removal in $O(\log N)$ parallel time with m CPUs. We refer the reader to the Circuit OPRAM paper for an exposition of the simultaneous removal algorithm. As noted in the Circuit OPRAM paper [5], simulatenous removal from m fetch paths can be accomplished in $O(\log m + \log \log N)$ parallel steps with $O(m \cdot \log N)$ total work.
- *Selection of eviction candidates and pool-to-subtree routing.* At this moment, we will select exactly one eviction candidate from the pool for each subtree. If there exists one or more blocks in the pool to be evicted to a certain subtree, then the *deepest* block (where deepest is precisely defined in Circuit ORAM [21]) with respect to the current eviction path will be chosen. Otherwise, a dummy block will be chosen for this subtree. Roughly speaking, using the above criterion as a preference rule, we can rely on oblivious routing to route the selected eviction candidate from the pool to each subtree. This can be accomplished in $O(\log m)$ parallel steps with m CPUs assuming that m is not too small.
- *Eviction.* Now, each subtree performs exactly 1 eviction. This can be accomplished in $O(\log N)$ runtime using the sequential procedure described in the original Circuit ORAM paper [21]. At the end of this step, each subtree will output an eviction leftover block: the leftover block is dummy if the chosen eviction candidate was successfully evicted into the subtree (or if the eviction candidate was dummy to start with); otherwise the leftover block is the orginal eviction candidate. All these eviction leftovers will be merged back into the central pool.
- *Pool cleanup.* Notice that in the process of serving a batch of requests, the pool size has grown—however, blocks that have entered the pool may be dummy. In particular, we shall prove that the pool's occupancy will never exceed $c \cdot m + \alpha \log N$ for an appropriate constant c except with $\mathsf{negl}(N)$ probability. Therefore, at the end of the maintain phase, we must compress the pool back to $c \cdot m + \alpha \log N$. Such compression can easily be achieved through oblivious sorting in $O(\log m)$ parallel steps with m CPUs, assuming that m is not too small.

Recursion. Thus far, we have assumed that the position map is stored on the CPU-side, such that the CPU knows where every block is in physical memory.

To get rid of the position map, Circuit OPRAM employs a standard recursion technique that comes with the tree-based ORAM/OPRAM framework [18]. At a high level, the idea of the recursion framework is very simple: instead of storing the position map on the CPU side, we recurse and store the position map in a smaller OPRAM in physical memory; and then we recurse again and store the position map of this smaller OPRAM in a yet smaller OPRAM in physical memory, and so on. If each block can store $\gamma > 1$ number of position labels, then every time we recurse, the OPRAM's size reduces by a factor of γ. Thus in at most $\log N$ recursion levels, the metadata size becomes at most $O(1)$ blocks—and at this moment, the CPU can store all the metadata locally in cache.

Although most prior tree-based ORAM/OPRAM papers typically treat this recursion as a standard, blackbox technique, in this paper we cannot—on the contrary, it turns out that the recursion becomes the most non-trivial part of our low-depth OPRAM algorithm. Thus, henceforth the reader will need to think of the recursion in an expanded form—we now explain what exactly happens in the recursion in an expanded form. Imagine that one of the memory requests among the batch of m requests asks for the logical address $(0101100)_2$ in binary format, and suppose that each block can store 2 position labels. Henceforth we focus on what happens for fetching this logical address $(0101100)_2$—but please keep in mind that there are m such addresses and thus the following process is repeated m times in parallel.

- First, the 0th recursion level (of constant size) will tell the 1st recursion level the position label for the address $(0*)_2$.
- Next, the 1st recursion level fetch the metadata block at level-1 address $(0*)_2$ and this fetched block contains the position labels for $(00*)_2$ and $(01*)_2$.
- Now, level-1 informs level-2 of the position label for $(01*)_2$; at this moment, level-2 fetches the metadata block for the level-2 address $(01*)_2$ and this fetched block contains the position labels for the addresses $(010*)_2$ and $(011*)_2$; and so on.
- This continues until the D-th recursion level (i.e., the final recursion level)—this final recursion level stores actual data blocks rather than metadata, and thus the desired data block will be fetched at the end.

As mentioned, the above steps are in fact replicated m times in parallel since there are m requests in a batch. This introduces a couple additional subtleties:

- First, notice that for obliviousness, conflict resolution must be performed upfront for each recursion level before the above procedure starts—this step can be parallelized across all recursion levels.
- Second, how do the m fetch CPUs at one recursion level *obliviously* route the fetched position labels to the m fetch CPUs waiting in the next recursion level? Circuit OPRAM relies on a standard oblivious routing procedure (initially described by Boyle et al. [1]) for this purpose, thus completely hiding which CPUs route to which.

Important observation. At this moment, we make an important observation. In the Circuit OPRAM algorithm, the fetch phase operations are inherently sequential across all recursion levels, and the maintain phase operations can be parallelized across all recursion levels. In particular, during the fetch phase, the m fetch CPUs at recursion level d must block waiting for recursion level $d-1$ to pass down the fetched position labels before its own operations can begin. Due to the sequential nature of the fetch phase, Circuit OPRAM incurs at least $(\log m + \log\log N)\log N$ depth, where the $\log m$ stems from level-to-level oblivious routing, $\log\log N$ stems the depth needed to parallel-fetch from a path of length $\log N$ (and other operations), and the $\log N$ factor is due to the number of recursion levels. In comparison, the depth of the maintain phase is not the limiting factor due to the ability to perform the operations in parallel across recursion levels.

4.2 Other Important Building Blocks

Permutation-related building blocks. We will rely on the following building blocks related to generating and applying permutations. In the interest of this space, we describe the abstractions of the building blocks but defer their full specification to our online full version [3].

1. **Apply a pre-determined permutation to an array.** It is not difficult to see that we can in parallel apply a pre-determined permutation to an array in a single parallel step (see our online full version [3] for the detailed algorithm).
2. **Permute an array by a secret random permutation.** One can generate a secret random permutation and apply it to an array obliviously, without revealing any information about the permutation—and this can be accomplished in $O(\log n)$ depth and $O(n \log n)$ work for an array of size n. The formal specification and proofs are deferred to the online full version [3].
3. **Obliviously construct a routing permutation that permutes a source to a destination array.** In our online full version [3] we show how to accomplish the following task: given a source array snd of length k containing distinct real elements and dummies (where each dummy element contains unique identifying information as well), and a destination array rcv also of length k containing distinct real elements and dummies, with the guarantee that the set of real elements in snd are the same as the set of real elements in rcv. Now, construct a routing permutation $\pi : [k] \to [k]$ (in an oblivious manner) such that for all $i \in [k]$, if $\mathsf{snd}[i]$ contains a real element, then $\mathsf{rcv}[\pi[i]] = \mathsf{snd}[i]$. This can be accomplished in $O(n \log n)$ work and $O(\log n)$ depth by calling oblivious sort $O(1)$ number of times.

Oblivious bin-packing. Oblivious bin-packing is the following primitive.

- *Inputs:* Let B denote the number of bins, and let Z denote the target bin capacity. We are given an input array denoted In, where each element is either a dummy denoted $\perp$ or a real element that is tagged with a bin number

$g \in [B]$. It is guaranteed that there are at most Z elements destined for each bin.
- *Outputs:* An array $\mathsf{Out}[1 : BZ]$ of length $B \cdot Z$ containing real and dummy elements, such that $\mathsf{Out}[(g-1)B+1 : gB]$ denotes contents of the g-th bin for $g \in [B]$. The output array Out must guarantee that the g-th bin contains all elements in the input array In tagged with the bin number g; and that all real elements in bin g must appear in the input array In and are tagged with g.

There is an oblivious parallel algorithm that accomplishes oblivious bin packing in total work $O(\widetilde{n} \log \widetilde{n})$ and parallel runtime $O(\log \widetilde{n})$ where $\widetilde{n} = \max(|\mathsf{In}|, B \cdot Z)$. The algorithm works as follows:

1. For each group $g \in [B]$, append Z filler elements of the form $(\mathtt{filler}, g)$ to the resulting array—these filler elements ensure that every group will receive at least Z elements after the next step.
2. Obliviously sort the resulting array by the group number, placing all dummies at the end. When elements have the same group number, place filler elements after real elements.
3. By invoking an instance of the oblivious aggregation algorithm [1,16] (see full version [3] for the definition of oblivious aggretation), each element in the array finds the leftmost element in its own group. Now for each element in the array, if its offset within its own group is greater than Z, replace the element with a dummy $\perp$.
4. Oblivious sort the resulting array placing all dummies at the end. Truncate the resulting array and preserve only first $B \cdot Z$ blocks.
5. For every filler element in the resulting array, replace it with a dummy.

5 A Small-Depth OPRAM: Level-to-Level Routing Algorithm

5.1 Overview of Our OPRAM

We now show how we can improve the depth of OPRAM schemes [1] by a logarithmic factor, through employing the help of more CPUs; and importantly, we achieve this without incurring extra total work in comparison with the best known OPRAM scheme [5].

Challenges. As argued earlier in Sect. 4.1, for the case of general block sizes, the most sequential part of the Circuit OPRAM algorithm stems from the (up to) $\log N$ recursion levels. More specifically, (apart from the final data level), each recursion level's job is to fetch the metadata (referred to as position labels) necessary, and route this information to the next recursion level. In this way, the next recursion level will know where in physical memory to look for the metadata needed by its next recursion level, and so on (we refer the reader to Sect. 4.1 for a more detailed exposition of the recursion).

Thus, the fetch phase operations of Circuit OPRAM are inherently sequential among the D recursion levels, incurring $(D(\log m + \log\log N))$ in depth, where the $\log m$ term stems from the level-to-level oblivious routing of fetched metadata, and the $\log\log N$ term stems from fetching metadata blocks from a path of length $\log N$. Igoring the $\log\log N$ term, our goal therefore is to get rid of the $\log m$ depth that stems from level-to-level oblivious routing.

Our result. Our main contribution is to devise a low-depth algorithm to perform level-to-level routing of metadata. At first sight, this task seems unlikely to be successful—since each recursion level must *obliviously* route its metadata to the next level, it would seem like we are inherently subject to the depth necessary for an oblivious routing algorithm [1]. Since oblivious routing in some sense implies oblivious sorting, it would seem like we have to devise an oblivious sorting algorithm of less than logarthmic depth to succeed in our goal.

Perhaps somewhat surprisingly, we show that this need not be the case. In particular, we show that by (1) allowing a negligible statistical failure probability; (2) exploiting special structures of our routing problem; and (3) introducing an offline/online paradigm for designing parallel oblivious algorithms, we can devise a special-purpose level-to-level oblivious routing algorithm such that

1. all work that is inherently $\log m$ in depth is pushed to an offline phase that can be parallelized across all recursion levels; and
2. during the online phase that is inherently sequential among all $\log N$ recursion levels, we can limit the work-depth of each recursion level to only $\log\log N$ rather than $\log m$—note that for most interesting parameter regimes that we care about, $\log m \gg \log\log N$.

We defer the detailed introduction of this algorithm and its proofs to Sect. 5.2. As a result, we obtain a new, *statistically* secure OPRAM algorithm (for general block sizes) that achieves $O(\log N \log\log N)$ depth blowup and $O(\log^2 N)$ total work blowup. In comparison, under our new performance metrics, the best known OPRAM algorithm [5] achieves $O(\log^2 N)$ total work blowup and $O(\log^2 N)$ depth blowup. Thus we achieve a logarithmic factor improvement in terms of depth.

Extensions. We consider several extensions. First, using a standard technique described by Fletcher et al. [7] and extended to the OPRAM setting by Chan and Shi [5], we show how to obtain a *computationally* secure OPRAM scheme with $O(\log^2 N/\log\log N)$ total work blowup and $O(\log N)$ depth blowup, and supporting general block sizes. In light of our aforementioned OPRAM depth lower bound (which also applies to computationally secure OPRAMs), our OPRAM scheme is optimal for $m = N^\epsilon$ where $\epsilon > 0$ is an arbitrarily small constant.

Finally, we consider a setting with sufficiently large blocks, say, the block size is N^ϵ for any constant $\epsilon > 0$—in this case, the recursion depth becomes $O(1)$. In this case, the limiting factor to an OPRAM's work depth now is the eviction algorithm (rather than the level-to-level routing). We show how to leverage a non-trivial devise and conquer technique to devise a new, small-depth eviction algorithm, allowing us to perform eviction along a path of length $\log N$ in $\log \log N$ depth rather than $\log N$—however, this is achieved at the cost of a small $\log \log N$ blowup in total work. As a result, we show that for sufficiently large blocks, there is an OPRAM scheme with depth as small as $O(\log \log N + \log m)$ where the $\log \log N$ part arises from our low-depth eviction algorithm (and other operations), and the $\log m$ part arises from the conflict resolution and oblivious routing of fetched data back to requesting CPUs—thus tightly matching our depth lower bound as long as m is at least logarithmic in N.

5.2 Small-Depth Routing of Position Identifiers: Intuition

Problem statement. As we explained earlier, in each recursion level, m fetch CPUs fetch the metadata (i.e., position labels) required for the next recursion level. The next recursion level contains m fetch CPUs waiting to receive these position labels, before its own operations can begin. Circuit OPRAM performs such level-to-level routing using a standard oblivious routing building block, thus incurring at least $D \log m$ depth where D is the number of recursion levels which can be as large as $\log N$, and $\log m$ is the depth of standard oblivious routing. How can we reduce the depth necessary for level-to-level routing?

We will first clarify some details of the problem setup. Recall that in each PRAM step, we receive a batch of m memory requests, i.e., m logical addresses. Given these m logical addresses, we immediately know which level-d addresses to fetch for each recursion level d (see Sect. 4.1 for details). We assume that conflict resolution has been performed for each recursion level d on all of the m level-d addresses, and thus, every real (i.e., non-dummy) level-d address is distinct. Now, note that from all these level-d addresses (and even without fetching the actual metadata in each recursion level), we can already determine the routing topology from level to level: as an example, a level-2 CPU that needs to fetch the level-2 address $(010*)$ would like to receive position labels from the level-1 fetch CPU with the address $(01*)$.

Our goal here is to improve the OPRAM's depth to $O(\log N \log \log N)$ for general (worst-case) block sizes. We use the parameter Γ to denote the number of position labels that a block can store; we let $\gamma := \min\{\Gamma, m\}$ be an upper bound on the number of position labels in a block that is "useful" for the next recursion level. To achieve this, in the part of the algorithm that is sequential among all recursion levels (henceforth also referred to as the *online* part), we can only afford $O(\log \log N)$ depth rather than the $\log m$ necessary for oblivious routing. Indeed, for a general oblivious routing problem consisting of m senders and m receivers, it appears the best one can do is to rely on an oblivious routing network [1,6] that has $\log m$ depth—so how can we do better here? We rely on two crucial insights:

1. First, we observe that our routing problem has *small fan-in and fan-out*: each sender has at most γ recipients; and each recipient wants to receive from at most 1 sender. This is because that each fetched metadata block contains at

most γ position labels, and obviously each fetch CPU in the next level only needs one position label to proceed.

2. Second, we will rely on an *offline-online paradigm*—in the offline phase, we are allowed to perform preparation work that indeed costs $\log m$ depth; however, in the online phase, the depth is kept to be small. Later when we employ this offline/online oblivious routing building block in our full OPRAM algorithm, we will show that the offline phase does not depend on any fetched data, and thus can be paralellized across all recursion levels, whereas the online phase must still be sequential—but recall that the online phase has much smaller depth.

First insight: localized routing. Our first idea is to rely on this observation to restrict oblivious routing to happen only within small groups—as we shall explain later, for this idea to work, it is essential that our routing problem has small fan-in and fan-out. More specifically, we would like that each small group of senders talk to a corresponding small group of receivers, say, sender group S_i talks only to receiver group R_i, where both S_i and R_i are $\mu := \alpha\gamma^2 \log N$ in size, where the choice of μ is due to Lemma 5. If we do this, then oblivious routing within each small group costs only $\log \mu$ depth.

How can we arrange senders and receivers into such small groups? For correctness we must guarantee that for every i, each receiver in R_i will be able to obtain its desired item from some sender in S_i.

To achieve this, we rely on a randomized load balancing approach. The idea is very simple. First, we pad the sender array with dummy senders to a size of $2m$—recall that there are at most m real senders. Similarly, we pad the receiver array to a size of $2m$ as well. Henceforth if a receiver wants an item from a sender, we say that the sender and receiver are connected. Every dummy sender is obviously connected to 0 receivers.

Now, if we pick a random sender from the sender array, in expectation this sender will be connected to 0.5 receivers. Thus a random subset of μ senders will in expectation is connected to $0.5\,\mu$ receivers—using measure concentration techniques, it is not difficult to show that a random subset of μ senders is connected to μ receivers except with negligible probability—note that this measure concentration result holds only when our routing problem has small fan-in and fan-out (see Lemma 5 for details).

Our idea is to randomly permute the source array, and have the first μ sender be group 1, the second μ senders be group 2, and so on. By relying on $O(1)$ number of oblivious sorts, we can now arrange the receiver array to be "loosely aligned" with the sender array, i.e., all receivers connected to sender group 1 are in the first size-μ bucket of the receiver array, all receivers connected to sender group 2 are in the second size-μ bucket of the receiver array, and so on.

Using the above idea, the good news is that oblivious routing is now constrained to μ-sized groups (each containing γ addresses), thus costing only $\log \mu$ depth. However, our above algorithm still involves randomly permuting the sender array and oblivious routing to loosely align the receiver array with the sender array—these steps cost $\log m$ depth. Thus our idea is to perform these steps in an offline phase that can be parallelized across all recursion levels, and thus the depth does not blow up by the number of recursion levels. Nonetheless how to instantiate this offline/online idea is non-trivial as we explain below.

Second insight: online/offline paradigm. One challenge that arises is how to coordinate among all recursion levels. To help the reader understand the problem, let us first describe what would have happened if everything were performed online, sequentially level by level:

Imagine that each recursion has $2m$ fetch CPUs (among which at most m are real) first acting as receivers. Once these receivers have received the position labels, they will fetch data from the OPRAM's tree data structure. At this point, they hold the position labels desired by the next recursion level, and thus the receivers now switch roles and become senders with resepct to the next recursion level. Before the receivers become senders, it is important that they be randomly permuted for our earlier load balancing technique to work. Now, we can go ahead and prepare the next recursion level's receivers to be loosely aligned with the permuted senders, and proceed with the localized oblivious routing.

Now let us consider how to divide this algorithm into a parallel offline phase and a subsequent low-depth online phase. Clearly, the oblivious routing necessary for loosely aligning each recursion level's receivers with the last level's senders must be performed in the offline phase—and we must paralellize this step among all recursion levels. Thus, our idea is the following:

- First, for each recursion level d in parallel, we randomly permutate level d's fetch CPUs in an oblivious fashion (using a building block called oblivious random permutation), at the end of which we have specified the configuration of level d's sender array (that is, after level d's fetch CPUs switch roles and become senders).
- At this point, each recursion level d can prepare its receiver array based on the configuration of level $(d-1)$'s sender array. This can be done in parallel too.
- During the online phase, after fetching metadata from the OPRAM tree, the receivers must permute themselves to switch role to senders—since the offline stage has already dictated the sender array's configuration, this permutation step must respect the offline stage's decision.

  To achieve this in small online depth, our idea is that during the offline phase, each recursion level relies on an instance of oblivious routing to figure out exactly what permutation to apply (henceforth called the "routing permutation") to switch the receiver array to the sender array's configuration—and this can be done in parallel among all recursion levels once a level's receiver and sender arrays have both been determined. Once the offline stage has written down this routing permutation, in the online stage, the receivers can simply apply the permutation, i.e., each receiver writes itself to some array location as specified by the permutation that offline stage has written down. Applying the permutation online takes a single parallel step.

One observation is that during the online stage, the routing permutation is revealed in the clear. To see why this does not leak information, it suffices to see that the result of this routing permutation, i.e., the sender array, was obliviously randomly permuted to start with (using a building block called oblivious random permutation). Thus, even conditioned on having observed the oblivious random permutation's access patterns, each permutation is still equally likely—and thus the routing permutation that is revealed is indistinguishable from a random permutation (even when conditioned on having observed the oblivious random permutation's access patterns).

5.3 Core Subroutine: Localized Routing

Notations and informal explanation. In the OPRAM's execution, the instructions waiting to receive position labels at a recursion level d is denoted $\mathsf{Instr}^{\langle d\rangle}$. $\mathsf{Instr}^{\langle d\rangle}$ has been obliviously and randomly permuted in the offline phase. When these incomplete instructions have received position labels, they become complete and are now called $\mathsf{CInstr}^{\langle d\rangle}$ where $\mathsf{CInstr}^{\langle d\rangle}$ and $\mathsf{Instr}^{\langle d\rangle}$ are arranged in the same order. When data blocks are fetched in recursion level d, they are called $\mathsf{Fetched}^{\langle d\rangle}$, and $\mathsf{Fetched}^{\langle d\rangle}$ has the same order as $\mathsf{CInstr}^{\langle d\rangle}$. In the offline phase, $\mathsf{Instr}^{\langle d\rangle}$ is obliviously sorted to be loosely aligned with $\mathsf{Fetched}^{\langle d-1\rangle}$ resulting in $\overline{\mathsf{Instr}}^{\langle d\rangle}$, such that $\overline{\mathsf{Instr}}^{\langle d\rangle}$ can receive position labels from $\mathsf{Fetched}^{\langle d-1\rangle}$ through localized oblivious routing. The offline phase also prepares a routing permutation $\pi^{d\to d+1}$, that will permute $\overline{\mathsf{Instr}}^{\langle d\rangle}$ (after having received position labels) back to $\mathsf{CInstr}^{\langle d\rangle}$—and the online phase will apply this routing permutation $\pi^{d\to d+1}$ in a single parallel step. We now describe our algorithms more formally.

We consider the following problem where there is a source array and a destination array, and the destination array wants to receive position identifiers from the source. Specifically, the source array is a set of fetched blocks in randomly permuted order, where each block may contain up to γ position labels corresponding to γ addresses in the next recursion level. The destination array is an incomplete instruction array where each element contains the address of the block to be read at the next recursion level—and each address must receive its corresponding position label before the fetch operations at the next recursion level can be invoked.

- *Inputs:* The inputs contain a randomly permuted source array $\mathsf{Fetched}^{\langle d\rangle}$ that represent the fetched position identifier blocks at recursion level d, and a randomly permuted destination array $\mathsf{Instr}^{\langle d+1\rangle}$ which represents the incomplete instruction array at recursion level $d+1$.
  - The source array $\mathsf{Fetched}^{\langle d\rangle}$ contains $2m$ blocks, each of which contains up to γ (logical) pairs of the form $(\mathsf{addr}, \mathsf{pos})$ that are needed in the next recursion level. All the γ addresses in the same block comes from Γ contiguous addresses, and thus in reality the address storage is actually compressed—however, we think of each block in $\mathsf{Fetched}^{\langle d\rangle}$ as *logically* containing pairs of the form $(\mathsf{addr}, \mathsf{pos})$.

- The destination array $\mathsf{Instr}^{\langle d+1\rangle}$ contains m elements each of which is of the form $(\mathsf{addr}, \_)$, where "_" denotes a placeholder for receiving the position identifier for addr later. This array $\mathsf{Instr}^{\langle d+1\rangle}$ is also referred to as the incomplete instruction array.
- We assume that
  (1) all addresses in the destination array must occur in the source array;
  (2) the γ addresses contained in the same block come from Γ contiguous addresses; and
  (3) both the source array $\mathsf{Fetched}^{\langle d\rangle}$ and the destination array $\mathsf{Instr}^{\langle d+1\rangle}$ have been randomly permuted.

– *Outputs:* A complete instruction array denoted CInstr of length $2m$ where $\mathsf{CInstr}^{\langle d+1\rangle}[i]$ is of the form $(\mathsf{addr}_i, \mathsf{pos}_i)$ such that
  - $\mathsf{Instr}^{\langle d+1\rangle}[i] = (\mathsf{addr}_i, \_)$, i.e., the sequence of addresses contained in the output $\mathsf{CInstr}^{\langle d+1\rangle}$ agree with those contained in the input $\mathsf{Instr}^{\langle d+1\rangle}$; and
  - The tuple $(\mathsf{addr}_i, \mathsf{pos}_i)$ exists in some block in $\mathsf{Fetched}^{\langle d\rangle}$, i.e., the position identifier addr_i receives is correct (as defined by $\mathsf{Fetched}^{\langle d\rangle}$).

Offline phase. The inputs are the same as the above. In the offline phase, we aim to output the following arrays:

(a) A permuted destination array $\overline{\mathsf{Instr}}^{\langle d+1\rangle}$ that is a permutation of $\mathsf{Instr}^{\langle d+1\rangle}$ such that it is *somewhat aligned* with the source $\mathsf{Fetched}^{\langle d\rangle}$, where *somewhat aligned* means the following:

[Somewhat aligned:] Fix $\alpha := \omega(1)$ to be any super-constant function. For each consecutive $\mu := \alpha\gamma^2 \log N$ contiguous source blocks denoted $\mathsf{Fetched}^{\langle d\rangle}[k\mu + 1 : (k+1)\mu]$, there is a segment of μ contiguous destination blocks $\overline{\mathsf{Instr}}^{\langle d+1\rangle}[k\mu+1 : (k+1)\mu]$ such that all addresses in $\mathsf{Instr}^{\langle d+1\rangle}$ that are contained in $\mathsf{Fetched}^{\langle d\rangle}[k\mu + 1 : (k+1)\mu]$ appear in the range $\overline{\mathsf{Instr}}^{\langle d+1\rangle}[k\mu + 1 : (k+1)\mu]$.

(b) A routing permutation $\pi^{d\rightarrow d+1} : [2m] \rightarrow [2m]$.

In other words, the goal of the offline phase is to prepare the source and the destination arrays such that in the online phase, we only perform oblivious routing from every $\mu := \alpha\gamma^2 \log N$ blocks (each containing at most γ labels) in the source to every μ tuples in the destination where $\alpha = \omega(1)$ is any super-constant function. This way, the online phase has $O(\log \mu)$ parallel runtime.

Before explaining how to accomplish the above, we first prove that if the source array, i.e., $\mathsf{Fetched}^{\langle d\rangle}$ has been randomly permuted, then every μ contiguous blocks contain at most μ position identifiers needed by the destination.

Lemma 5. *Let* arr *denote an array of* $2m$ *randomly permuted blocks, each of which contains* γ *items such that out of the* $2m \cdot \gamma$ *items, at most* m *are real and the rest are dummy.*

Then, for any consecutive n *blocks in* arr*, with probability at least* $1 - \exp(-\frac{n}{2\gamma^2})$*, the number of real items contained in them is at most* n*.*

The proof of Lemma 5 follows by a standard concentration argument and is to the online full version [3].

We now explain the offline algorithm, i.e., permute the destination array to be somewhat aligned with the source array such that localized oblivious routing will be sufficient. We describe a parallel oblivious algorithm that completes in $O(m \log m)$ total work and $O(\log m)$ parallel runtime.

1. For each block in $\mathsf{Fetched}^{\langle d \rangle}$, write down a tuple $(\mathsf{minaddr}, \mathsf{maxaddr}, i)$ where $\mathsf{minaddr}$ is the minimum address contained in the block, $\mathsf{maxaddr}$ is the maximum address contained in the block, and i is the offset of the block within the $\mathsf{Fetched}^{\langle d \rangle}$ array.
   Henceforth we refer to the resulting array as $\mathsf{SrcMeta}$.
2. Imagine that the resulting array $\mathsf{SrcMeta}$ and the destination array $\mathsf{Instr}^{\langle d+1 \rangle}$ are concatenated. Now, oblivious sort this concatenated array such that each metadata tuple $(\mathsf{minaddr}, \mathsf{maxaddr}, i) \in \mathsf{SrcMeta}$ is immediately followed by all tuples from $\mathsf{Instr}^{\langle d+1 \rangle}$ whose addresses are contained within the range $[\mathsf{minaddr}, \mathsf{maxaddr}]$.
3. Relying on a parallel oblivious aggregate operation [1,16] (see full version [3] for the definition), each element in the array (resulting from the above step) learns the first metadata tuple $(\mathsf{minaddr}, \mathsf{maxaddr}, i)$ to its left. In this way, each address will learn which block (i.e., i) within $\mathsf{Fetched}^{\langle d \rangle}$ it will receive its position identifier from.
   The result of this step is an array such that each metadata tuple of the $(\mathsf{minaddr}, \mathsf{maxaddr}, i)$ is replaced with a dummy entry $\perp$, and each address addr is replaced with (addr, i), denoting that the address addr will receive its position identifier from the i-th block of $\mathsf{Fetched}^{\langle d \rangle}$.
4. For each non-dummy entry in the above array, tag the entry with a group number $\lfloor \frac{i}{\mu} \rfloor$. For each dummy entry, tag it with $\perp$.
5. Invoke an instance of the oblivious bin packing algorithm and pack the resulting array into $\lceil \frac{2m}{\mu} \rceil$ bins of capacity μ each. We refer to the resulting array as $\overline{\mathsf{Instr}}^{\langle d+1 \rangle}$.
6. Obliviously compute the routing permutation $\pi^{d \to d+1}$ that maps $\overline{\mathsf{Instr}}^{\langle d+1 \rangle}$ to $\mathsf{Instr}^{\langle d+1 \rangle}$.
7. Output $\overline{\mathsf{Instr}}^{\langle d+1 \rangle}$ and $\pi^{d \to d+1}$.

Online phase. The online phase consists of the following steps:

1. For every k, fork an instance of the oblivious routing algorithm such that $\overline{\mathsf{Instr}}^{\langle d+1 \rangle}[k\mu + 1 : (k+1)\mu]$ will receive its position identifiers from $\mathsf{Fetched}^{\langle d \rangle}[k\mu + 1 : (k+1)\mu]$.
   This completes in $O(m \log \mu)$ total work and $O(\log \mu)$ parallel runtime.
2. Apply the routing permutation $\pi^{d \to d+1}$ to $\overline{\mathsf{Instr}}^{\langle d+1 \rangle}$, and output the result as $\mathsf{CInstr}^{\langle d+1 \rangle}$.

5.4 Level-to-Level Routing

Given our core localized routing building block, the full level-to-level position identifier routing algorithm is straightforward to state.

Offline phase. Upon receiving a batch of m memory requests, for each recursion level d in parallel:

- Truncate the addresses to the first d bits and perform conflict resolution. The result is an array of length m containing distinct addresses and dummies to read from recursion level d.
- Randomly permute the resulting array, and obtain an incomplete instruction array $\mathsf{Instr}^{\langle d\rangle}$. It is important for security that the random permutation is performed obliviously such that no information is leaked to the adversary about the permutation.
  For $d = 0$, additionally fill in the position map identifiers and complete the instruction array to obtain $\mathsf{CInstr}^{\langle 0\rangle}$.
- From the $\mathsf{Instr}^{\langle d\rangle}$ array, construct a corresponding incomplete $\mathsf{Fetched}^{\langle d\rangle}$ array where all position identifier fields are left blank as "_". The blocks in $\mathsf{Fetched}^{\langle d\rangle}$ are ordered in the same way as $\mathsf{Instr}^{\langle d\rangle}$.
- If d is not the data level, fork an instance of the localized routing algorithm with input arrays $\mathsf{Fetched}^{\langle d\rangle}$ and $\mathsf{Instr}^{\langle d+1\rangle}$, and output a permuted version of $\mathsf{Instr}^{\langle d+1\rangle}$ denoted $\overline{\mathsf{Instr}}^{\langle d+1\rangle}$ a routing permutation $\pi^{d\to d+1}$.

Online phase. From each recursion level $d = 0, 1, \ldots D$ sequentially where $D = O(\frac{\log N}{\log \Gamma})$ is the total number of recursion levels:

- Based on the completed instruction $\mathsf{CInstr}^{\langle d\rangle}$, allocate an appropriate number of processors for each completed instruction and perform the fetch phase of the OPRAM algorithm. The result is a fetched array $\mathsf{Fetched}^{\langle d\rangle}$.
- Execute the online phase of the localized routing algorithm for recursion level d with the inputs $\mathsf{Fetched}^{\langle d\rangle}$, $\overline{\mathsf{Instr}}^{\langle d+1\rangle}$, and $\pi^{d\to d+1}$. The result is a completed instruction array $\mathsf{CInstr}^{\langle d+1\rangle}$ for the next recursion level.

5.5 Main Upper Bound Theorems

In the interest of space, we defer the full details of our OPRAM construction and proofs to the online full version [3]. Our main theorem is the following:

Theorem 4 (Statistically secure, small-depth OPRAM). *There exists a statistically secure OPRAM scheme (for general block sizes) with $O(\log^2 N)$ total work blowup, and $O(\log N \log\log N)$ parallel runtime blowup, where the OPRAM consumes only $O(1)$ blocks of CPU private cache.*

Using a standard PRF-and-counter compression trick first proposed by Fletcher et al. [7] and later improved and extended to the parallel setting by Chan and Shi [5], we obtain the following computationally secure variant.

Corollary 1 (Computationally secure, small-depth OPRAM). *Assume that one-way functions exist. Then, there exists a computationally secure OPRAM scheme (for general block sizes) with* $O(\log^2 N/\log\log N)$ *total work blowup and* $O(\log N)$ *parallel runtime blowup, where the OPRAM consumes only* $O(1)$ *blocks of CPU private cache.*

Finally, in our online full version [3], we include additional algorithmic results that specifically optimize our OPRAM's depth for sufficiently large block sizes.

Acknowledgments. We thank Rafael Pass for numerous helpful discussions and for being consistently supportive. We thank Feng-Hao Liu and Wei-Kai Lin for helpful conversations regarding the lower bound. This work is supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, CNS-1601879, CNS-1617676, an Office of Naval Research Young Investigator Program Award, a DARPA Safeware grant (subcontract under IBM), a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, a Baidu Research Award, and a VMWare Research Award.

References

1. Boyle, E., Chung, K.-M., Pass, R.: Oblivious parallel RAM and applications. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 175–204. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_7
2. Boyle, E., Naor, M.: Is there an oblivious RAM lower bound? In: TCC (2016)
3. Chan, T.-H.H., Chung, K.-M., Shi, E.: On the depth of oblivious parallel RAM. Cryptology ePrint Archive, Report 2017/861 (2017). http://eprint.iacr.org/2017/861
4. Chan, T.-H.H., Guo, Y., Lin, W.-K., Shi, E.: Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In: ASIACRYPT (2017)
5. Chan, T.-H.H., Shi, E.: Circuit OPRAM: unifying statistically and computationally secure ORAMs and OPRAMs. In: TCC (2017)
6. Chen, B., Lin, H., Tessaro, S.: Oblivious parallel RAM: improved efficiency and generic constructions. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 205–234. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_8
7. Fletcher, C.W., Ren, L., Kwon, A., van Dijk, M., Devadas, S.: Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In: ASPLOS (2015)
8. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC (1987)
9. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**, 431–473 (1996)
10. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22012-8_46
11. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: CCS (2012)

12. Hagerup, T.: Fast and optimal simulations between CRCW PRAMs. In: Finkel, A., Jantzen, M. (eds.) STACS 1992. LNCS, vol. 577, pp. 45–56. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55210-3_172
13. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious RAM and a new balancing scheme. In: SODA (2012)
14. Liu, C., Hicks, M., Harris, A., Tiwari, M., Maas, M., Shi, E.: Ghostrider: a hardware-software system for memory trace oblivious computation. In: ASPLOS (2015)
15. Nayak, K., Katz, J.: An oblivious parallel ram with $O(\log^2 N)$ parallel runtime blowup. Cryptology ePrint Archive, Report 2016/1141 (2016)
16. Nayak, K., Wang, X.S., Ioannidis, S., Weinsberg, U., Taft, N., Shi, E.: GraphSC: parallel secure computation made easy. In: IEEE S&P (2015)
17. Ren, L., Yu, X., Fletcher, C.W., van Dijk, M., Devadas, S.: Design space exploration and optimization of path oblivious RAM in secure processors. In: ISCA, pp. 571–582 (2013)
18. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: ASIACRYPT (2011)
19. Stefanov, E., Shi, E.: Oblivistore: high performance oblivious cloud storage. In: IEEE Symposium on Security and Privacy (S&P) (2013)
20. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S., Path ORAM – an extremely simple oblivious ram protocol. In: CCS (2013)
21. Wang, X.S., Chan, T.-H.H., Shi, E.: Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In: ACM CCS (2015)
22. Wang, X.S., Huang, Y., Chan, T.-H.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: CCS (2014)
23. Williams, P., Sion, R., Tomescu, A.: Privatefs: a parallel oblivious file system. In: CCS (2012)