Circuit OPRAM: A Unifying Framework for Statistically and Computationally Secure ORAMs and OPRAMs

T-H. Hubert Chan
The University of Hong Kong
hubert@cs.hku.hk

Elaine Shi
Cornell University
elaine@cs.cornell.edu

Abstract

An Oblivious Parallel RAM (OPRAM) provides a general method to simulate any Parallel RAM (PRAM) program, such that the resulting memory access patterns leak nothing about secret inputs. OPRAM was originally proposed by Boyle et al. as the natural parallel counterpart of Oblivious RAM (ORAM), which was shown to have broad applications, e.g., in cloud outsourcing, secure processor design, and secure multi-party computation. Since parallelism is common in modern computing architectures such as multi-core processors or cluster computing, OPRAM is naturally a powerful and desirable building block as much as its sequential counterpart ORAM is.

Although earlier works have shown how to construct OPRAM schemes with polylogarithmic simulation overhead, in comparison with best known sequential ORAM constructions, all existing OPRAM schemes are (poly-)logarithmic factors more expensive. In this paper, we present a new framework in which we construct both statistically secure and computationally secure OPRAM schemes whose asymptotical performance matches the best known ORAM schemes in each setting. Since an OPRAM scheme with simulation overhead χ directly implies an ORAM scheme with simulation overhead χ, our result can be regarded as providing a unifying framework in which we can subsume all known results on statistically and computationally secure ORAMs and OPRAMs alike. Particularly for the case of OPRAMs, we also improve the state-of-the-art scheme by superlogarithmic factors.

To achieve the aforementioned results requires us to combine a variety of techniques involving 1) efficient parallel oblivious algorithm design; and 2) designing tight randomized algorithms and proving measure concentration bounds about the rather involved stochastic process induced by the OPRAM algorithm.

1 Introduction

Oblivious RAM (ORAM), initially proposed by Goldreich and Ostrovsky [16, 17], is a powerful primitive that allows oblivious accesses to sensitive data, such that access patterns during the computation reveal no secret information. Since its original proposal [17], ORAM has been shown to be promising in various application settings including secure processors [10, 12, 13, 24, 29], cloud outsourced storage [18, 31, 32, 38] and secure multi-party computation [14, 15, 19, 21, 23, 35].

Although ORAM is broadly useful, it is inherently sequential and does not support parallelism. On the other hand, parallelism is universal in modern architectures such as cloud platforms and multi-core processors. Motivated by this apparent discrepancy, in a recent seminal work [3], Boyle et al. extended the ORAM notion to the parallel setting. Specifically, they defined Oblivious Parallel RAM (OPRAM), and demonstrated that any PRAM program can be simulated obliviously while incurring roughly $O(\log^4 N)$ blowup in running time when consuming the same number of CPUs as the PRAM where N is the total memory size. The result by Boyle et al. [3] was later improved by Chen et al. [6], who showed a logarithmic factor improvement, attaining $O(\log^3 N)$ overhead.

However, we still know of no OPRAM algorithm whose performance can "match" the state-of-the-art sequential counterparts [22, 33, 34]. In particular, in the sequential setting, it is known that computationally secure ORAMs can be constructed with $O(\frac{\log^2 N}{\log \log N})$ simulation overhead [22], and statistically secure ORAMs can be achieved with $O(\log^2 N)$ simulation overhead [34] — these results apply when assuming $O(1)$ blocks of CPU private cache, and they hold for general block sizes, as long as the block is large enough to store its own address. Thus in comparison, state-of-the-art OPRAM schemes are at least a logarithmic factor slower. We thus ask the question:

Can we construct an OPRAM scheme whose asymptotical performance matches the best known sequential counterpart?

Our paper answers this question in the affirmative. To this end, we construct the Circuit OPRAM framework — under this framework we demonstrate both statistically and computationally secure ORAMs whose performance matches the best known ORAM schemes in these respective settings. Our main results are summarized in the following informal theorems.

Theorem 1 (Informal: statistically secure OPRAM)**.** *There exists a statistically secure OPRAM scheme that achieves $O(\log^2 N)$ simulation overhead for general block sizes and $O(1)$ blocks of CPU cache.*

Theorem 2 (Informal: computationally secure OPRAM)**.** *There exists a computationally secure OPRAM scheme that achieves $O(\frac{\log^2 N}{\log \log N})$ simulation overhead for general block sizes and $O(1)$ blocks of CPU cache.*

In both the above theorems, an OPRAM simulation overhead of χ means the following: suppose that the original PRAM consumes m CPUs and computes a program in T time; then we can compile the PRAM into an OPRAM also consuming m CPUs, but completes in $\chi \cdot T$ time[1]. Since an OPRAM scheme with χ simulation overhead immediately implies an ORAM scheme with χ simulation overhead — in some sense, our work provides a unifying framework under which we subsume all known results for statistically secure and computationally secure ORAMs and OPRAMs — and specifically for the case of OPRAM, we improve best known results by at least a logarithmic factor.

[1]Thus, by classical metrics of the parallel algorithms literature, an OPRAM scheme with χ *simulation overhead* incurs a *total work* blowup and a *parallel runtime* blowup of both χ in comparison with the original PRAM.

For generality, we describe our construction in a way that supports the case of varying number of CPUs, i.e., when the underlying PRAM consumes a different number of CPUs in different PRAM steps — this is a desirable property phrased in the original OPRAM work by Boyle et al. [3], although the subsequent work by Chen et al. [6] fails to achieve it.

Last but not the least, we show that when the block size is sufficiently large, our framework implies an OPRAM scheme with $O(\log N)$ simulation overhead (when m is not too small) — also matching the best-known sequential ORAM result for large block sizes [34].

1.1 Technical Highlights

Obtaining an OPRAM as tight as its sequential counterpart turns out to be rather non-trivial. Part of the technical sophistication stems from the fact that we did not find any generic method that can blackbox-compile an efficient ORAM to an OPRAM scheme with matching overhead. As a result, our construction requires opening up and building atop the Circuit ORAM scheme [34] (which is a state-of-the-art ORAM scheme among others) in a *non-blackbox* manner[2].

We follow the paradigm for constructing OPRAM schemes proposed by Boyle et al. [3] and Chen et al. [6]. On a high level, we leverage a tree-based ORAM scheme [30, 34] but truncate the tree at a level with m nodes, thus creating m disjoint subtrees. At this point, a simple approach that is taken by earlier works [6] is to have a single CPU in charge of each subtree. When a batch of m memory requests come in, each request will want to fetch data from a random subtree. By a simple balls-and-bins argument, while each subtree receives only $O(1)$ requests in expectation, the most unlucky sub-tree will need to serve super-logarithmically many requests (to obtain a negligible failure probability). Thus the naive approach is to have each subtree's CPU serve these super-logarithmically many requests sequentially. After fetching the m blocks, the OPRAM data structure must be maintained by remapping every fetched block to a random new subtree. Again, although each subtree gets assigned $O(1)$ remapped blocks in expectation during this maintain stage, the most unlucky subtree can obtain super-logarithmically many blocks. For obliviousness, it is important that we hide from the adversary to which subtree each block gets remapped. Unfortunately this also means that we cannot disclose how many remapped blocks are received by each subtree — and thus previous works [6] adopt the simple approach of padding: even when a subtree may receive only 1 remapped block, it still must perform dummy operations to pretend that it receives superlogarithmically many blocks.

Thus, a primary reason that causes existing constructions [3,6] to be inefficient is the *discrepancy between the average-case contention experienced by subtree and the worst-case contention.* In the above description, this discrepancy reflects in both the request phase and the maintain phase. In both phases, each subtree receives $O(1)$ requests or blocks in expectation, but the worst-case can be superlogarithmic (assuming negligible failure probability).

Thus the core of the question is how to avoid the blowup resulting from the aforementioned average-case and worst-case discrepancy. We would like our scheme to incur a cost that reflects the average-case contention, not the worst-case. To achieve this, we need different techniques for the online request and offline maintain phases respectively:

- For the request phase, it does not violate obliviousness to disclose how many requests are received by each subtree, and thus the nature of the problem is how to design an efficient parallel oblivious algorithm to serve all m requests in parallel with m CPUs, and avoid any subtree's CPU having to process super-logarithmically many requests sequentially. As we will show in later sections, the core of the problem is how to design an efficient and oblivious parallel removal algorithm

[2]We did not build atop the Path ORAM [33] scheme since Path ORAM achieves the same simulation overhead as Circuit ORAM [34] but consuming super-logarithmic CPU cache rather than $O(1)$.

(referred to "simultaneous removal" in later sections) that removes fetched blocks from the tree-paths — this is challenging since several CPUs may read paths that overlap with one another, leading to possible write contention.

- In the maintain phase, on the contrary, it would violate obliviousness to disclose how many remapped blocks are received by each subtree. Earlier schemes achieve this by pretending to reroute superlogarithmically many blocks to every subtree, thus always incurring the worst-case cost. Our idea is to redesign the underlying stochastic process to avoid this padding-related loss. To this end, we introduce a new technique called "lazy eviction", where we do not route remapped blocks to their new subtrees immediately — instead, with every operation, each subtree has a budget for receiving only a constant number of remapped blocks; and the overflowing blocks that do not have a chance to be rerouted to their assigned subtrees will remain in a "pool" data structure whose size we shall bound with measure concentration techniques.

When we put these techniques together, we obtain on OPRAM scheme that induces a stochastic process that is somewhat involved to reason about. Analyzing this OPRAM-induced stochastic process and proving measure concentration results (e.g., bounds on pool and stash sizes) are non-trivial challenges that we have to overcome in this paper. Although we build on top of the Circuit ORAM scheme in a non-blackbox manner, we wish to maximally reuse the (somewhat involved) measure concentration results proven in the Circuit ORAM work [34] (albeit in a non-blackbox manner). Thus, when we design our Circuit OPRAM algorithm, we take care to ensure that the resulting randomized process is *stochastically dominated* by that of the underlying Circuit ORAM algorithm (in terms of overflows) — to this end, our algorithm tries to "imitate" the stochastic behavior of Circuit ORAM in several places, e.g., in selecting which remapped block gets priority to be rerouted back to its subtree. A rather technical part of our proof is to show that the resulting OPRAM scheme is indeed stochastically dominated by Circuit ORAM in terms of overflows.

1.2 Related Work

Closely related and independent works. Subsequent to our online technical report, Nayak and Katz [25] also released a technical report that claimed seemingly similar results. We stress that *our construction is a* $\log N \cdot \mathsf{poly} \log \log N$ *factor more efficient than the work by Nayak and Katz* — despite their paper's title claiming to achieve $O(\log^2 N)$ overhead, their $O(\log^2 N)$ overhead did not account for the inter-CPU communication which is $O(\log^3 N \mathsf{poly} \log \log N)$ in their scheme assuming $O(1)$ CPU cache — more specifically, Nayak and Katz's scheme does not improve the inter-CPU communication in comparison with Chen et al. [6] (whereas we improve by a super-logarithmic factor); but they adopt a variant of our simultaneous removal algorithm to improve the CPU-memory communication of Chen et al. [6].

In our paper, we adopt a more general and cleaner model than earlier and concurrent OPRAM works [3,6,25], in that we assume that all inter-CPU communication is routed through memory too. In this way, we use a single metric called simulation overhead to characterize both CPU-memory cost and inter-CPU communication. Using our metric, an OPRAM scheme with simulation overhead χ means that both the CPU-memory cost and the inter-CPU communication have at most χ blowup in comparison with the original PRAM.

Below we review the line of works on constructing ORAMs and OPRAMs.

Oblivious RAM (ORAM). Oblivious RAM (ORAM) was initially proposed by Goldreich and Ostrovsky [16, 17] who showed that any RAM program can be simulated obliviously incurring only $O(\alpha \log^3 N)$ runtime blowup, while achieving a security failure probability that is negligible in N. Numerous subsequent works [9, 18, 22, 27, 28, 30, 33, 34, 34–38] improved Goldreich and Ostrovsky's

seminal result in different application settings including cloud outsourcing, secure processor, and secure multi-party computation.

Most of these schemes follow one of two frameworks: the hierarchical framework, originally proposed by Goldreich and Ostrovsky [16, 17], or the tree-based framework proposed by Shi et al. [30]. To date, some of the (asymptotically) best schemes include the following: 1) Kushilevitz et al. [22] showed a *computationally secure* ORAM scheme with $O(\log^2 N/\log\log N)$ runtime blowup for general block sizes; and 2) Wang et al. construct Circuit ORAM [34], a *statistically secure* ORAM that achieves $O(\alpha \log^2 N)$ runtime blowup for general block sizes[3] and $O(\alpha \log N)$ runtime blowup for large enough blocks. At the time of the writing, we are not aware of any approach that transforms a state-of-the-art hierarchical ORAM such as Kushilevitz et al. [22] into an OPRAM scheme with matching simulation overhead — even if this could be done, it still would not be clear how to match the best known ORAM results for the statistical security setting. Our work henceforth builds on top of the tree-based ORAM framework, and specifically, Circuit ORAM [34].

On the lower bound side, Goldreich and Ostrovsky [16,17] demonstrated that any ORAM scheme (with constant CPU cache) must incur at least $\Omega(\log N)$ runtime blowup. This well-known lower bound was recently shown to be tight (under certain parameter ranges) by the authors of Circuit ORAM [34], who showed a matching upper bound for sufficiently large block sizes. Goldreich and Ostrovsky's lower bound applies to OPRAM too since by our definition of simulation overhead, an OPRAM scheme with χ simulation overhead implies an ORAM scheme with χ simulation overhead. We note that while the Goldreich and Ostrovsky lower bound is quite general, it models each block as being opaque — recently, an elegant result by Boyle and Naor [4] discussed the possibility of proving a lower bound without this restriction. Specifically, they showed that proving a lower bound without the block opaqueness restriction is as hard as showing a superlinear lower bound on the sizes of certain sorting circuits. Further, the Goldreich-Ostrovsky lower bound is also known not to hold when the memory (i.e., ORAM server) is capable of performing computation [2,9] — in this paper, we focus on the classical ORAM/OPRAM setting where the memory does not perform any computation besides storing and fetching data at the request of the CPU.

Oblivious Parallel RAM (OPRAM). Given that many modern computing architectures support parallelism, it is natural to extend ORAM to the parallel setting. As mentioned earlier, Boyle et al. [3] were the first to formulate the OPRAM problem, and they constructed an elegant scheme that achieves $O(\alpha \log^4 N)$ blowup both in terms of total work and parallel runtime. Their result was later improved by Chen et al. [6] who were able to achieve $O(\alpha \log^3 N)$ blowup both in terms of total work and parallel runtime under $O(\log^2 N)$ blocks of CPU cache. These results can easily be recast to the $O(1)$ CPU cache setting by applying a standard trick that leverages oblivious sorting to perform eviction [34, 35]. We note that Chen et al. [6] actually considered *CPU-memory communication* and *inter-CPU communication* as two separate metrics, and their scheme achieves $O(\alpha \log^2 N \log\log N)$ CPU-memory communication blowup, but $O(\alpha \log^3 N)$ inter-CPU communication blowup. In this paper, we consider the more general PRAM model where all inter-CPU communication is implemented through CPU-memory communication. In this case, the two metrics coalesce into one (i.e., the maximum of the two).

Besides OPRAM schemes in the standard setting, Dachman-Soled et al. [8] considered a variation of the problem (which they refer to as "Oblivious Network RAM") where each memory bank is assumed to be oblivious within itself, and the adversary can only observe which bank a request goes to. Additionally, Nayak et al. [26] show that for parallel computing models that are more restrictive than the generic PRAM (e.g., the popular GraphLab and MapReduce models), there

[3]Note that in this paper, the new OPRAM techniques we introduce allow us to remove the super-constant factor α and thus we achieve $O(\log^2 N)$ overhead for general block sizes. Therefore, strictly speaking, we improve the best-known results for statistic security [34] by a super-constant factor. For sufficiently large block sizes, we achieve $O(\alpha \log N)$ simulation overhead, matching the sequential counterpart Circuit ORAM.

exist efficient parallel oblivious algorithms that asymptotically outperform known generic OPRAM. Some of the algorithmic techniques employed by Nayak et al. [26] are similar in nature to those of Boyle et al. [3].

Subsequent work. In subsequent work, Chan et al. [5] consider a new model for OPRAM, where the OPRAM has access to more CPUs than the original PRAM. In that model, they characterize an OPRAM's overhead using two metrics, *total work blowup* and *parallel runtime blowup* (the latter metric also referred to as depth blowup). Chan et al. show that any OPRAM scheme that treats block contents as opaque must incur at least $\Omega(\log m)$ depth blowup where m is the number of CPUs of the original PRAM. Further, they devise non-trivial algorithmic techniques that improves the depth of Circuit OPRAM (while preserving total work) by recruiting the help of logarithmically many more CPUs. Further, Chan et al. [5] show that their algorithm's depth is tight in the parameter m when the block size is sufficiently large.

2 Informal Overview of Our Results

In this section, we take several intermediate steps to design a basic Circuit OPRAM construction. Specifically, we start out by reviewing the high-level idea introduced earlier by Chen et al. [6]. Then, we point out why their scheme suffers from an extra logarithmic blowup in performance in comparison with the best known sequential algorithm. Having made these observations, we describe our new techniques to avoid this blowup. For simplicity, in this section, we focus on describing the basic, statistically secure Circuit OPRAM algorithm with a fixed number of CPUs denoted m. In later formal sections, we will describe the full scheme supporting the case of varying m, and additional techniques that allow us to shave another $\log \log N$ factor by leveraging a PRF to compress the storage of the random position maps. Like in earlier ORAM/OPRAM works [16–18, 30, 33, 34], we will assume that the number of blocks N is also the security parameter.

2.1 Background: Circuit ORAM

We review tree-based ORAMs [7, 30, 33, 34] originally proposed by Shi et al. [30]. We specifically focus on describing the Circuit ORAM algorithm [34] which we build upon.

We assume that memory is divided into atomic units called blocks. We first focus on describing the *non-recursive version*, in which the CPU stores in its local cache a *position map* henceforth denoted as posmap that stores the position for every block.

Data structures. The memory is organized in the form of a binary tree, where every tree node is a *bucket* with a capacity of $O(1)$ blocks. Buckets hold blocks, where each block is either dummy or real. Throughout the paper, we use the notation N to denote the total number of blocks. Without loss of generality, we assume that $N = 2^L$ is a power of two. The ORAM binary tree thus has height L.

Besides the buckets, there is also a *stash* in memory that holds overflowing blocks. The stash is of size $O(\alpha \log N)$, where $\alpha = \omega(1)$ is a parameter related to the failure probability. Just like buckets, the stash may contain both real and dummy blocks. Henceforth, for convenience, we will often *treat the stash as part of the root bucket.*

Main path invariant. The main invariant of tree-based ORAMs is that every block is assigned to the path from the root to a randomly chosen leaf node. Hence, the path for each block is indicated by the *leaf identifier* or the *position identifier*, which is stored in the aforementioned position map posmap. A block with virtual address i must reside on the path indicated by $\mathsf{posmap}[i]$.

Operations. We describe the procedures for reading or writing a block at virtual address i.

- *Read and remove.* To read a block at virtual address i, the CPU looks up its assigned path indicated by $\mathsf{posmap}[i]$, and reads this entire path. If the requested block is found at some location on the path, the CPU writes a dummy block back into the location. Otherwise, the CPU simply writes the original block back. In both cases, the block written back is re-encrypted such that the adversary cannot observe which block is removed.

- *Remap.* Once a block at virtual address i is fetched, it is immediately assigned to a new path. To do this, a fresh random path identifier is chosen and $\mathsf{posmap}[i]$ is modified accordingly. The block fetched is then written to the last location in the stash (the last location is guaranteed to be empty except with negligible probability at the end of each access). If this is a write operation, the block's contents may be updated prior to writing it back to the stash.

- *Evict.* Two paths (particularly, one to the left of the root and one to the right of the root) are chosen for eviction according to an appropriate data independent criterion. Specifically, for the remainder of the paper, we will assume that the paths are chosen based on the deterministic reverse lexicographical order algorithm adopted in earlier works [14,34], the choice of eviction path is non-essential to the understanding of the algorithm (but matters to the stochastic analysis).

  For each path chosen (that includes the stash), the CPU performs an eviction procedure along this path. On a high level, eviction is a maintenance operation that aims to move blocks along tree paths towards the leaves — and importantly, in a way that respects the aforementioned path invariant. The purpose of eviction is to avoid overflow at any bucket.

  Specifically in Circuit ORAM, this eviction operation involves making two metadata scans of the eviction path followed by a single data block scan [34].

A useful property of Circuit ORAM's eviction algorithm. For the majority of this paper, the reader need not know the details of the eviction algorithm. However, we point out a useful observation regarding Circuit ORAM's eviction algorithm.

Fact 1 (Circuit ORAM eviction)**.** *Suppose Circuit ORAM's eviction algorithm is run once on some path denoted* $\mathsf{path}[0..L]$*, where by convention we use* $\mathsf{path}[0]$ *to denote the root (together with the stash) and* $\mathsf{path}[L]$ *is the leaf in the path. Then, for every height* $i \in \{1, \ldots, L\}$*, it holds that at most one block moves from* $\mathsf{path}[0..i-1]$ *to* $\mathsf{path}[i..L]$*. Further, if a block did move from* $\mathsf{path}[0..i-1]$ *to* $\mathsf{path}[i..L]$*, then it must be the block that can be evicted the deepest along the eviction path (and if more than one such block exists, an arbitrary choice could be made).*

Recursion. So far, we have assumed that the CPU can store the entire position map posmap in its local cache. This assumption can be removed using a standard recursion technique [30]. Specifically, instead of storing the position map in the CPU's cache, we store it in a smaller ORAM in memory — and we repeat this process until the position map is of constant size.

As long as each block can store at least two position identifiers, each level of the recursion will reduce the size of the ORAM by a constant factor. Therefore, there are at most $O(\log N)$ levels of recursion. Several tree-based ORAM schemes also describe additional tricks in parametrizing the recursion for larger block sizes [33, 34]. We will not describe these tricks in detail here, but later in Section 12 we will recast these tricks in our OPRAM context and describe further optimizations for large block sizes.

Circuit ORAM performance. For general block sizes, Circuit ORAM achieves $O(\alpha \log N)$ blowup (in terms of bandwidth and the number of accesses) in the non-recursive version, and $O(\alpha \log^2 N)$ blowup across all levels of recursion. The CPU needs to hold only $O(1)$ blocks at any point in time.

2.2 Warmup: The CLT OPRAM Scheme

We outline the elegant approach by Chen et al. [6] which achieves $O(\log^3 N)$ simulation overhead. Although Chen et al. [6]'s construction builds on top of Path ORAM [33], we describe a (slightly improved) variant of their scheme [6] that builds atop Circuit ORAM instead, but in a way that captures the core ideas of Chen et al. [6].

Suppose we start with Circuit ORAM [34], a state-of-the-art tree-based ORAM. Circuit ORAM is sequential, i.e., supports only one access at a time — but we now would like to support m simultaneous accesses. Without loss of generality, we assume that $m \leq N$ throughout the paper. In our informal overview, we often assume that m is not too small for convenience, and we deal with the case of small m in later technical sections.

Challenge for parallel accesses: write conflicts. A strawman idea for constructing OPRAM is to have m CPUs perform m ORAM access operations simultaneously. Reads are easy to handle, since the m CPUs can read m paths simultaneously. The difficulty is due to write conflicts, which arise from the need for m CPUs to 1) each remove a block from its bucket if it is the requested one; and 2) to perform eviction after the reads. In particular, observe that the paths accessed by the m CPUs overlap, and therefore it may be possible that two or more CPUs will be writing the same location at the same time. It is obvious that if such write conflicts are resolved arbitrarily where an arbitrary CPU wins, we will not be able to maintain even correctness.

Subtree partitioning to reduce write contention. Chen et al.'s core idea is to remove buckets from smaller heights of the Circuit ORAM tree, and start at a height with m buckets. In this way, we can view the Circuit ORAM tree as m disjoint subtrees — write contentions can only occur inside each subtree but not across different subtrees.

Now since there are m CPUs in the original PRAM, each batch contains m memory access requests — without loss of generality, we will assume that all of these m requests are distinct — had it not been the case, it is easy to apply the conflict resolution algorithm of Boyle at al. [3] to suppress duplicates, and then rely on oblivious routing to route fetched results back to all m requesting CPUs.

Each of these m requests will look for its block in a random subtree independently. By the Chernoff bound, each subtree receives $O(\alpha \log N)$ requests with all but $\mathsf{negl}(N)$ probability where $\alpha = \omega(1)$ is any super-constant function. Chen et al.'s algorithm proceeds as follows, where performance metrics are *without recursion*.

1. *Fetch.* A designated CPU per subtree performs the read phase of these $O(\alpha \log N)$ requests *sequentially*, which involves reading up to $O(\alpha \log N)$ paths in the tree. Since each path is $O(\log N)$ in length, this incurs $O(\alpha \log^2 N)$ parallel steps.
2. *Route.* Obliviously route the fetch results to the requesting CPUs. This incurs $O(\log m)$ parallel steps with m CPUs.
3. *Remap.* Assign each fetched block to a random new subtree and a random leaf within that subtree. Similarly, each subtree receives $\mu = O(\alpha \log N)$ blocks with all but $\mathsf{negl}(N)$ probability. Now, adopt an oblivious routing procedure to route exactly μ blocks back to each subtree, such that each tree receives blocks destined for itself together with padded dummy blocks. This incurs $O(\alpha \log m \log N)$ parallel steps with m CPUs.
4. *Evict.* Each subtree CPU *sequentially* performs $\mu = O(\alpha \log N)$ evictions for its own subtree. This incurs $O(\alpha \log^2 N)$ parallel steps with m CPUs.

Note that to make the above scheme work, Chen et al. [6] must assume that each subtree CPU additionally stores an $O(\alpha \log N)$-sized stash that holds all overflowing blocks that are destined for

the particular subtree — we will get rid of this CPU cache, such that each CPU only needs $O(1)$ blocks of transient storage and does not need any permanent storage.

Recursive version. The above performance metrics assumed that all CPUs get to store, read, and update a shared position map for free. To remove this assumption, we can employ the standard recursion technique of the tree-based ORAM framework [30] to store this position map. We stress that when applying recursion, we must perform conflict resolution at each recursion level to ensure that all non-dummy requests have distinct addresses at each recursion level. The position identifiers fetched at a position map level will be obliviously routed to the fetch CPUs at the next recursion level.

Assuming that each block has size at least $\Omega(\log N)$ bits, there can be up to $\log N$ levels of recursion. Therefore, Chen et al.'s OPRAM scheme incurs $O(\alpha \log^3 N)$ simulation overhead using the same number of CPUs as the original PRAM.

2.3 Our Construction: Intuition

Why the CLT OPRAM is inefficient. First, we need to observe why the CLT OPRAM [6] is inefficient. There are two fundamental reasons why the CLT OPRAM scheme suffers from an extra $\log N$ factor in overhead.

1. During the fetch phase, a single CPU per subtree acts sequentially to fetch all requests that belong to the subtree. Although on average, each subtree receives $O(1)$ requests, in the worst case a subtree may receive up to $\alpha \log N$ requests (to obtain $\mathsf{negl}(N)$ security failure). Since serving each request involves reading a tree path of $\log N$ in length and then removing the block fetched from the path, serving all $\alpha \log N$ requests sequentially with a single CPU would then require $O(\alpha \log^2 N)$ time — over all $O(\log N)$ recursion levels, the blowup would then be $O(\alpha \log^3 N)$.

2. Similarly, during the eviction phase, a single CPU is in charge of performing all evictions a subtree receives. Although on average, each subtree receives $O(1)$ evictions, in the worst case a subtree may receive up to $\alpha \log N$ evictions (to obtain $\mathsf{negl}(N)$ security failure). Similarly, to serve all $\alpha \log N$ evictions with a single CPU would require $O(\alpha \log^2 N)$ time — and after recursion, the blowup would be $O(\alpha \log^3 N)$.

Therefore, the crux is how to improve the efficiency of the above two steps. To this end, we need to introduce a few new ideas described below.

Simultaneous removal. Reading data from the m subtrees can be split into two steps: 1) reading m paths to search for the m blocks requested; and 2) removing the m fetched blocks. Reading m paths can be parallelized trivially by having m CPUs each read a path — note that it is safe to reveal how many requests go to each subtree. Therefore, the crux is how to in parallel remove the m fetched blocks from the respective tree paths. The challenge here is that the tree paths may intersect — recall that each subtree may receive up to $\alpha \log N$ requests in the worst-case, and therefore the simultaneous removal algorithm must handle potential write conflicts.

We detail our new simultaneous removal algorithm in Section 7.1.

Lazy eviction. The eviction stage is more tricky. Unlike the fetch phase where it is safe to reveal which requests go to which subtrees, here it must be kept secret from the adversary how many evictions each subtree receives. At first sight, it would seem like it is necessary to pad the number of evictions per subtree to $\alpha \log N$ to hide the actual number of evictions each subtree receives.

Our idea is to perform eviction lazily. We perform only a single (possibly dummy) eviction per subtree for each batch of m requests — for technical reasons we will have $2m$ subtrees in total

instead of m subtrees, since this makes evictions on average faster than the rate of access. In particular, if there exists one or more blocks wanting to be evicted to a subtree, a real eviction takes place; otherwise, a dummy eviction takes place for the corresponding subtree.

Obviously, such lazy eviction would mean that some elements will be left over and cannot be evicted back into the subtrees. Therefore, we introduce a new data structure called a *pool* to store the leftover blocks that fail to be evicted. Later, we will prove that the pool size is upper bounded by $O(m + \alpha \log N)$ except with $\mathsf{negl}(N)$ probability.

Due to the introduction of the pool, when a batch of requests come, we will need to serve these requests not only from the subtrees, but also from the pool as well — serving requests from the pool can be done in parallel through a standard building block called oblivious routing [3].

Selection of eviction candidates and pool occupancy. Recall that during the eviction stage, we would like to perform a single eviction per subtree. This would require an oblivious algorithm to select eviction candidates from the pool and route these candidates to the respective subtrees. Intuitively, if multiple blocks in the pool are destined for a given subtree, we should select one that has a maximum chance of being evicted, since this can hopefully give us a tight bound on the leftover blocks in the pool. As a result, suppose that a certain path denoted path is being evicted for a certain subtree, we will select a block in the pool that is *deepest* with respect to path for this subtree — as defined in the Circuit ORAM [34] work, this means that this block can legally reside in a deepest height (i.e., closest to the leaf) in path.

It turns out that using this eviction candidate selection strategy, we can view the union of the subtrees and the pool logically as a big Circuit ORAM tree — where the subtrees represent heights $\log_2(2m)$ or higher; and the pool represents smaller heights below $\log_2(2m)$ as well as the stash of Circuit ORAM. At this moment, it would seem like bounding on the pool occupancy would directly translate to bounding blocks remaining in the smaller heights of Circuit ORAM — although there is one additional subtlety: in Circuit ORAM, we perform one access followed by one eviction, whereas here we perform a batch of m accesses followed by a batch of m evictions. To handle this difference, we prove a stochastic domination result, showing that such *batched* eviction can only reduce the number of blocks in height $\log_2(2m)$ or smaller than non-batched — in this way, we can reuse Circuit ORAM's stochastic analysis for bounding the pool size.

2.4 Putting it Altogether

Putting the above ideas together would expose a few more subtleties. We give a high-level overview of our basic construction below.

A pool and $2m$ subtrees: reduce write contention by partitioning. Following the approach of Chen et al. [6], we reduce write contention by partitioning the Circuit ORAM into $2m$ subtrees[4]. However, on top of Chen et al. [6], we additionally introduce the notion of a pool, a data structure that we will utilize to amortize evictions across time.

We restructure a standard Circuit ORAM tree in the following manner. First, we consider a height with $2m$ buckets, which gives us $2m$ disjoint subtrees. All buckets from smaller heights, including the Circuit ORAM's stash, contain at most $O(m+\alpha \log N)$ blocks— we will simply store these $O(m+\alpha \log N)$ blocks in an unstructured fashion in memory, henceforth referred to as a *pool*.

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. [3] to suppress duplicates, and after data has been fetched, rely on oblivious routing to send fetched data to all request CPUs.

[4]Although we choose $2m$ for concreteness, any $c \cdot m$ for a constant $c > 1$ would work.

Now, we look up the requested blocks in two places, both the pool and the subtrees:

- *Subtree lookup:* Suppose that the position labels of the m requests have been retrieved (we will later show how to achieve this through a standard recursion technique) — this defines m random paths in the $2m$ subtrees. We can now have m fetch CPUs each read a path to look for a desired block. All fetched blocks are merged into the central 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 $O(\log N)$ parallel steps with m CPUs. We will treat the case of small m separately later in the paper.

Maintain. In the maintain phase, we must 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.

  Therefore, we propose a new oblivious parallel algorithm for efficient simultaneous removal. Our algorithm allows removal of the m fetched blocks across all trees in $O(\log N)$ time using m CPUs. We defer the detailed description of this simultaneous removal algorithm to Section 7.

- *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 with respect to the current eviction path will be chosen (as mentioned later, eviction paths are chosen using a standard deterministic order lexicographical ordering mechanism [14, 34]). 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 $O(\log N)$ parallel steps with m CPUs assuming that m is not too small — we defer the treatment of small m to later parts of the paper. The details of this algorithm will be spelled out in Section 10.1.

- *Eviction.* We then perform eviction over one tree path for every subtree where the eviction path is selected using the standard deterministic lexicographically order algorithm — since the details of eviction path selection are non-essential to the understanding of our Circuit OPRAM, we refer the reader to earlier works for a detailed exposition [14, 34]. 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 original 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

$O(\log N)$ parallel steps with m CPUs, assuming that m is not too small. We defer the special treatment of small m to later parts of the paper.

Recursion and performance. So far, we have assumed that a position map can be stored and accessed by the CPUs for free. We can remove this assumption through a standard recursion technique [3, 30]. Note that we need to perform conflict resolution at all levels of recursion, and perform oblivious routing to route the fetched position identifiers to the fetch CPUs at the next recursion level. When we count all $O(\log N)$ recursion levels, the above basic construction achieves $O(\log^2 N)$ blowup when m is not too small — we defer the special-case treatment of small m to later parts of the paper.

2.5 Extensions

Improve performance asymptotically with PRFs. In Section 11, we will describe additional techniques that allow us to improve the OPRAM's blowup to $O(\frac{\log^2 N}{\log \log N})$ assuming the usage of a pseudo-random function (PRF) — of course, the resulting scheme would then only have computational security rather than statistical security. To this end, we rely on an elegant technique first proposed by Fletcher et al. [11] that effectively "compresses" position labels by relying on a PRF to compute the blocks' leaf identifiers from "compressed counters". Fletcher et al.'s technique was designed for tree-based ORAMs — as we show later in the paper, we need to make some adaptations to their algorithm to make it work with OPRAMs.

Varying number of CPUs. Our overview earlier assumes that the original PRAM always has the same number of CPUs in every time step, i.e., all batches of memory requests have the same size. We can further extend our scheme for the case when the number of PRAM CPUs varies over time. Below we briefly describe the idea while leaving details to Sections 7. Without loss of generality, henceforth we assume that in every time step, the number of requests in a batch m is always a power of 2 — if not, we can simply round it to the nearest power of 2 incurring only $O(1)$ penalty in performance.

Suppose that the OPRAM scheme currently maintains $2\widehat{m}$ subtrees, but the incoming batch has $m > \widehat{m}$ number of requests. In this case, we will immediately adjust the number of subtrees to $2m$. This can be done simply by merging more heights of the tree into the pool.

The more difficult case is when the incoming batch contains less than $m < \widehat{m}$ requests. In this case, we need to decrease the number of subtrees. In the extreme case when m drops from $\sqrt{N}$ to 1, it will be too expensive to reconstruct up to $\Theta(\log N)$ heights of the ORAM tree.

Instead, we argue in Section 3.2 that without loss of generality, we may assume that if m decreases, it may only decrease by a factor of 2. Hence, every time we just need to halve the number of subtrees — and to achieve this we only need to reconstruct one extra height of the big ORAM tree, which can be achieved through oblivious sorting in $O(\log \widehat{m})$ parallel steps with $O(\widehat{m})$ CPUs.

Results for large block sizes. Finally, we note that when the block size is N^ϵ for any constant $0 < \epsilon < 1$, Circuit OPRAM achieves $O(\log N)$ simulation overhead when $m \geq \alpha \log \log N$. In light of Goldreich and Ostrovsky's $\Omega(\log N)$ lower bound [16, 17], Circuit OPRAM is therefore asymptotically optimal under large block sizes.

2.6 Paper Organization

The remainder of the paper will formally present the ideas described in this section and describe additional results including

1. How to support the case when the number of CPUs varies over time (Sections 5, 6, 7, and 10);
2. Algorithmic details for the case of small m (Sections 5, 6, 7, and 10);
3. Additional techniques to improve the overhead of the scheme by a $\log \log N$ factor assuming the existence of PRFs and achieving computational (rather than statistical) security (Section 11);
4. Detailed proofs where the security proof (Section 9) is somewhat straightforward but the most technically involved part is to prove that Circuit OPRAM's stochastic process is dominated by that of Circuit ORAM (Section 8) such that we can leverage Circuit ORAM's stochastic analysis [34] for bounding the pool and stash sizes of Circuit OPRAM; and
5. Interpretations of our results under larger block sizes and other relevant metrics (Section 12).

3 Preliminaries

3.1 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 support a more general PRAM model where the number of CPUs in each time step may vary. Specifically, in each step $t \in [T]$, we use m_t to denote the number of CPUs. In each step, each CPU executes a next instruction circuit denoted Π, updates its CPU state; and further, CPUs interact with memory through request instructions $\vec{I}^{(t)} := (I_i^{(t)} : i \in [m_t])$. 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 $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$ performed on the virtual memory block with address addr and block value $\mathsf{data} \in \{0,1\}^B \cup \{\perp\}$. If $\mathsf{op} = \mathsf{read}$, 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 $\mathsf{op} = \mathsf{write}$, 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 just like earlier OPRAM works [3, 6]:

- The original PRAM supports concurrent reads and concurrent writes (CRCW) with an arbitrary, parametrizable rule for write conflict resolution. In other words, there exists some priority rule to determine which write operation takes effect if there are multiple concurrent writes in some time step t.
- 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 our compiled OPRAM uses a weaker PRAM model. For a detailed explanation on how stronger PRAM models can emulate weaker ones, we refer the reader to the work by Hagerup [20].

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, and the number of CPUs activated in each time step are *fixed* a priori and *publicly known* parameters. Therefore, we can consider a PRAM to be a tuple

$$\mathsf{PRAM} := (\Pi, N, T,\ (m_t : t \in [T])),$$

where Π denotes the next instruction circuit, N denotes the total memory size (in terms of number of blocks), T denotes the PRAM's total runtime, and m_t denotes the number of CPUs to be activated in each time step $t \in [T]$. Henceforth, we refer to the vector $(m_1, \ldots, m_T)$ as the PRAM's *activation schedule* as defined by Boyle et al. [3].

Without loss of generality, we assume that $N \geq m_t$ for all t. Otherwise, if some $m_t > N$, we can adopt a trivial parallel oblivious algorithm (through a combination of conflict resolution and oblivious multicast) to serve the batch of m_t requests in $O(\log m_t)$ parallel time with m_t CPUs.

3.2 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, and that the CPU activation pattern in each time step is also fixed a priori and publicly known.

Statistical and computational indistinguishability. Given two ensembles of distributions $\{X_N\}$ and $\{Y_N\}$ (parameterized with N), we use the notation $\{X_N\} \overset{\epsilon(N)}{\equiv} \{Y_N\}$ to mean that for any (possibly computationally unbounded) adversary $\mathcal{A}$,

$$\left|\Pr[\mathcal{A}(x) = 1 \,|x \overset{\$}{\leftarrow} X_N] - \Pr[\mathcal{A}(y) = 1 \,|y \overset{\$}{\leftarrow} Y_N]\right| \leq \epsilon(N).$$

We use the notation $\{X_N\} \overset{\epsilon(N)}{\equiv_{\mathsf{c}}} \{Y_N\}$ to mean that for any non-uniform probabilistic polynomial-time adversary $\mathcal{A}$,

$$\left|\Pr[\mathcal{A}(1^N, x) = 1 \,|x \overset{\$}{\leftarrow} X_N] - \Pr[\mathcal{A}(1^N, y) = 1 \,|y \overset{\$}{\leftarrow} Y_N]\right| \leq \epsilon(N).$$

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. More specifically, for each time step $t \in [T]$, $\mathsf{Addresses}(\mathsf{PRAM}, x)$ includes the memory addresses requested by the set of active CPUs S_t in time step t along with their CPU identifiers, as well as whether each memory request is a read or write operation.

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. Henceforth we often use the notation OPRAM to denote a PRAM that satisfies obliviousness.

In this paper, following the convention of most existing ORAM and OPRAM works [16, 17, 22, 33, 34], we will require that the security failure probability be negligible in the N, i.e., the PRAM's total memory size.

Oblivious simulation and performance measures. We say that a given OPRAM *simulates* a PRAM if for every input $x \in \{0,1\}^*$, $\Pr[\mathsf{OPRAM}(x) = \mathsf{PRAM}(x)] = 1$ where 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.

Like in prior works on OPRAM [3, 6], in this paper, we consider *activation-preserving* oblivious simulation of PRAM. Specifically, let $(m_1, \ldots, m_T)$ be the original PRAM's activation schedule, we require that the corresponding OPRAM's activation schedule to be

$$(m_1)_{i=1}^{\chi}, (m_2)_{i=1}^{\chi}, \ldots, (m_T)_{i=1}^{\chi},$$

where χ is said to be the OPRAM's *simulation overhead* (also referred to as *blowup*). In other words, henceforth in the paper, we will simulate the i-th step of the PRAM using m_i CPUs — the same number as the original PRAM. Without loss of generality, we will often assume $O(m_i)$ CPUs are available, since we can always use one CPU to simulate $O(1)$ CPUs with only constant blowup. As a special case, when the number of CPUs is fixed for the PRAM, i.e., $m_i = m$ for any $i \in [T]$, an oblivious simulation overhead of χ means that the OPRAM needs to run in $\chi \cdot T$ steps consuming m CPUs (same as the original PRAM) where T is the runtime of the original PRAM.

An oblivious simulation overhead of χ also implies the OPRAM's CPU-to-memory bandwidth overhead is a factor of χ more than the original PRAM. Since our model simulates all inter-CPU communication with memory-to-CPU communication, an OPRAM with simulation overhead χ under our model immediately implies that the inter-CPU communication is bounded by χ too. In this sense, our metrics are stronger than those adopted in earlier work [6] which treated CPU-to-memory communication and inter-CPU communication separately — this makes our upper bound results more general.

Assumption on varying number of CPUs. Without loss of generality, henceforth in the paper we may assume that in the original PRAM, the number of CPUs in adjacent steps can increase arbitrarily, but may only decrease by a factor of 2. In other words, we may assume that for any $i \in [T-1]$, $m_{i+1} \geq \frac{m_i}{2}$. This assumption is without loss of generality, since it is not hard to see that *any PRAM where the number of CPUs can vary arbitrarily can be simulated by a PRAM where the number of CPUs can decrease by at most $\frac{1}{2}$ in adjacent steps — and such simulation preserves the PRAM's total work and parallel runtime asymptotically.* Such a simulation is straightforward: if the original PRAM consumes more CPUs than the simulated PRAM in the next step, then the simulated PRAM immediately increases the number of CPUs to a matching number. If the original PRAM's consumes fewer CPUs than the simulated PRAM in the next step, the simulated PRAM decreases its CPUs by at most a factor of 2 each time (and if there are more CPUs in the simulation than needed by the PRAM, the additional CPUs simply idle and perform dummy work).

4 Building Blocks

We now describe some standard or new building blocks that we use.

Oblivious sort. Parallel oblivious sort solves the following problem. The input is an array denoted arr containing n elements and a total ordering over all elements. The output is a sorted array arr' that is constructed obliviously. Parallel oblivious sorting can be achieved in a straightforward way through sorting networks [1], by using $O(n)$ CPUs and consuming $O(n \log n)$ total work and $O(\log n)$ parallel steps.

Oblivious conflict resolution. Oblivious conflict resolution solves the following problem: given a list of memory requests of the form $\mathsf{In} := \{(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i)\}_{i \in [m]}$, output a new list of requests denoted Out also of length m, such that the following holds:

- Every non-dummy entry in Out appears in In;
- Every address addr that appears in In appears exactly once in Out. Further, if multiple entries in In have the address addr, the following priority rule is applied to select an entry: 1) writes are preferred over reads; and 2) if there are multiple writes, a parametrizable function $\mathsf{priority}$ is used to select an entry.

We will use the standard parallel oblivious conflict resolution algorithm described by Boyle et al. [3], which can accomplish the above in $O(m \log m)$ total work and $O(\log m)$ parallel steps. More specifically, Boyle et al.'s conflict resolution algorithm relies on a constant number of oblivious sorts and oblivious aggregation.

Oblivious aggregation for a sorted array. Given an array $\mathsf{Inp} := \{(k_i, v_i)\}_{i \in [n]}$ of (key, value) pairs sorted in increasing order of the keys, we call all elements with the same key a *group*. We say that index $i \in [n]$ is a *representative* of its group if it is the leftmost element of its group. Let Aggr be a commutative and associative aggregation function and we assume that its output range can be described by $O(1)$ number of blocks. The goal of oblivious aggregation is to output the following array:

$$\mathsf{Outp}_i := \begin{cases} \mathsf{Aggr}\left(\{v \,:\, (k, v) \in \mathsf{Inp} \text{ and } k = k_i\}\right), & \text{if } i \text{ is a representative;} \\ \perp, & \text{o.w.} \end{cases}$$

Boyle et al. [3] and Nayak et al. [26] show that oblivious aggregation for a sorted array of length n can be accomplished in $O(\log n)$ parallel time consuming n CPUs.

When the input array has a maximum group size of k, we show that oblivious aggregation can be accomplished in $O(\log k)$ parallel steps consuming $O(\frac{n}{\log k})$ CPUs. We defer the detailed description of the algorithm to Appendix A.

Oblivious routing. Oblivious routing solves the following problem. Suppose n source CPUs each holds a data block with a distinct key (or a dummy block). Further, n destination CPUs each holds a key and requests a data block identified by its key. An oblivious routing algorithm routes the requested data block to the destination CPU in an oblivious manner. Boyle et al. [3] showed that through a combination of oblivious sorts and oblivious aggregation, oblivious routing can be achieved in $O(n \log n)$ total work and $O(\log n)$ parallel runtime.

In this paper, we sometimes also need a variant of the oblivious routing algorithm, a source CPU gets informed in the end whether its block is successfully routed to one or more destination CPUs. We elaborate how to modify Boyle et al. [3]'s oblivious routing building block to accomplish this.

Oblivious bin packing. Oblivious bin packing is the following primitive. We are given B bins each of capacity Z, and an input array of possibly dummy elements where each real element is

tagged with a destined bin number and priority value. We wish to maximally pack each bin with elements destined for the bin — if there are more than Z elements destined for a bin, the Z elements with the highest priority should be chosen. Let n be the size of the input array, In the end, the algorithm outputs an array of size $B \cdot Z$ denoting the packed bins and an array of size n denoting the remaining elements — both padded with dummies.

Let $\widehat{n} := \max(n, B \cdot Z)$. We devise an algorithm for performing such oblivious bin packing in $O(\log \widehat{n})$ parallel steps consuming $\widehat{n}$ CPUs. The details of this algorithm and a more formal definition of oblivious bin packing are deferred to Appendix A.

5 Our Basic OPRAM Construction

We now describe our basic OPRAM construction.

5.1 Notations

Addresses in each recursion level. Recall that we reviewed the Circuit ORAM construction earlier. Here we define some notations for expressing recursion levels, including given each logical memory request, which metadata blocks to fetch from each recursion level.

In the presentation below, we assume that each position map block can store the position labels of γ blocks at the next recursion level, i.e., the *branching factor* is denoted by γ. Given a logical address addr of a data block, we say that its level-d prefix (denoted $\mathsf{addr}^{\langle d \rangle}$) is the d most significant characters of addr when expressed in base-γ format. Specifically, a block at address $\mathsf{addr}^{\langle d \rangle}$ in recursion level d will store the position labels for the γ blocks at addresses $\{(\mathsf{addr}^{\langle d \rangle}||j) : j \in [\gamma]\}$ in recursion level $d+1$; we say that the level-$(d+1)$ address $(\mathsf{addr}^{\langle d \rangle}||j)$ is the jth child of the level-d address $\mathsf{addr}^{\langle d \rangle}$. For the special case $\gamma = 2$, we sometimes refer to the level-$(d+1)$ addresses $(\mathsf{addr}^{\langle d \rangle}||0)$ and $(\mathsf{addr}^{\langle d \rangle}||1)$ as the *left child* and the *right child* respectively of the level-d address $\mathsf{addr}^{\langle d \rangle}$.

Example 1. *We give an example for $\gamma = 2$, i.e., when each position map block can store exactly two position labels. Imagine that one of the memory requests among the batch of m requests asks for the logical address $(0101100)_2$ in binary format. For this request,*

- *A fetch CPU at recursion level 0 will look for the level-0 address $(0*)$, and the fetched block will contain the position labels for the level-1 addresses $(00*)$ and $(01*)$; and a corresponding fetch CPU at recursion level 1 will receive the position label for the level-1 address $(01*)$.*
- *A fetch CPU at recursion level 1 will look for the level-1 address $(01*)$, and the fetched block will contain the position labels for the level-2 addresses $(010*)$ and $(011*)$; and a corresponding fetch CPU at the next recursion level is to receive the position label for $(010*)$;*
- *This goes on until the final recursion level is reached. Except for the final recursion level which stores actual data blocks, all other recursion levels store position map blocks.*

Here we focused on what happens for fetching one logical address $(0101100)_2$ — but keep in mind that there are m such addresses in a batch and thus the above process is repeated m times in parallel.

Notations for varying number of CPUs. For simplicity, below we use m (omitting the subscript t) to denote the number of CPUs of the present PRAM step; we use the notation $\widehat{m}$ to denote the number of CPUs in the previous PRAM step. Without loss of generality, we also assume that both

m and $\widehat{m}$ are powers of 2, since if not, we can always round it to the nearest power of 2 while incurring only a constant factor blowup. Recall that due to our bounded change assumption on the number of CPUs, we may also assume without loss of generality that $m \geq \frac{\widehat{m}}{2}$. Therefore, if $m < \widehat{m}$ it must be the case that $m := \frac{\widehat{m}}{2}$.

Our OPRAM scheme will try to maintain the following invariant: at the end of a PRAM step with m CPUs, the OPRAM data structure will have exactly $2m$ disjoint subtrees. Henceforth, we assume that at the beginning of the PRAM step we are concerned about, there are exactly $2\widehat{m}$ disjoint subtrees since $\widehat{m}$ denotes the number of CPUs in the previous PRAM step.

Parameter α. Throughout the description, we use $\alpha = \omega(1)$ to denote an appropriately small super constant function in N such that the failure probability is at most $\frac{1}{N^{\Theta(\alpha)}}$, i.e., negligible in N.

5.2 Data Structures

Subtrees and overflowing pool. For each of the recursion levels, we maintain a binary tree structure as in Circuit ORAM [34]. We refer the reader to Section 2.1 for a review of the Circuit ORAM algorithm. However, instead of having a complete tree, our OPRAM scheme truncates the tree at height $\ell := \log_2(2m)$ containing $2m$ buckets. In this way, we can view the tree data structure as $2m$ disjoint subtrees.

In the Circuit ORAM algorithm, all buckets with heights smaller than ℓ contain at most $O(m + \alpha \log N)$ blocks. In our OPRAM scheme, these blocks are treated as overflowing blocks, and they are held in an overflowing data structure called a *pool* as described below.

Position map. As in Circuit ORAM (see Section 2.1), each address addr is associated with a random path in one of the subtrees, and the path is identified by a leaf node. We use a position map $\mathsf{posmap}[\mathsf{addr}]$ to store the position identifier for address addr.

Our main *path invariant* states that a block with address addr must reside on the path to the leaf $\mathsf{posmap}[\mathsf{addr}]$ in one of the subtrees, or reside in the overflowing pool. When block addr is accessed (via read or write), its position $\mathsf{posmap}[\mathsf{addr}]$ will be updated to a new leaf chosen uniformly and independently at random. As in previous works [30, 33, 34], the position map is stored in a smaller OPRAM recursively. We use the notation $\mathsf{pos\text{-}OPRAM}$s to denote all recursion levels for storing the position map, and we use $\mathsf{data\text{-}OPRAM}$ to denote the top recursion level for storing data blocks.

5.3 Overview of One Simulated PRAM Step

To serve each batch of memory requests, a set of CPUs interact with memory in two synchronized phases: in the *fetch* phase, the request CPUs receive the contents of the requested blocks; in the second *maintain* phase, the CPUs collaborate to maintain the data structure to be ready for the next PRAM step. The description below can be regarded as an expanded version of Section 2.4. In particular, we now spell out what happens if m_t varies over time. Further, it turns out that for OPRAM, the recursion is somewhat more complicated than ORAM, we also spell out all the details of the recursion — this choice is made also partly in anticipation of the additional computational security techniques described later in Section 11 where it is somewhat important to not treat the recursion as a blackbox like most earlier tree-based ORAM/OPRAM works [3, 6, 7, 33, 34]. Our algorithm below employs several subroutines the details of which will be expanded in Sections 6 and 7 respectively.

Fetch phase. The fetch phase has an array of m addresses as input denoted $(\mathsf{addr}_1, \ldots, \mathsf{addr}_m)$. Recall that at the beginning of the fetch phase, each recursion level has $2\widehat{m}$ disjoint subtrees, where $\widehat{m}$ is the number of active CPUs in the previous PRAM step.

(i) *Preparation: all recursion levels in parallel.* For all recursion levels $d := 0, 1, \ldots, D$ in parallel, perform the following:

- *Generate level-d prefix addresses.* Write down the level-d prefixes of all m requests addresses $(\mathsf{addr}_1, \ldots, \mathsf{addr}_m)$. Clearly, this step can be accomplished in $O(1)$ parallel step with m CPUs.
- *Conflict resolution.* Given a list of m possibly dummy level-d addresses denoted $(\mathsf{addr}_1^{\langle d \rangle}, \ldots, \mathsf{addr}_m^{\langle d \rangle})$, we run an instance of the oblivious conflict resolution algorithm to suppress duplicate requests (and pad the resulting array with dummies). This step can be accomplished in $O(\log m)$ parallel steps with m CPUs.
- *Discover which children addresses are needed by the next recursion level.* Let $\mathsf{Addr}^{\langle d \rangle} := \{\mathsf{addr}_i^{\langle d \rangle}\}_{i \in [m]}$ denote the list of level-d addresses after conflict resolution. Each of these m level-d addresses has γ children addresses in the next recursion level. By jointly examining $\mathsf{Addr}^{\langle d \rangle}$ and $\mathsf{Addr}^{\langle d+1 \rangle}$, recursion level d learns for each non-dummy $\mathsf{addr}_i^{\langle d \rangle} \in \mathsf{Addr}^{\langle d \rangle}$, which of its children are needed for the next recursion level (see Section 6.1 for details of this subroutine). At the end of this step, each of the m level-d addresses receives a bit vector containing γ bits, indicating whether each child address is needed by the next recursion level. As mentioned in Section 6.1, this can be accomplished through $O(1)$ number of oblivious sorts. Therefore, it takes m CPUs $O(\log m)$ steps to complete.
- *Choose fresh position labels for the next recursion level.* For any child that is needed, recursion level d chooses a new position label for the next recursion level. For recursion level d, the result of this step is a new position array

$$\{\mathsf{addr}_i^{\langle d \rangle}, (\mathsf{npos}_j : j \in [\gamma])\}_{i \in [m]}$$

  where npos_j is a fresh random label in level $d+1$ if $\mathsf{addr}_i^{\langle d \rangle} || j$ is needed in the next recursion level, otherwise $\mathsf{npos}_j := \perp$. Later in our algorithm, each recursion level d will inform the next recursion level $d+1$ of the chosen new position labels.

  This step can be accomplished in $O(\gamma)$ steps with m CPUs — for our statistically secure OPRAM scheme, we shall assume $\gamma = O(1)$.
- *Pool lookup.* We have m CPUs each of which now seeks to fetch the level-d block at address $\mathsf{addr}^{\langle d \rangle}$. The m CPUs first tries to fetch the desired blocks inside the central pool; and at the end, the fetched blocks will be marked as dummy in the pool.

  If $m \geq \alpha \log \log N$, then we rely on an instance of the oblivious routing algorithm, such that each of these m CPUs will attempt to receive the desired block from the pool. If $m < \alpha \log \log N$, oblivious routing is too expensive, instead we invoke a special-case algorithm for small m to accomplish this in $O(\alpha \log N)$ steps with m CPUs.

  We defer the details of the algorithm to Section 6.

(ii) *Fetch: level by level.* Now, for each recursion level, m CPUs will each look for a block in one of the subtrees. This step must be performed sequentially one recursion level at a time since each recursion level must receive the position labels from the previous level before looking for blocks in the subtrees.

For each recursion level $d = 0, 1, \ldots D$ in sequential order, we perform the following:

- *Receive position labels from previous recursion level.* Unless $d = 0$ in which case the position labels can be fetched in $O(1)$ parallel step, each of the m level-d addresses will receive a pair of position labels from the previous recursion level denoted $(\mathsf{pos}, \mathsf{npos})$, where pos represents the tree path to look for the desired block, and npos denotes a freshly chosen label to be assigned to

the block after the fetch is complete. This can be accomplished through an instance of oblivious routing consuming $O(\log m)$ parallel steps with m CPUs.

- *Subtree lookup.* Now, each of the m CPUs receives an instruction of the form $(\mathsf{addr}^{\langle d \rangle}, \mathsf{pos})$ that could be possibly dummy. Each CPU will now read a tree path leading to the leaf node numbered pos, in search of the block with logical address $\mathsf{addr}^{\langle d \rangle}$ (but without removing the block). If found, the CPU will remember the location where the block is found — and this information will later be useful for the simultaneous removal step that is part of the maintain phase. If a CPU receives a dummy instruction, it simply scans through a randomly chosen path in a random subtree.

  At this moment, each of the m CPUs has fetched the desired block either from the pool or the tree path (or the CPU has fetched dummy if it received a dummy instruction to start with). The fetched position labels (as well as the new position labels chosen for the next recursion level) are ready to be routed to the next recursion level.

(iii) *Oblivious multicast: once per batch of requests.* Finally, when the $\mathsf{data\text{-}OPRAM}$ has fetched all requested blocks, we rely on a standard oblivious routing algorithm (see Section 4) to route the resulting blocks to the request CPUs. This step takes $O(\log m)$ parallel time with m CPUs.

Remark 1. Note that in the above exposition, we made explicit which steps can be parallelized across recursion levels and which steps must be performed sequentially across recursion levels — in particular, the level-to-level position label routing must be performed sequentially across recursion levels since the next recursion level must receive the position labels before learning which tree paths to traverse. Although this distinction may not be very useful in this paper, it will turn out to be important in a companion paper by Chan et al. [5], where the authors further parallelize the level-to-level routing algorithm. In particular, Chan et al. [5] introduce a new and better notion of an OPRAM's "depth" by assuming that the OPRAM can consume more CPUs than the original PRAM. In this case, they show that an OPRAM's depth can be made asymptotically smaller by further parallelizing Circuit OPRAM's level-to-level routing algorithm.

Maintain phase. All of the following steps are performed in parallel across all recursion levels $d = 0, 1, \ldots, D$:

(i) *Simultaneous removal of fetched blocks from subtrees.* After each of the m CPUs fetches its desired block from m tree paths, they perform a simultaneous removal procedure to remove the fetched blocks from the tree paths. This step can be accomplished in $O(\log N)$ parallel steps using m CPUs. We defer a detailed description of this new simultaneous removal subroutine in Section 7.1.

(ii) *Passing updated blocks to the pool.* Each CPU updates the contents of the fetched block — if the block belongs to a position map level, the block's content should now store the new position labels (for the next recursion level) chosen earlier in the preparation phase. Further, each block will be tagged with a new position label that indicates where the block can now reside in the current recursion level — this position label was received earlier from the previous recursion level during the fetch phase (recall that each recursion level chooses position labels for the next recursion level).

  The updated blocks are merged into the pool. The pool temporarily increases its capacity to hold these extra blocks, but the extra memory will be released at the end of the maintain phase during a cleanup operation.

(iii) *Increasing the number of subtrees if necessary.* At this moment, if $m > \widehat{m}$, i.e., if the number of CPUs has increased since the last PRAM step, then we increase the number of subtrees to $2m$, and merge all smaller heights of the tree into the pool. If the number of CPUs has decreased (by a factor of 2) since the last PRAM step, we will handle this case later.

(iv) *Selection of eviction candidates.* Following the deterministic, reverse-lexicographical order eviction strategy of Circuit ORAM [34], we choose the next $2m$ eviction paths (pretending that all subtrees are part of the same big ORAM tree). The $2m$ eviction paths will go through $2m$ subtrees henceforth referred to as *evicting subtrees*. If m has decreased (by a factor of 2) since the last PRAM step, then not all subtrees are evicting subtrees.

We devise an eviction candidate selection algorithm that will output one (possibly dummy) block to evict for each evicting subtree, as well as the remainder of the pool (with these selected blocks removed). The block selected for each evicting subtree is based on the *deepest* criterion with respect to the current eviction path. When $m \geq \alpha \log \log N$, we rely on oblivious sorting to accomplish this in $O(\log N)$ parallel steps with m CPUs. When $m < \alpha \log \log N$, oblivious sorting will be too expensive, so we rely on a different algorithm to accomplish this step in $O(\alpha \log N)$ parallel steps with m CPUs. We defer a detailed description of the algorithm to Section 7.2.

(v) *Eviction into subtrees.* In parallel, for each evicting subtree, the eviction algorithm of Circuit ORAM [34] is performed for the candidate block the subtree has received. The straightforward strategy takes $O(\log N)$ parallel steps consuming m CPUs.

After the eviction algorithm completes, if the candidate block fails to be evicted into the subtree, it will be returned to the pool; otherwise if the candidate block successfully evicts into the subtree, a dummy block is returned to the pool.

(vi) *Decreasing the number of subtrees if necessary.* If $m < \widehat{m}$, this means that the number of CPUs has decreased since the previous PRAM step. Also note that in this case, by assumption, it must be the case that $m = \frac{\widehat{m}}{2}$. At this moment, we will halve the number of subtrees by reconstructing one more height of the big Circuit ORAM tree containing $2m$ buckets. Let Z be the bucket size of the ORAM tree. To reconstruct a height of size $2m$, we must reconstruct $2m$ buckets each of size Z. This can be achieved by repeating the eviction candidate selection algorithm Z number of times (see Section 7.3 for details).

(vii) *Cleanup.* Finally, since the pool size has grown in the above process, we perform a compression procedure to remove dummy blocks and compress the pool back to $c \cdot m + \alpha \log N$ size. Probabilistic analysis in Section 8 shows that the pool occupancy is bounded by $c \cdot m + \alpha \log N$ except with $\mathsf{negl}(N)$ probability, and thus ensures that no real blocks are lost during this reconstruction with all but negligible probability.

Again, if $m \geq \alpha \log \log N$, this can be accomplished through a simple oblivious sort procedure in $O(\log N)$ steps with m CPUs. Else if $m < \alpha \log \log N$, we devise a different procedure to perform the pool cleanup that completes in $O(\alpha \log N)$ parallel steps consuming m CPUs.

6 Details of the Fetch Phase

The outline of the fetch phase was described in Section 5. Almost all steps are self-explanatory as described in Section 5, and it remains to spell out only a couple subroutines of the preparation stage.

6.1 Discovering Which Children Addresses are Needed

Recall that during the preparation stage, for each recursion level, each conflict resolved address wants to learn which of its γ child addresses are needed by the next recursion level. Henceforth we assume that $\gamma = O(\log N)$.

We can accomplish this task using the following algorithm. We use $\mathsf{Addr}^{\langle d \rangle}$ to denote an array of size m that contain the conflict resolved (possibly dummy) addresses for recursion level d.

For each recursion level $d = 0, 1, \ldots, D-1$ in parallel:

- Let X be the concatenation of $\mathsf{Addr}^{\langle d \rangle}$ and $\mathsf{Addr}^{\langle d+1 \rangle}$ where each element additionally carries a tag denoting whether it comes from $\mathsf{Addr}^{\langle d \rangle}$ or $\mathsf{Addr}^{\langle d+1 \rangle}$.
- Oblivious sort X such that the addresses from $\mathsf{Addr}^{\langle d \rangle}$ always appear immediately before its up to γ children addresses that come from $\mathsf{Addr}^{\langle d+1 \rangle}$ — henceforth we say that these addresses share the same key. Let the resulting array be X'.
- Invoke an instance of the oblivious aggregation algorithm, such that each address in X' that comes from $\mathsf{Addr}^{\langle d \rangle}$ receives a (compacted) bit vector indicating whether each of its γ children is needed in the next recursion level. Notice that as long as $\gamma := O(\log N)$, the resulting bit vector can be packed in a single block.
- For each element of the resulting array in parallel, if the element comes from $\mathsf{Addr}^{\langle d+1 \rangle}$, mark it as dummy. Let the resulting array be denoted Y.
- Obliviously sort the resulting array Y such that all dummy elements are pushed to the end. Output $Y[1:m]$.

Clearly, the above algorithm can be completed in $O(\log m)$ steps with m CPUs.

6.2 Fetching and Removing Blocks from the Pool

Recall that another step of the preparation stage is to look for desired blocks from the pool and then remove any fetched block from the pool (by marking it as dummy). To achieve this, we consider two cases — and recall that the pool size is upper bounded by $O(m + \alpha \log N)$ except with negligible probability due to our probabilistic analysis in Section 8.

- **Case 1:** $m \geq \alpha \log \log N$**.** In this case, we simply invoke an instance of the oblivious routing algorithm (particularly, the variant that removes routed elements from the source array) to accomplish this. This step can be completed in $O(\log N)$ parallel steps consuming m CPUs for an appropriately small super-constant $\alpha = \omega(1)$.
- **Case 2:** $m < \alpha \log \log N$**.** In this case, oblivious sorting would be too expensive. Therefore, we instead adopt the following algorithm. Recall that in this case, the pool size is dominated by $O(\alpha \log N)$.
  - First, each of the m CPUs perform a linear scan of the pool to look for its desired block.
  - Next, all m CPUs perform a a pipelined linear scan of the pool. During the linear scan, each CPU marks its fetched block (if any) as dummy. To ensure no write conflicts, we require that CPU number i starts its scan in the i-th step, i.e., in a pipelined fashion.

  Clearly, the above algorithm can be accomplished in $O(m + \alpha \log N)$ parallel steps consuming m CPUs.

6.3 Performance of the Fetch Phase

Taking into account the cost of all steps of the fetch phase, we have the following lemma.

Lemma 1 (Performance of the fetch phase)**.** *Suppose that the block size $B = \Omega(\log N)$. Then, to serve the batch of m requests, the fetch phase, over all $O(\log N)$ levels of recursion, completes in $O(\log^2 N)$ parallel steps with m CPUs when $m \geq \alpha \log\log N$; and in $O(\alpha \log^2 N)$ parallel steps when $m < \alpha \log\log N$.*

7 Details of the Maintain Phase

An overview of the maintain phase was provided in Section 5.3. It remains to spell out the details of various subroutines needed by the maintain phase.

7.1 Simultaneous Removal of Fetched Blocks from Subtrees

Problem definition. Suppose that there are m fetch paths for each batch of m memory requests. Simultaneous removal provides the following abstraction:

- *Inputs:* Each of m CPUs has a tuple of the form (pathid_i, s_i) or $\perp$. More specifically, $\perp$ denotes nothing to remove, or else
  - pathid_i denotes the leaf identifier of a random tree path containing $O(\log N)$ slots. In particular, a tree path contains $O(\log N)$ heights and each height contains $O(1)$ slots; and
  - s_i denotes a slot in the tree path to remove a block from;

  Note that each tree path is random such that each disjoint subtree may receive at most $\alpha \log N$ tree paths. Although the m paths, we are guaranteed that all the non-dummy inputs of the m CPUs must correspond to distinct slots, i.e., no two CPUs want to remove from the same slot.
- *Outputs:* Each of the m CPUs outputs an array of length $O(\log N)$, denoting for each slot on its path: 1) whether the CPU is the representative CPU; and 2) if so, whether the block in the slot needs removal. The outputs should maintain the following invariants: every slot on the m input paths has exactly one representative, and if some CPU wanted to remove the block in the slot, then the representative is informed of the removal instruction.

Note that given the above output, each CPU simply carry out the instruction for every physical slot it is representative for:

- if the instruction is to remove, the CPU reads the block and writes dummy back;
- if the instruction is not to remove, the CPU reads the block and writes the same block back;
- if the CPU is not a representative for this physical slot, do nothing for this slot.

Simultaneous removal algorithm. We describe our simultaneous removal algorithm below. We note that since all the fetch paths are already observable by the adversary, it is okay for us to employ a non-oblivious propagation algorithm.

- *Sorting fetch paths.* All m CPUs write down their input tuple, forming an array of size m. We now obliviously sort this array by their fetch path such that the leftmost fetch path appears first, where the other of the fetch paths are determined by the leaves they intersect. This step takes $O(m \log m)$ total work and $O(\log m)$ parallel steps.

- *Table creation.* In parallel, fill out a table Q where each row corresponds to a slot in the tree, and each column corresponds to a fetch path (in sorted order from left to right). Specifically, $Q[\ell][i] = 1$ if the i-CPU wants to remove the block in slot ℓ on its fetch path; else $Q[\ell][i] = 0$. It is not hard to see that this step can be completed in $O(1)$ parallel steps with $m \log N$ CPUs.

  Notice that since the m fetch paths may overlap, table Q may contain entries that correspond to the same physical slot. However, since the fetch paths were sorted from left to right, all entries corresponding to the same physical slot must appear in consecutive locations in the same row. Further, it is not hard to see that except with negligible probability, at most $\alpha \log N$ entries in Q correspond to the same physical slot (since each disjoint subtree receives at most $\alpha \log N$ fetch paths except with negligible probability).

  Henceforth we say that $Q[\ell][i]$ is a *representative* if $Q[\ell][i]$ is the first occurrence of a physical slot in the row $Q[\ell]$.

- *Oblivious aggregation.* Now, for each row of the table Q, invoke an instance of the oblivious aggregation algorithm (for bounded-size groups) such that the representative of each group learns the OR of all entries belonging to the group. As mentioned above, since the group size is bounded by $\alpha \log N$, we can complete such oblivious aggregation in $O(\log \log N)$ parallel steps with $\frac{m}{\log \log N}$ CPUs, or alternatively, in $O(\log N)$ steps with $\frac{m}{\log N}$ CPUs.

  Therefore, over all rows of the table Q, this step completes in $O(\log N)$ parallel steps with m CPUs.

7.2 Evictions

Recall that in the sequential Circuit ORAM [34], whenever a fetched (and possibly updated) block is added to the root, two path evictions must be performed. The goal of Circuit OPRAM is to simulate the stochastic process of Circuit ORAM. However, since Circuit OPRAM does not maintain the tree structure for lower heights of the tree, we only need to partially simulate Circuit ORAM's stochastic process for the $O(m)$ disjoint subtrees that Circuit OPRAM does maintain. Our algorithms described below make use of certain non-blackbox properties of the Circuit ORAM algorithm [34]. In our description below, we will point out these crucial properties as the need arises, without re-explaining the entire Circuit ORAM construction [34].

Select $2m$ distinct eviction paths in $2m$ distinct subtrees. At this point, a batch of m requests have been made, and m possibly dummy blocks have been fetched, possibly update, and merged into the pool. As mentioned earlier, we now consider the pool as a flattened data structure containing all the smaller levels of the big Circuit ORAM tree as well as the stash. To simulate Circuit ORAM's stochastic process, at this point we must perform $2m$ evictions on $2m$ distinct paths. We leverage Circuit ORAM's deterministic, reverse-lexicographical order for determining the next $2m$ eviction paths. The specifics of the eviction path selection criterion is not important here, and the reader only needs to be aware that this selection criterion is fixed a priori and data independent. For more details on eviction path selection, we refer the reader to Circuit ORAM [34].

Fact 2. *Observe that at this point, the number of disjoint subtrees is at least $2m$. Due to Circuit ORAM's eviction path selection criterion, all $2m$ eviction paths will not only be distinct, but also correspond to distinct subtrees — henceforth we refer to these subtrees as evicting subtrees. For the special case when m stays the same over time, all $2m$ subtrees are evicting subtrees, and exactly one path is evicted in each subtree.*

Select $2m$ eviction candidates. We will now leverage a special property of the Circuit ORAM's eviction algorithm described earlier by Fact 1 such that we perform a "partial eviction" only on

the subtrees maintained by our Circuit OPRAM. Recall that Fact 1 says the following:

- For Circuit ORAM's eviction algorithm, at most one block passes from $\mathsf{path}[:i]$ to $\mathsf{path}[i+1:]$ for each height i on the eviction path denoted path. In this case we also say that the block passes through the boundary between height i and height $i+1$.
- Moreover, if a block does pass through the boundary between height i and height $i+1$, it must be the block that is *deepest* with respect to the eviction path, where *deepest* is a criterion defined by Circuit ORAM [34]. Intuitively, a block is deeper if it can reside in a bucket on the eviction path with higher height. The reader can refer to Circuit ORAM [34] for details.

Therefore, we only need to elect one candidate block from the pool for each of the $2m$ eviction paths on which we would like to perform eviction. We describe an algorithm for performing such selection based on two different cases:

- **Case 1: when $m > \alpha \log\log N$.** We devise an algorithm based on a constant number of oblivious sorts. Since the pool contains $O(m + \alpha \log N)$ blocks, this algorithm completes in $O(\log N)$ parallel steps with m CPUs.

(a) In the beginning, each block in the pool is tagged with the block's position identifier. Now, for each block in the pool in parallel, compute and tag the block with the additional metadata $(\mathsf{treeid}, \mathsf{priority})$ which will later be used as a sort key:
   - treeid denotes the block's destined subtree if the destined subtree is an evicting subtree, otherwise $\mathsf{treeid} := \perp$. All dummy blocks have $\mathsf{treeid} := \perp$.
   - $\mathsf{priority}$ denotes the block's eviction priority within the subtree. The block's $\mathsf{priority}$ value can be computed based on the block's position identifier and the current eviction path (in the subtree identified by treeid), a higher priority is assigned to blocks that can be reside deeper (i.e., closer to leaf) along the eviction path. The definition of *deep* is the same as in Circuit ORAM [34].

(b) Now, invoke an instance of the oblivious bin-packing algorithm, where each evicting subtree can be regarded as a bin of capacity 1, and all the blocks are balls tagged with its destination bin. We wish to place one ball into each bin — if multiple balls are eligible for a bin, we prefer to place the ball with a higher $\mathsf{priority}$ value. The output of the algorithm is one (possibly dummy) eviction candidate for each evicting subtree, as well as the remainder of the pool minus those chosen blocks.

- **Case 2: when $m \le \alpha \log\log N$.** In this case, the pool contains $\Theta(\alpha \log N)$ blocks, and performing oblivious sort will cause a total work of $\Omega(\log N \log\log N)$, which is too expensive if m is small. Instead, we perform the following, which can be accomplished in $O(\alpha \log N)$ parallel steps with $2m$ CPUs — below we describe the algorithm assuming $2m$ CPUs, but clearly the algorithm also works with m CPUs since we can always have each CPU simulate $O(1)$ CPUs.

(a) Assign one CPU for each of the $2m$ eviction paths. Each CPU linearly scans through the pool and selects the deepest element with respect to the eviction path. If no element is eligible for the current eviction path, a dummy element is selected. Clearly, this incurs $O(\alpha \log N)$ parallel steps. This step outputs an array of $2m$ elements selected for eviction for each of the $2m$ eviction paths. The rest of the algorithm will output the remainder of the pool.

(b) In $O(\alpha \log N + m)$ parallel steps, the $2m$ CPUs make a "pipelined linear scan", where CPU i starts its linear scan in the i-th step (note that this avoids write conflicts). When each CPU is making a linear scan, if the (real) block is what the CPU has selected for eviction, replace it with dummy; otherwise, write the original block back.

(c) Output the resulting pool.

Evictions. At this point, each of the $2m$ eviction paths has received one candidate block (which can be dummy). Hence, these $2m$ evictions can be carried out in parallel, each according to the (sequential) eviction algorithm of Circuit ORAM [34]. More specifically, we first expand the capacity of each eviction path by adding a bucket at the beginning of the path that holds the eviction candidate selected earlier; we call this the *smallest bucket* on the path. We then run Circuit ORAM's (sequential) eviction algorithm on each of these $2m$ (expanded) paths in parallel.

At the end, the block in the smallest bucket on each eviction path is returned to the pool. Note that if the eviction candidate has been successfully evicted into the path, then the smallest block on the path would be dummy, and thus a dummy block is returned to the pool. Doing this according to Circuit ORAM's eviction algorithm [34] takes $O(\log N)$ parallel runtime with $2m$ CPUs.

In a final cleanup step described later, we suppress a subset of the dummy blocks in the pool such that the pool size will not keep growing.

7.3 Data Structure Cleanup

Adjusting the number of subtrees. If $\widehat{m} > m$, i.e., the number of CPUs has decreased (by a factor of 2 according to our assumption) since the last PRAM step, we will halve the number of subtrees. This means that we must reconstruct one more height of the big Circuit ORAM tree.

Let $Z = O(1)$ be the bucket size of the ORAM tree. To reconstruct a height of size $2m$, we must reconstruct $\widehat{m}$ buckets each of size Z. To achieve this, we invoke an instance of the oblivious bin packing algorithm, where we wish to pack $2m$ buckets each of capacity Z. If a block can legally reside in a bucket by Circuit ORAM's path invariant, it is deemed eligible for a bucket. If more than Z blocks are eligible for a bucket, we break ties arbitrarily. Such oblivious bin packing can be completed in $O(\log m)$ parallel steps with m CPUs.

Although the reconstructed height of the big ORAM tree may contain different blocks from the scenario had we maintained the whole tree from the start, later in Section 8, we will show that the difference is in our favor in the sense that it will not make the pool occupancy larger.

Compress the pool. During the simulation of this PRAM step, the pool size has enlarged by at most $O(m)$. We now compress the pool size by removing a subset of the dummy blocks. There are two cases — recall also that the pool size is bounded by $O(m + \alpha \log N)$ except with negligible probability which we shall formally prove in Section 8:

- **Case 1:** $m \geq \alpha \log \log N$**.** In this case, we can perform such compression through a simple oblivious sort operation that move all dummy blocks to the end of the array representing the pool, and then truncating the array retaining only the first $c \cdot m + \alpha \log N$ blocks for an appropriate constant c. This can be completed in $O(\log N)$ parallel steps with m CPUs.

- **Case 2:** $m < \alpha \log \log N$**.** In this case, oblivious sorting would be too expensive. Instead, we perform compression by conducting a pipelined, partial bubble sort. Let $s = O(m + \alpha \log N)$ be the current pool size, and suppose that we need to compress the array back to $s' := s - O(m)$ blocks. Recall that a normal bubble sort of s elements would make s bubbling passes over the array, where after the i-th pass, the largest i elements are at the end. Here we make only $O(m)$ bubbling passes where each CPU is in charge of $O(1)$ passes. The passes are performed in a pipelined fashion to avoid write conflicts. At the end of this partial bubble sort, the last $s - s'$ blocks of the array may be removed.

  This is completed in $O(m + \alpha \log N)$ parallel steps with m CPUs.

7.4 Performance of the Maintain Phase

Accounting for the cost of all of the above steps, we can easily derive the following lemma for the performance of the maintain phase.

Lemma 2 (Performance of the maintain phase)**.** *Suppose that the block size $B = \Omega(\log N)$. Then, to serve the batch of m requests, the maintain phase, over all $O(\log N)$ levels of recursion, completes in $O(\log^2 N)$ parallel steps with m CPUs when $m \geq \alpha \log\log N$; and in $O(\alpha \log^2 N)$ parallel steps when $m < \alpha \log\log N$.*

8 Stochastic Analysis

Recall that in the maintain phase, the pool increases its capacity temporarily to receive the newly updated blocks. However, at the end of simulating each PRAM step, a pool cleanup operation is performed and the temporary extra memory is released. In this section, we prove a stochastic bound to show that except with negligible probability, none of the real blocks are lost during this cleanup process due to the possibility of overflowing the pool whose size bound is set a-priori.

8.1 Proof Roadmap

First, we note that the parallel nature of our OPRAM algorithm is not relevant to the stochastic bounds. So for the purpose of this section, one can simply regard our OPRAM algorithms as being sequential. Recall that our Circuit OPRAM algorithm is designed to maximally resemble the stochastic process of Circuit ORAM [34] — the Circuit ORAM algorithm has a rather involved stochastic analysis and we would like to maximally reuse those analysis. In the remainder of this section, we will focus on the difference between the stochastic processes induced by Circuit OPRAM and Circuit ORAM, i.e., whether operations are "batched". We will show that Circuit OPRAM's batched nature of operations only helps with reducing overflows.

More specifically, we will adopt the following proof roadmap:

1. We define a modification to Circuit ORAM called Batched Circuit ORAM. In Batched Circuit ORAM, a batch of $2m$ evictions are performed after making m fetches; whereas in Circuit ORAM, we interleave every fetch and every two evictions.

2. The simpler case is that the number of CPUs m never decreases in the PRAM. We then prove that in this case, for every request sequence and every random string consumed by the ORAM/OPRAM algorithm, Circuit OPRAM's pool occupancy would be the same as the number of blocks in Batched Circuit ORAM's stash and all buckets below (i.e., closer to the root) height $\log_2(2m)$. Therefore, analyzing Circuit OPRAM's pool occupancy translates to analyzing Batched Circuit ORAM's stash usage.

3. Then, we show that Batched Circuit ORAM's stash usage is stochastically dominated by that of Circuit ORAM — in this way, we can reuse Circuit ORAM's stash analysis to upper bound Batched Circuit ORAM's stash size.

4. Finally, we analyze the scenario when the number m of PRAM's CPUs decreases (by a factor of 2) and the OPRAM algorithm needs to reconstruct an extra height of the big Batched Circuit ORAM tree. Although the blocks in the reconstructed height in the Circuit OPRAM may not agree with those in the corresponding height in the Batched Circuit ORAM, such reconstruction is "more aggressive" than Batched Circuit ORAM's eviction, and thus stash utilization can only decrease in comparison with Batched Circuit ORAM; we will formalize what this means later.

8.2 Additional Preliminaries

Circuit ORAM eviction and stash usage. Recall that in Circuit ORAM, for every updated block added to the root stash, evictions are performed on two paths, each for the left and the right branch at the root. The details of the eviction algorithm are given in the Circuit ORAM paper [34]. Most of the details are not important for our proof. For the purpose of our proof, we will only need to know a special property of Circuit ORAM's eviction algorithm described earlier (Fact 1) and the following result on the stash usage of Circuit ORAM. We consider Circuit ORAM's stash as part of the root bucket.

Fact 3 (Stash usage in Circuit ORAM [34])**.** *Let R be an integer. After each oblivious access, the probability that Circuit ORAM's stash contains more than R blocks is at most $O(e^{-R})$.*

∞-ORAM. Given an ORAM algorithm X using buckets with finite capacity (where X can be Circuit ORAM or Batched Circuit ORAM, for instance), the ∞-ORAM variant of X is the same as the original algorithm X, except that every bucket has infinite capacity. The concept of an ∞-ORAM is a proof construct (not a real-world efficient ORAM algorithm) that was originally proposed in the Path ORAM work by Stefanov et al. [33], and was later adopted in the proofs of other tree-based ORAMs [34].

8.3 A Variant of the Main Construction

For technical reasons in the proof, we will prove stochastic bounds for a variant of our main construction. Since the modifications are not very interesting, we chose to present our main construction rather than this variant for clarity; moreover, implementing our main construction will give slightly better practical performance (up to a small constant factor) than the variant we prove bounds for.

In this variant construction, we make the following modifications to our main OPRAM construction:

- In the maintain phase, we do not actually remove a fetched block but instead mark it as stale (we can easily modify our simultaneous removal algorithm to mark fetched blocks as stale rather than replacing them with dummies). The difference is that a stale block will occupy a slot in a bucket and cannot simply be dropped like a dummy. The reason is that in the analysis, a fetched block will not create a "hole", which would otherwise make the comparison with the ∞-ORAM more difficult.

  For the purpose of the analysis, a non-dummy block can either be a real or a stale block.

- We imagine that every leaf of every OPRAM subtree has a single phantom child bucket of infinite capacity. A phantom bucket can hold only stale blocks. Note that the phantom child is imaginary, and it does not actually occupy any space in memory.

- When performing eviction on a path leading to a leaf, imagine that we actually perform eviction on the extended path that includes the leaf's phantom child as well. We assume that a stale block destined for some leaf can legally reside in the leaf's phantom child, but a real block will stop before the phantom child.

The ∞-ORAM of our new variant is defined in a similar way by adding a phantom child to every leaf, and by assuming that any stale block destined for a leaf can legally reside in the leaf's phantom child — however, in the ∞-ORAM of our new variant, all buckets, not only the phantom buckets, have infinite capacity.

Remark 2 (A technical note)**.** The Circuit ORAM proof [34] proves stochastic bounds for another variant (different from ours) that requires performing an additional partial eviction on any read path till the height in which a block is fetched and removed. This is needed for the equivalence of the ∞-ORAM and the real ORAM to hold. Unfortunately the same does not work for our Circuit OPRAM. Specifically, we cannot perform partial evictions on read paths since unlike our chosen eviction paths that are one path per disjoint subtree, read paths may actually overlap thus leading to write contentions — at this moment, we do not know a way to parallelize these evictions if the eviction paths intersect. We get around this problem by considering a different variant using stale blocks as described above. If one examines the actual proof of the Circuit ORAM paper [34], the proof actually generalizes to our new variant that does not perform any partial evictions on the read path. More specifically, Circuit ORAM's proof implies the following:

- The strong equivalence of the ∞-ORAM and the real ORAM still holds for our new variant of Circuit ORAM.
- Circuit ORAM's stochastic analysis of the ∞-ORAM holds for the ∞-ORAM of our new variant.

Henceforth in this section, whenever we refer to Circuit ORAM, Batched Circuit ORAM, or Circuit OPRAM, we always refer to the variant of our basic algorithm with these aforementioned modifications.

8.4 Detailed Proof

Batched Circuit ORAM. Recall that for the purpose of our stochastic analysis, it helps to consider an alternative random process which we call "Batched Circuit ORAM". Specifically, it is a variant of the (sequential) Circuit ORAM algorithm using the full complete binary tree. For a batch of m memory requests:

1. We first perform the fetch phase of all m requests sequentially following the Circuit ORAM algorithm, placing all fetched blocks in the root bucket (and therefore the root bucket is allowed to be larger).
2. Then, we perform $2m$ path evictions (also sequentially) following Circuit ORAM's strategy of choosing eviction paths and following its eviction algorithm.

In other words, Batched Circuit ORAM is almost identical to Circuit ORAM, except that Circuit ORAM interleaves fetches and evictions such that each fetch is followed by two evictions; however, here we perform $2m$ evictions after every m fetches, i.e., fetches and evictions are batch-processed.

For Batched Circuit ORAM, Circuit ORAM, or Circuit OPRAM, an execution trace is defined by the pair $(\psi, \vec{S})$, where ψ is a random string that defines position identifiers for all accessed blocks in temporal order, and $\vec{S} := \{S_t : t \in [T]\}$ denotes the memory request sequence.

Circuit ORAM stochastically dominates Batched Circuit ORAM. We now show that Circuit ORAM stochastically dominates Batched Circuit ORAM in terms of stash usage. The intuition is that as opposed to eviction performed in a batch, a block added later in Circuit ORAM cannot benefit from the eviction due to accessing an earlier block. This intuition is formalized in the following Lemma 3.

Let $\overline{\texttt{stash}}(\psi, \vec{S})$ denote the stash usage of Circuit ORAM at the end of an execution trace defined by $(\psi, \vec{S})$. Let $\texttt{stash}(\psi, \vec{S})$ denote the stash usage of Batched Circuit ORAM at the end of an execution trace defined by $(\psi, \vec{S})$.

Lemma 3. *For every ψ and every $\vec{S}$, it holds that*

$$\overline{\texttt{stash}}(\psi, \vec{S}) \geq \texttt{stash}(\psi, \vec{S}).$$

Proof. We consider the ∞-ORAMs of both the Circuit ORAM and the Batched Circuit ORAM. Henceforth, we consider the stash as part of the root bucket, and we use the terminology a "rooted subtree" to denote any subtree in the ORAM tree that contains the root. We use the notation Tr to denote a rooted subtree, and let $|\mathsf{Tr}|$ be the number of non-root buckets in Tr. We use $\overline{\texttt{subtree}}^{\mathsf{Tr}}(\psi, \vec{S})$ to denote the number of non-dummy blocks in Tr after executing the Circuit ORAM's ∞-ORAM over the trace $(\psi, \vec{S})$. Similarly, we use $\texttt{subtree}^{\mathsf{Tr}}(\psi, \vec{S})$ to denote the number of non-dummy blocks in Tr after executing Batched Circuit ORAM's ∞-ORAM over the trace $(\psi, \vec{S})$.

The Circuit ORAM work [34] shows the following useful fact where $Z = O(1)$ denotes the bucket size.

Fact 4 (Equivalence of ∞-ORAM and Circuit ORAM [34])**.** *For every ψ and every $\vec{S}$, let R be the minimum value such that for every rooted subtree Tr, $\overline{\texttt{subtree}}^{\mathsf{Tr}}(\psi, \vec{S}) \leq |\mathsf{Tr}| \cdot Z + R$, then it holds that $\overline{\texttt{stash}}(\psi, \vec{S}) = R$.*

The above fact extends in a straightforward fashion to Batched Circuit ORAM too. Therefore, we have the following:

Fact 5 (Equivalence of ∞-ORAM and Batched Circuit ORAM)**.** *For every ψ and every $\vec{S}$, let R be the minimum value such that for every rooted subtree Tr, $\texttt{subtree}^{\mathsf{Tr}}(\psi, \vec{S}) \leq |\mathsf{Tr}| \cdot Z + R$, then it holds that $\texttt{stash}(\psi, \vec{S}) = R$.*

Due to Facts 4 and 5, it suffices to show that for every ψ and every $\vec{S}$,

$$\texttt{subtree}^{\mathsf{Tr}}(\psi, \vec{S}) \geq \overline{\texttt{subtree}}^{\mathsf{Tr}}(\psi, \vec{S})$$

Let $\overline{\texttt{residue}}^{\mathsf{path},i}(\psi, \vec{S})$ denote the number of non-dummy blocks in $\mathsf{path}[:i]$ that can legally reside in $\mathsf{path}[\min\{i{+}1, L\}:]$ (where L is the height of the leaf buckets) after executing the ∞-Circuit ORAM algorithm over the trace $\psi, \vec{S}$; and similarly let $\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S})$ denote the number of non-dummy blocks in $\mathsf{path}[:i]$ that can legally reside in $\mathsf{path}[\min\{i+1, L\}:]$ after executing the ∞-Batched Circuit ORAM algorithm over the trace $\psi, \vec{S}$.

Observe that the non-dummy blocks in $\texttt{subtree}^{\mathsf{Tr}}$ can be expressed as a disjoint union of blocks from $\texttt{residue}^{\mathsf{path},i}$ for appropriate choices of path and i; a similar result also holds for the batched variant. Hence, it suffices to prove that for every ψ, every $\vec{S}$, every path, and every i, it holds that

$$\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}) \leq \overline{\texttt{residue}}^{\mathsf{path},i}(\psi, \vec{S}).$$

It suffices to show that this invariant is maintained. Specifically, we show that for any t, if $\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[:t]) \leq \overline{\texttt{residue}}^{\mathsf{path},i}(\psi, \vec{S}[:t])$, then it holds that the above invariant is maintained after serving the next batch of m accesses, i.e., $\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[:t{+}1]) \leq \overline{\texttt{residue}}^{\mathsf{path},i}(\psi, \vec{S}[:t+1])$.

Observe that both the ∞-Circuit ORAM and the ∞-Batched Circuit ORAM receive the same m new requests. Moreover, we fix the randomness of both Circuit ORAM and Batched Circuit ORAM. Hence, the two algorithms select the same $2m$ eviction paths for the m requests.

Consider any path and i. Let m' be the number of blocks among the m requests that can legally reside in $\mathsf{path}[\min\{i+1, L\} :]$. Further, suppose that the path prefix $\mathsf{path}[: \min\{i+1, L\}]$ is covered k times by the $2m$ eviction paths. By Fact 1, it holds that

$$\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t+1]) = \max(\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t]) + m' - k, 0). \tag{1}$$

On the other hand, since ∞-Circuit ORAM's $2m$ evictions interleave with the fetches, not all of these m' candidates will have a chance to be evicted from $\mathsf{path}[: i]$ into $\mathsf{path}[i+1 :]$, therefore we have that

$$\overline{\texttt{residue}}^{\mathsf{path},i}(\psi, \vec{S}[: t+1]) \geq \max(\overline{\texttt{residue}}^{\mathsf{path},i}(\psi, \vec{S}[: t]) + m' - k, 0). \tag{2}$$

From the induction hypothesis, we have $\overline{\texttt{residue}}^{\mathsf{path},i}(\psi, \vec{S}[: t]) \geq \texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t])$. Hence, by comparing (1) and (2), the induction step is completed.

□

Relating Circuit OPRAM's pool occupancy to Batched Circuit ORAM's stash. We first consider the scenario that the number m of PRAM CPUs does not decrease. We say that $\vec{S}$ is *non-decreasing*, if for any $1 \leq i < j \leq T$, it holds that $|S_i| \leq |S_j|$. Let $\texttt{pool}(\psi, \vec{S})$ denote the number of non-dummy blocks in the pool in Circuit OPRAM after an execution trace defined by $(\psi, \vec{S})$; and let $\texttt{tree}_\blacktriangledown(\psi, \vec{S})$ be the number of non-dummy blocks in the stash and all heights smaller than $\log_2(2m)$ of Batched Circuit ORAM at the end of an execution trace defined by $(\psi, \vec{S})$, where m is the number of CPUs in the last batch of requests in $\vec{S}$. We next show the following lemma.

Lemma 4. *For every ψ and every non-decreasing $\vec{S}$,*

$$\texttt{pool}(\psi, \vec{S}) = \texttt{tree}_\blacktriangledown(\psi, \vec{S}).$$

Proof. We first consider the case when the number of CPUs stays fixed, i.e., for any $i, j \in [T]$, $|S_i| = |S_j|$. In this case, the lemma follows directly from Fact 1 and the construction of Circuit OPRAM's eviction algorithm.

Next, observe that whenever the number of CPUs increases, Circuit OPRAM simply merges more smaller heights of the ORAM tree into the pool. Therefore, the lemma holds as long as the number of CPUs does not decrease. □

Let $\mathbf{pool}(\vec{S})$ and $\mathbf{tree}_\blacktriangledown(\vec{S})$ be the random variables corresponding to $\texttt{pool}(\psi, \vec{S})$ and $\texttt{tree}_\blacktriangledown(\psi, \vec{S})$, respectively. Then, we have the following lemma.

Lemma 5. *For appropriate constants c and c', we have that for every $\vec{S}$,*

$$\Pr[\mathbf{tree}_\blacktriangledown(\vec{S}) > c \cdot m + R] < c' \cdot e^{-R}.$$

Furthermore, for every non-decreasing $\vec{S}$,

$$\Pr[\mathbf{pool}(\vec{S}) > c \cdot m + R] < c' \cdot e^{-R},$$

where the probability is taken over the randomness chosen by the Batched Circuit ORAM and the Circuit OPRAM algorithm respectively.

Proof. Follows in a straightforward fashion from Lemma 3, Fact 3 and Lemma 4. □

Handling the case of decreasing m. Lemma 5 handles the case when the number of CPUs does not decrease. Recall that when the number of CPUs decreases, our OPRAM algorithm reconstructs an extra height of the big ORAM tree. Although the blocks in this reconstructed height might not be exactly the blocks found in the corresponding height had we not merged smaller heights of the big ORAM tree into the pool, we show that our reconstruction algorithm can only make it more favorable in terms of stash utilization.

Consider running Batched Circuit ORAM where the entire ORAM tree is maintained without merging the smaller heights into a pool. At any point, we may apply one or more rearranging operations on the ORAM tree's memory layout. The following operations are *admissible*, in the sense that they will not increase the stash utilization:

1. **Partial eviction.** Choose any path and any height ℓ^*, perform the Circuit ORAM eviction algorithm on $\mathsf{path}[:\ell^*]$.
2. **Switch.** For any two blocks $B \in \mathsf{path}[i]$ and $B' \in \mathsf{path}[j]$ respectively where $j > i$, suppose that B can legally reside in $\mathsf{path}[j]$, we switch the locations of B and B'.

For convenience, we introduce the following notations:

- Let the pair $(\vec{S}, P)$ denote a sequence of requests in Batched Circuit ORAM that are interleaved with admissible rearranging operations in P, which also contains information on how the rearranging operations interleave with $\vec{S}$.
- Let $\texttt{tree}_{\blacktriangledown}(\psi, \vec{S}, P)$ denote the number of non-dummy blocks that remain below (i.e., closer to the root) height $\log_2(2m)$ (including the stash) in Batched Circuit ORAM after executing over the trace $(\psi, \vec{S})$ and performing the sequence of rearranging operations P, where m is the number of requests in the last batch of requests in $\vec{S}$.
- Let $\texttt{tree}_{\clubsuit}(\psi, \vec{S}, P)$ be the block layout of all heights at least $\log_2(2m)$ in Batched Circuit ORAM after executing over the trace $\psi, \vec{S}$ and performing the sequence of rearranging operations P, where m is the number of requests in the last batch of requests in $\vec{S}$.
- Recall that $\texttt{pool}(\psi, \vec{S})$ denotes the number of non-dummy blocks in the pool after executing the Circuit OPRAM algorithm over the trace $(\psi, \vec{S})$.
- Additionally, we use the notation $\texttt{allsubtrees}(\psi, \vec{S})$ to denote the layout of all disjoint subtrees in Circuit OPRAM after executing over the trace $(\psi, \vec{S})$.
- For clarity, we use $\equiv$ to denote equivalence of the block layout (i.e., which blocks in each bucket), and we use $=$ to denote equivalence of utilization (i.e., how many blocks in each bucket).

Lemma 6 (Equivalence to Batched Circuit ORAM with rearranging)**.** *For any ψ and $\vec{S}$, there exists a sequence of interleaving admissible rearranging operations P such that*

$$\texttt{allsubtrees}(\psi, \vec{S}) \equiv \texttt{tree}_{\clubsuit}(\psi, \vec{S}, P) \quad \textit{and} \quad \texttt{pool}(\psi, \vec{S}) = \texttt{tree}_{\blacktriangledown}(\psi, \vec{S}, P).$$

Proof. It suffices to show that for any t, if there is a sequence $P[:t]$ of interleaving admissible rearranging operations such that

$$\texttt{allsubtrees}(\psi, \vec{S}[:t]) \equiv \texttt{tree}_{\clubsuit}(\psi, \vec{S}[:t], P[:t]) \quad \text{and} \quad \texttt{pool}(\psi, \vec{S}[:t]) = \texttt{tree}_{\blacktriangledown}(\psi, \vec{S}[:t], P[:t]),$$

then there is a sequence $P[t+1]$ of admissible rearranging operations appended at the end such that

$$\texttt{allsubtrees}(\psi, \vec{S}[:t+1]) \equiv \texttt{tree}_{\clubsuit}(\psi, \vec{S}[:t+1], P[:t]||P[t+1]), \quad \text{and}$$
$$\texttt{pool}(\psi, \vec{S}[:t+1]) = \texttt{tree}_{\blacktriangledown}(\psi, \vec{S}[:t+1], P[:t]||P[t+1]).$$

If in PRAM step $t+1$, the number of CPUs does not decrease, then the above trivially holds for $P[t+1] = \emptyset$ due to Fact 1. Henceforth, we focus on the case that the number of CPUs decreases (by a factor of 2) in PRAM step $t+1$. In this case, our Circuit OPRAM algorithm needs to reconstruct an extra height of the big ORAM tree, and let ℓ^* be the height that is being reconstructed. Due to Fact 1, after PRAM step $t+1$, the block layout of the tree for heights larger than ℓ^* is identical between Circuit OPRAM and Batched Circuit ORAM.

It suffices to show that there are some admissible rearranging operations on Batched Circuit ORAM such that the block layout of height ℓ^* will be identical between Batched Circuit ORAM and Circuit OPRAM after PRAM step $t+1$. Moreover, this also implies that $\texttt{pool}(\psi, \vec{S}[:t+1]) = \texttt{tree}_{\blacktriangledown}(\psi, \vec{S}[:t+1], P[:t]||P[t+1])$, because any block that is not in height ℓ^* or higher must be in one of the lower heights (or the pool).

Let bucket be any bucket in the height being reconstructed in PRAM step $t+1$. Recall that our OPRAM algorithm places the maximal number of blocks in the pool that can legally reside in bucket into bucket. If there are more than Z such blocks, an arbitrary tie breaking rule is used. Therefore, the number of blocks in any such reconstructed bucket must be no smaller than the number of blocks in bucket in Batched Circuit ORAM after executing the first $t+1$ steps of the trace $(\psi, \vec{S})$.

Therefore, starting from the block layout of Batched Circuit ORAM after the first $t+1$ PRAM steps, if there are blocks at height less than ℓ^* that that can legally reside in bucket and bucket is not yet full, we can always invoke one or more "partial eviction" operations to move such blocks into bucket, until we either run out of such blocks or bucket is full.

At this moment, the blocks in bucket in Batched Circuit ORAM and Circuit OPRAM may not be the same blocks. If this is the case, we can invoke one or more "switch" operations such that the same blocks appear in bucket in Batched Circuit ORAM and Circuit OPRAM after the first $t+1$ PRAM steps.

For each such bucket in height ℓ^*, we perform the rearranging operations as described above to form $P[t+1]$ that satisfies the following:

$$\texttt{pool}(\psi, \vec{S}[:t+1]) = \texttt{tree}_{\blacktriangledown}(\psi, \vec{S}[:t+1], P[:t]||P[t+1]),$$

as required. □

Lemma 7 (Admissible Rearranging Cannot Increase Stash Size)**.** *For any ψ and $\vec{S}$, and any sequence of admissible rearranging operations P in Batched Circuit ORAM, we have*

$$\texttt{tree}_{\blacktriangledown}(\psi, \vec{S}, P) \leq \texttt{tree}_{\blacktriangledown}(\psi, \vec{S}).$$

Proof. Let $\texttt{ORAMtree}^{\infty}(\psi, \vec{S}, P)$ denote the ∞-Batched Circuit ORAM's block layout after executing over the trace $\psi, \vec{S}$, as well as all of the partial eviction operations (but not the switch operations) in P. Let $\texttt{ORAMtree}^{\text{real}}(\psi, \vec{S}, P)$ denote the real Batched Circuit ORAM's memory layout after executing over the trace $(\psi, \vec{S})$, as well as all operations as indicated in P.

We claim that the following is true:

Claim 8. *For any ψ and $\vec{S}$, and any admissible rearranging sequence P, there exists an admissible post-processing Q (where post-processing is defined in the Circuit ORAM work [34]) such that*

$$Q(\texttt{ORAMtree}^{\infty}(\psi, \vec{S}, P)) \equiv \texttt{ORAMtree}^{\text{real}}(\psi, \vec{S}, P).$$

Proof. It suffices to show that for any ψ, $\vec{S}$ and t, if there exists an admissible post-processing Q such that

$$Q(\texttt{ORAMtree}^{\infty}(\psi, \vec{S})[:t], P[:t]) \equiv \texttt{ORAMtree}^{\text{real}}(\psi, \vec{S}[:t], P[:t]),$$

then there exists an admissible post-processing Q' such that

$$Q'(\texttt{ORAMtree}^{\infty}(\psi, \vec{S})[:t+1], P[:t+1]) \equiv \texttt{ORAMtree}^{\text{real}}(\psi, \vec{S}[:t+1], P[:t+1]).$$

The following lemma was proved in the Circuit ORAM paper [34], and it also holds for Batched Circuit ORAM. In particular, Fact 6 holds because Circuit ORAM's strong equivalence to ∞-ORAM proof also handles the case of partial eviction from the root to a certain height ℓ^*.

Fact 6 (Strong Equivalence to ∞-ORAM: Inductive Form [34])**.** *Suppose* $\texttt{ORAMtree}^{\text{real}}$ *denotes a block layout of a Batched Circuit ORAM, and* $\texttt{ORAMtree}^{\infty}$ *denotes that of the ∞-variant such that* $\texttt{ORAMtree}^{\text{real}}$ *is an admissible post-processing of* $\texttt{ORAMtree}^{\infty}$*.*

Suppose further that the the same sequence of requests and (possibly partial) evictions are performed on both $\texttt{ORAMtree}^{\text{real}}$ *and* $\texttt{ORAMtree}^{\infty}$*. Let the corresponding resulting block layouts be* $\texttt{NewORAMtree}^{\text{real}}$ *and* $\texttt{NewORAMtree}^{\infty}$*, respectively.*

Then, there exists an admissible post-processing Q, such that

$$Q(\texttt{NewORAMtree}^{\infty}) \equiv \texttt{NewORAMtree}^{\text{real}}.$$

Due to Fact 6, given the induction hypothesis that for some Q,

$$Q(\texttt{ORAMtree}^{\infty}(\psi, \vec{S}, P)) \equiv \texttt{ORAMtree}^{\text{real}}(\psi, \vec{S}, P),$$

we have that there is some post-processing Q'' such that

$$Q''(\texttt{ORAMtree}^{\infty}(\psi, \vec{S})[:t+1], P[:t+1]) \equiv \texttt{ORAMtree}^{\text{real}}(\psi, \vec{S}[:t+1], \widehat{P}[:t+1]),$$

where $\widehat{P}[:t+1]$ is defined as follows:

$$\widehat{P}[:t] = P[:t] \text{ and } \widehat{P}[t+1] = (\text{partial evictions operations in } P[t+1]).$$

Hence, we can set admissible post-processing Q' to be Q'' plus the the switch operations in $P[t+1]$, and we have

$$Q'(\texttt{ORAMtree}^{\infty}(\psi, \vec{S})[:t+1], P[:t+1]) \equiv \texttt{ORAMtree}^{\text{real}}(\psi, \vec{S}[:t+1], P[:t+1]).$$

This completes the proof of Claim 8. □

Note that Claim 8 implies a generalization of Fact 4 that includes admissible rearranging operations. Specifically, we have the following corollary.

Corollary 1. *For every ψ and $\vec{S}$, and every admissible rearranging sequence P, suppose $\texttt{subtree}^{\mathsf{Tr}}(\psi, \vec{S}, P)$ denotes the number of non-dummy blocks in the rooted subtree Tr, after executing the ∞-Batched Circuit ORAM over trace $\psi, \vec{S}$ and the partial eviction operations in P, and $\texttt{stash}(\psi, \vec{S})$ denotes the stash utilization of the real Batched Circuit ORAM after executing the ∞-Batched Circuit ORAM over trace $\psi, \vec{S}$ and all operations in P.*

Suppose further that R is the minimum value such that for every rooted subtree Tr, $\texttt{subtree}^{\mathsf{Tr}}(\psi, \vec{S}, P) \leq |\mathsf{Tr}| \cdot Z + R$, where Z is the bucket capacity.

Then, we have $\texttt{stash}(\psi, \vec{S}, P) = R$.

Let $\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}, P)$ denote the number of non-dummy blocks in $\mathsf{path}[:i]$ that can legally reside in $\mathsf{path}[\min\{i+1, L\} :]$ after executing the ∞-Batched Circuit ORAM algorithm over the trace $(\psi, \vec{S})$ plus the rearranging sequence indicated in P. Recall that as a special case, $\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S})$ denotes the same but without any rearranging operations. Therefore, it remains to show the following lemma.

Lemma 9. *For any $\psi, \vec{S}, P, \mathsf{path}$ and i,*

$$\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}, P) \leq \texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}).$$

Proof. We prove by induction on t. It suffices to show that for any t, path and i, if $\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t], P[: t]) \leq \texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t])$, then it holds that $\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t+1], P[: t+1]) \leq \texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t+1])$.

Let m' denote the number of blocks in the $(t+1)$-st batch of requests that can legally reside in $\mathsf{path}[\min\{i+1, L\} :]$. Suppose further that the path prefix $\mathsf{path}[: \min\{i+1, L\}]$ is covered k times by the $2m$ eviction paths selected for the $(t+1)$-st batch of requests. By Fact 1, it holds that

$$\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t+1]) = \max(\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t]) + m' - k, 0). \tag{3}$$

Similarly, suppose that the path prefix $\mathsf{path}[: i+1]$ is covered $k' \geq k$ times by the $2m$ eviction paths selected for the $(t+1)$-th batch of requests, as well as the partial evictions in $P[t+1]$. Then, by Fact 1, it holds that

$$\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t+1], P[: t+1]) = \max(\texttt{residue}^{\mathsf{path},i}(\psi, \vec{S}[: t], P[t]) + m' - k', 0). \tag{4}$$

Then, by using the induction hypothesis and comparing (3) and (4), the inductive step is completed. □

The proof of Lemma 7 then follows directly from Claim 8, Fact 6 and Lemma 9. □

Theorem 3. *For appropriate constants c and c', we have that for every $\vec{S}$,*

$$\Pr[\mathbf{pool}(\vec{S}) > c \cdot m + R] < c' \cdot e^{-R}$$

where the probability is taken over the randomness chosen by the Circuit OPRAM algorithm.

Proof. This follows from Lemmas 5, 6 and 7. □

9 Security Analysis of Circuit OPRAM

Theorem 4 (Security of Circuit OPRAM)**.** *Circuit OPRAM satisfies obliviousness as formulated in Section 3.2.*

Proof. Since the security notion in Section 3.2 assumes that the number of PRAM steps and the number of active CPUs in each PRAM step are public information, it suffices to show that the distribution of the access pattern in each simulation of a PRAM step is independent of the batch of addresses requested.

The main idea for showing security is the same as the argument for all known tree-based ORAMs [30, 33, 34] and tree-based OPRAMs [3, 6] — whenever a block was last accessed, the OPRAM algorithm chooses a random path for the block without revealing the position identifier. Therefore, when the block is accessed next, an independent random position identifier is revealed to the adversary.

We next check that both the fetch and the maintain phases satisfy the security notion. For the fetch phase, in the description in Sections 5.3 and 6, it is straightforward to check that whenever the requested addresses are involved in the algorithm, the access pattern does not leak any information about them because of the security of the building blocks (for instance, conflict resolution, oblivious routing, oblivious sorting and oblivious multicast) used to handle them. Moreover, as mentioned above, the path in the OPRAM tree for each address is revealed only when it is requested, and that address will be assigned to a freshly generated random path afterwards.

In the maintain phase, observe that the m paths used in the simultaneous removal procedure have the distribution of m independent random paths. The path selection criteria are deterministic, and the security of all other procedures are again ensured by the building blocks. For instance, eviction candidates selection and tree height reconstruction are carried out by either oblivious bin packing or pipelined linear scan. □

10 Additional Optimizations and Performance Analysis

As seen in Lemmas 1 and 2, so far, our Circuit ORAM construction suffers from a super-constant factor α in its simulation overhead for small values of m, specifically when $m < \alpha \log \log N$. The reason is that several steps of our OPRAM algorithm involve oblivious sorting on the pool, such as fetching blocks from the pool, selecting eviction candidates from the pool, and cleaning up the pool. Recall that with all but $\mathsf{negl}(N)$ probability, the pool size is upper bounded by $O(m + \alpha \log N)$. When $m < \alpha \log \log N$, oblivious sorting on the pool would be too expensive for us. Therefore, in our earlier OPRAM construction, we treat the case of small m specially, by devising algorithms that require m CPUs to linearly scan through the pool (whose size is $O(\alpha \log N)$ assuming $m < \alpha \log \log N$). In this section, we show how to remove this extra α factor for the case when $m < \alpha \log \log N$. Then, for all values of m, we can achieve the same overhead as stated in the following theorem.

Theorem 5 (Statistically Secure OPRAM for General Block Sizes)**.** *Any PRAM whose block size is at least* $\log N$ *bits can be obliviously simulated with* $O(\log^2 N)$ *overhead and achieving* $\mathsf{negl}(N)$ *statistical failure probability.*

10.1 Removing the Super-Constant Factor α for Small m

We now propose an extension that removes the extra α factor in Lemmas 1 and 2 for the case when $m < \alpha \log \log N$. Our idea is the following.

- **Merge the pools from all recursion levels into a single pool when m becomes small.** Suppose that in some PRAM step, the number m of CPUs drops from above $\alpha \log\log N$ to below $\alpha \log\log N$. Then, at the end of the PRAM step, we merge all $D = O(\log N)$ recursion levels' pools into a single pool. Further, we compress the pool to $O(mD + \alpha \log N)$ number of blocks using a single oblivious sort — we shall later prove that the merged pool has size at most $O(mD + \alpha \log N)$ except with $\mathsf{negl}(N)$ probability. This oblivious sorting can be completed in $O(\log\log N)$ steps consuming $O(\alpha D \log N)$ CPUs, or alternatively, in $O(D \log N)$ parallel steps with $m = \Theta(\alpha \log\log N)$ CPUs — recall that by assumption, the number of CPUs cannot decrease by more than a factor of 2 between adjacent PRAM steps.

  When the PRAM's number of CPUs stays below $\alpha \log\log N$, our OPRAM construction will now operate with a single merged pool. We therefore need to make a few minor modifications to our OPRAM algorithm:

  1. *Pool lookup.* When blocks are requested in the fetch phase from the pool, we now use a single instance of oblivious routing to route the desired blocks to all recursion levels — note that we need to route at most m blocks to each recursion level. Since the merged pool is bounded by $O(mD + \alpha \log N)$ in size, this oblivious routing can be accomplished in $O((mD + \alpha \log N) \log\log N) = O(\log N \log\log^3 N)$ steps with a single CPU.
  2. *Selection of eviction candidates.* Selection of eviction candidates from the pool is also now performed with a single instance of oblivious routing for the $2mD$ eviction paths across all recursion levels. Similarly as before, this can be accomplished in $O(\log N \log\log^3 N)$ steps with a single CPU.
  3. *Pool cleanup.* At the end of simulating each PRAM step, pool cleanup (i.e., compression) can also be performed with a single oblivious sort over the pool, Similarly as before, this can be accomplished in $O(\log N \log\log^3 N)$ steps with a single CPU.

- **Split into separate pools again when m becomes large.** Suppose that in some PRAM step, the number of CPUs m increases from below $\alpha \log\log N$ to at least $\alpha \log\log N$. Then, at the beginning of simulating this PRAM step, we split the merged pool into D separate pools, one for each recursion level. Since each recursion level receive up to $O(\alpha \log N)$ blocks with all but negligible probability, the splitting can be achieved with an oblivious bin packing operation where we have D bins each of capacity $O(\alpha \log N)$. Since $D = O(\log N)$, such oblivious bin packing can be achieved in $O(\log\log N)$ parallel steps with $O(\alpha D \log N)$ CPUs, or alternatively, in $O(D \log N)$ parallel steps with $m \geq \alpha \log\log N$ CPUs.

10.2 Detailed Analysis

It remains to prove that the merged pool's occupancy is upper bounded by $O(mD + \alpha \log N)$ except with $\mathsf{negl}(N)$ probability.

Lemma 10. *The merged pool's occupancy is upper bounded by $O(mD + \alpha \log N)$ except with $\mathsf{negl}(N)$ probability.*

Proof. For each level recursion level d, let Y_d be the occupancy of the pool at level d. As stated in Section 8, Fact 3 states that there exist constants $\kappa, A > 0$ such that for any $R > 0$,

$\Pr[Y_d \geq \kappa m + R] \leq Ae^{-R}$.

Define $X_d := Y_d - \kappa m$. For $0 < t \leq \frac{1}{2}$, consider the moment generating function

$E[e^{tX_d}] = \int_0^\infty \Pr[e^{tX_d} \geq r]dr \leq 1 + \int_1^\infty Ar^{-\frac{1}{t}}dr = 1 + \frac{A}{\frac{1}{t}-1}$.

Choose $t := \frac{1}{2}$ and $K := 2\ln(1 + A)$. Then, observing that the X_d's are independent over different d's, we have for any $R > 0$,

$\Pr[\sum_{d\in[D]} X_d \geq KD+R] \leq E[e^{t\sum_{d\in[D]} X_d}] \cdot e^{-t(KD+R)} \leq e^{-\frac{R}{2}}$, where the first inequality follows from a standard derivation using Markov's inequality, and the last inequality uses the independence of the X_d's and the above upper bound for the moment generating function.

Choosing $R = \alpha \log N$, we conclude that with all but $\mathsf{negl}(N)$ probability, the sum $\sum_{d\in[D]} Y_d = O(mD + \alpha \log N)$. □

Proof of Theorem 5: Lemmas 1 and 2 handle the case when $m \geq \alpha \log\log N$. When $m < \alpha \log\log N$, the above algorithm and Lemma 10 allows us to remove the extra α factor from the overhead. □

When Theorem 5 is cast to the sequential ORAM setting, it actually improves the state-of-the-art (i.e., Circuit ORAM [34]) by a super-constant factor — recall that Circuit ORAM achieves $O(\alpha \log^2 N)$ simulation overhead for general block sizes. Effectively, this means that the techniques we use in constructing OPRAMs (in particular, the technique for merging stashes across all recursion levels) can actually be used in the sequential ORAM setting to improve the state-of-the-art [34] by a super-constant factor. We summarize this observation in the following corollary which arises immediately as a special case of Theorem 5,

Corollary 2 (Statistically secure ORAM for general block sizes)**.** *Any RAM whose block size is at least* $\log N$ *bits can be obliviously simulated with* $O(\log^2 N)$ *overhead and achieving* $\mathsf{negl}(N)$ *statistical failure probability.*

11 Reducing the Simulation Overhead Using PRFs

Kushilevitz et al. [22] demonstrated a computationally secure ORAM construction using pseudorandom functions (PRFs) that achieves $O(\frac{\log^2 N}{\log\log N})$ simulation overhead. We now also show how to leverage PRFs to compress position map blocks to reduce our OPRAM's simulation overhead by a $\log\log N$ factor — and thus our result also strictly generalizes that of Kushilevitz et al. [22]. We not only obtain an OPRAM construction matching Kushilevitz et al.'s sequential ORAM result under computational security, our construction is also orders of magnitude more efficient in practice and conceptually much simpler than that of Kushilevitz et al. Further, we outperform Kushilevitz et al. for large blocks.

11.1 Intuition and Overview

To reduce a $\log\log N$ factor in the overhead, we rely on an elegant idea that was first described by Fletcher et al. [11] in a tree-based ORAM context. On a very high level, the idea is to compress the position map blocks using counters, and then generate the position labels pseudorandomly using a PRF whose secret key is known only to the CPU(s). Through such compression, we can pack $\Theta(\frac{\log N}{\log\log N})$ counters in each position map block, and therefore this reduces the depth of recursion by a $\log\log N$ factor. The concrete instantiation of the algorithm by Fletcher et al. [11] is only secure if the adversary cannot observe the boundaries of each PRAM step in the OPRAM's address sequence. We shall describe an improved variant of their idea that removes this additional assumption, and retains security even when the boundary of PRAM steps is known to the adversary. Furthermore, we describe how to modify our OPRAM construction to be compatible with this PRF technique.

Henceforth, like in all earlier hierarchical ORAM schemes [16–18, 22], we assume that the evaluation of the PRF can be done in $O(1)$ steps by a single CPU, and that the CPU cache for storing the PRF's secret key is for free. We also focus on the case of general block sizes, and hence $D = O(\frac{\log N}{\log\log N})$.

Position map block format. Instead of storing position identifiers in the position map, we store compressed counters for blocks in the position map. Specifically, in each block of $\Theta(\log N)$ bits, we pack counters for $\gamma := \frac{\log N}{\log \log N}$ blocks, where γ is referred to as the branching factor. All γ blocks share an *outer counter* that is $\Theta(\log N)$ bits long. Each of the γ blocks has an inner counter that is $3 \log \log N$ bits long. Therefore, a position map block at recursion level d is of the following format:

$$\boxed{\mathsf{addr}^{\langle d \rangle}, \mathsf{outer\_counter}, \left\{\mathsf{inner\_counter}[i] : i \in [0, \ldots, \gamma - 1]\right\}}$$

This means that for $i \in [0, \ldots, \gamma - 1]$, the block at address $(\mathsf{addr} \cdot \gamma + i)$ in the next recursion level (i.e., recursion level $d + 1$) resides at the leaf numbered

$$\mathsf{PRF}_{\mathsf{sk}}(\mathsf{addr}||d||\mathsf{outer\_counter}||\mathsf{inner\_counter}[i]),$$

where sk is a secret key known only to the CPU.

Overview of counter-related operations. We now describe how to update these counters and probabilistically reset them, such that we can avoid counter overflow with all but $\mathsf{negl}(N)$ probability.

- *Increment.* Every time a block at recursion level $d + 1$ and address $\mathsf{addr}^{\langle d+1 \rangle}$ is accessed, its inner counter, stored in the block identified by the address $\lfloor \frac{\mathsf{addr}^{\langle d+1 \rangle}}{\gamma} \rfloor$ of recursion level d, is incremented. This can be accomplished since recall that in our OPRAM algorithm, each position map block learns whether each of its children addresses is needed by the next recursion level during the preparation stage. Later we use a probabilistic resetting procedure to guarantee that inner counters do not overflow except with $\mathsf{negl}(N)$ probability.

- *Probabilistic reset.* For a batch of m memory requests, m blocks from each position map level will be accessed. For each position map level, we will randomly choose $\frac{m}{\gamma}$ of the fetched position map blocks to reset (we will describe how to accomplish this in parallel later); and thus each fetched position map block will be reset with probability $\frac{1}{\gamma} = \frac{\log \log N}{\log N}$. For any such block chosen for reset, the outer counter is incremented and all inner counters reset to 0.

  For any position map block being reset, all of its γ children blocks in the next recursion level must be relocated — in total, each recursion level must relocate m blocks. However, some of these relocated blocks are also requested, and hence, special care is needed to ensure that the adversary just observes that $2m$ independently random paths are searched. We describe how to accomplish this in detail below.

11.2 Detailed Algorithm

Making the above ideas fully work involve several subtleties. We describe the modified OPRAM algorithm below.

The modified fetch phase. Based on the above ideas, we now have the following modified fetch phase.

i) *Preparation: all recursion levels in parallel.* For each recursion level $d = 0, 1, \ldots, D$ in parallel:

- *Generate level-d prefix addresses.* Same as before.
- *Conflict resolution.* Same as before.

- *Discover which children addresses are needed .* As before, use the algorithm described in Section 6 such that each conflict resolved address in level d discovers which of its γ children are needed by the next recursion level. Let $\mathsf{Addr}^{\langle d \rangle}$ be the resulting conflict resolved list of m addresses at level d, where each address is tagged with a γ-bit vector denoting whether each of its children is needed in the next recursion level.
- *Randomly choose blocks to reset.* If d is a position map level, then randomly permute the array $\mathsf{Addr}^{\langle d \rangle}$ obliviously — this can be accomplished in $O(\log m)$ steps with m CPUs using oblivious sorting and using a PRF to compare elements. Now, the first $\frac{m}{\gamma}$ addresses represent position map blocks that will be reset — this means that all the children blocks in the next recursion level will need to relocate. Tag the first $\frac{m}{\gamma}$ entries of $\mathsf{Addr}^{\langle d \rangle}$ with the tag $\texttt{reset} := 1$; tag all other elements with $\texttt{reset} := 0$ — each address will always carry the `reset` bit throughout, and this `reset` bit is needed later when updating the block's content and passing the updated block back to the pool.

  For the $\frac{m}{\gamma}$ level-d addresses chosen for reset, the m children addresses at level $d+1$ must be relocated. Inform recursion level $d+1$ of these m addresses to be relocated, henceforth referred to as *relocate addresses*.
- *Duplicate suppression.* At this point, every recursion level d knows m conflict-resolved, possibly dummy level-d *fetch addresses*; as well as m *relocate addresses*. Henceforth we assume that the relocate addresses have the tag `relocate` set to 1. Note that the fetch addresses and the relocate addresses may overlap. It is important to suppress duplicates because otherwise we can leak how large this overlap is, and thus leak secret information.

  Each recursion level d now performs duplicate suppression: such that if an address is both a fetch address and a relocate address, only one entry is preserved (with the `relocate` tag set to 1); and the other duplicate entry is replaced with $\perp$. The result is an size-$2m$ array of addresses, some of which tagged with `relocate`. It is not hard to see that this step can be accomplished through oblivious sorting in $O(\log m)$ parallel steps with m CPUs for each recursion level.
- *Pool lookup.* This is same as before, but we look for $2m$ addresses in the pool, containing both *fetch addresses* and *relocate addresses*.

ii) *Fetch: level by level.* For each recursion level $d = 0, 1, \ldots, D$ in sequential order:

- *Receive position counters from previous recursion level.* For $d = 0$, the position labels can be fetched in $O(1)$ parallel step. For $d \geq 1$, each of the $2m$ addresses (containing both fetch and relocate addresses) receives its outer counter and inner counter from the previous recursion level. This can be accomplished through an instance of oblivious routing consuming $O(\log m)$ parallel steps with m CPUs.

  Recall that each level-d address also knows its own `relocate` bit. Therefore, once each level-d address receives its present outer and inner counter, each address can compute on its own its present position label (i.e., path to fetch) and its new position label (i.e., path to reassign after fetch).
- *Subtree lookup.* This is same as before, except that we are looking up $2m$ paths corresponding to both fetched and relocate addresses. For a position map level, each of the $2m$ fetched results will be of the following format:

$$\boxed{\mathsf{addr}^{\langle d \rangle},\ \mathsf{outer\_counter},\ \left\{\mathsf{inner\_counter}[i] : i \in [0, \ldots, \gamma - 1]\right\}}$$

  Now, if we union the subtree lookup result and the earlier pool lookup result (where each entry is of the same format), we obtain all position counters that are ready to be routed to the next recursion level.

iii) *Oblivious multicast.* This is the same as before, except that blocks that are relocated but not requested need not be routed.

The modified maintain phase. We now have the following modified maintain phase. For each recursion level $d = 0, 1, \ldots, D$ in parallel:

i) *Simultaneous removal of fetched blocks from subtrees.* This is the same as before, but now we also need to remove from $2m$ paths corresponding to both fetch addresses and relocate addresses.

ii) *Passing updated blocks to the pool.* For all fetched blocks (including fetch and relocate addresses), update the block's contents and add the updated block back to the pool. First, each fetched block can compute its new position label by evaluating the PRF. Next, suppose d is a position map level, we update the contents of each fetched block as below: if a block is chosen for reset (recall that the block's address carries a `reset` bit which indicates whether block is chosen for reset), we reset all inner counters to 0 and increment the outer counter; else, the inner counters of all blocks needed by the next recursion level are incremented[5].

iii) *Increasing the number of subtrees if necessary.* Same as before.

iv) *Selection of eviction candidates and eviction into subtrees.* This is the same as before, except that we perform this part twice to account for the fact that up to m relocated blocks have been placed into the pool. When this part is performed the second time, the $2m$ eviction paths will be different and are selected according to the same deterministic rule as outlined in [34].

v) *Decreasing the number of subtrees if necessary.* Same as before.

vi) *Cleanup.* Same as before.

11.3 Analysis

Counter overflow analysis. Since the inner counter has $3 \log \log N$ bits, we must guarantee that among any $2^{3 \log \log N} = \log^3 N$ visits to a position map block, there must be a reset event to this block except with negligible probability. This is easy to see because the probability that a block is not reset during an access is $1 - \frac{\log \log N}{\log N}$. Therefore, the probability that a block is not reset after $\log^3 N$ accesses is

$$(1 - \frac{\log \log N}{\log N})^{\log^3 N} = \mathsf{negl}(N)$$

Finally, by taking a union bound with polynomial loss, we conclude that among any $2^{3 \log \log N} = \log^3 N$ visits to a position map block, there must be a reset event to this block except with negligible probability.

Notice that since the total number of accesses is polynomial in N (the security parameter), if the outer counter is $\Theta(\log N)$ bits, the outer counter should never overflow. Alternatively, if we do not know an upper bound on the number of accesses in advance, we can simply set the outer counter to $5 \log N$ bits long, and every N^3 accesses, we rebuild the entire ORAM — the cost of this rebuilding can be easily amortized over the N^3 accesses.

Overhead analysis. Since the number of addresses reduces by a factor of $\gamma = \frac{\log N}{\log \log N}$ between successive recursion levels, the depth of the recursion is only $O(\frac{\log N}{\log \log N})$. Therefore, by adopting a

[5] We assume that this update for each position map block can be completed in $O(1)$ CPU steps since we already that the PRF can be evaluated in $O(1)$ CPU steps.

PRF, we reduce the simulation overhead by a $\log \log N$ factor in comparison with the statistically secure variant.

Based on the above description and analysis, we immediately derive the following theorem.

Theorem 6 (Computationally secure OPRAM for general block sizes)**.** *Assume the existence of one-way functions. Then, any PRAM whose block size is at least* $\log N$ *bits can be obliviously simulated with* $O(\frac{\log^2 N}{\log \log N})$ *overhead, achieving* $\mathsf{negl}(N)$ *failure probability against any probabilistic polynomial-time adversary.*

An interesting special case is when $m = 1$. Our result implies the existence of a computationally-secure tree-based ORAM scheme with $O(\frac{\log^2 N}{\log \log N})$ simulation overhead, matching the theoretical result of Kushilevitz et al. [22] — but in comparison our construction is conceptually much simpler and practically orders of magnitude more efficient. We summarize this resulting corollary which follows directly from Theorem 6.

Corollary 3 (Computationally secure ORAM for general block sizes)**.** *Assume the existence of one-way functions. Then, any RAM whose block size is at least* $\log N$ *bits can be obliviously simulated with* $O(\frac{\log^2 N}{\log \log N})$ *overhead, achieving* $\mathsf{negl}(N)$ *failure probability against any probabilistic polynomial-time adversary.*

12 Extensions for Other Block Sizes and Metrics

In this section, we will interpret our results for sufficiently large block sizes (when the recursion depth can be as small as constant). We will also interpret our results other other metrics of interest that have been considered earlier [30, 33].

12.1 Large Block Sizes

The number of recursion levels would decrease as the block size increases, since each block can now pack more position labels. Of particular interest is when when the block size is $\Omega(N^\epsilon)$ for an arbitrarily small constant $\epsilon > 0$ — in this case the depth of recursion becomes $O(1)$. We immediately have the following corollary for sufficiently large block sizes.

Corollary 4 (Asymptotically tight OPRAM for large blocks)**.** *Let* $\epsilon > 0$ *be any positive constant, and let* $\alpha = \omega(1)$ *be any appropriately small super-constant function.*

- *Any PRAM whose blocks are* N^ϵ *bits long can be obliviously simulated incurring only* $O(\alpha \log N)$ *overhead achieving* $\mathsf{negl}(N)$ *statistical failure probability.*
- *Further, any PRAM whose blocks are* N^ϵ *bits long and using at least* $\alpha \log \log N$ *CPUs in any parallel step can be obliviously simulated incurring only* $O(\log N)$ *overhead achieving* $\mathsf{negl}(N)$ *statistical failure probability.*

In light of Goldreich and Ostrovsky's lower bound [16, 17], our Circuit OPRAM construction is (almost) asymptotically optimal — the only possible room for improvement is removing the super-constant α factor for the case of small m. Note that our algorithm for removing the α factor (see Section 10.1) is not applicable here since its costs can no longer be asymptotically absorbed by the $O(\alpha \log N)$ overhead.

12.2 Non-Uniform Block Sizes and Bandwidth Blowup

So far, we focused on a general simulation overhead metric. In certain application settings, such as cloud storage outsourcing, we may care about other metrics, for example, bandwidth blowup. We define *bandwidth blowup* as the ratio of the total number of bits transferred between the CPU (i.e., client) and memory (i.e., cloud server) in the OPRAM case vs. the PRAM case. Under uniform block sizes, our simulation overhead metric and bandwidth blowup are somewhat equivalent. However, if the OPRAM is allowed to adopt different block sizes for the data-OPRAM and the pos-OPRAMs, then as suggested in earlier tree-based ORAM/OPRAM works [30, 33, 34], we can adopt a "big data block, little metadata block" trick to reduce the bandwidth blowup to $O(\log N)$ for data blocks of only $\Omega(\log^2 N)$ bits long. The idea is to let data blocks be $\Omega(\log^2 N)$ bits long, but let pos-OPRAM blocks to be only $\Theta(\log N)$ bits long — in this way, fetching $O(\log N)$ pos-OPRAM blocks costs the same (or less) bandwidth than fetching a data block.

Acknowledgments

We are extremely grateful to Rafael Pass without whose insights and support this work would not have been possible. We are grateful to Joshua Gancher for helpful discussions in an earlier phase of the project, and to Kai-Min Chung for many supportive conversations. We thank Ling Ren for (re)explaining the position map compression trick to us and Kartik Nayak for very helpful comments on a draft of the paper. We thank the beanbag in the systems lab that played a crucial role in the initial phase of the project, as well as the beautiful Beebe lake and Watkins Glen state park in the summer. This work is supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, CNS-1601879, CNS-1617676, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, a VMWare Research Award, and a Baidu Research Award.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[2] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *Public Key Cryptography*, pages 131–148, 2014.

[3] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 175–204, 2016.

[4] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, pages 357–368, 2016.

[5] Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel oram. manuscript, 2017.

[6] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 205–234, 2016.

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

[8] Dana Dachman-Soled, Chang Liu, Charalampos Papamanthou, Elaine Shi, and Uzi Vishkin. Oblivious network ram and leveraging parallelism to achieve obliviousness. In *Asiacrypt*, 2015.

[9] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 145–174, 2016.

[10] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

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

[12] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[13] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[14] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[15] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345, 2014. `http://eprint.iacr.org/`.

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

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

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

[19] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[20] Torben Hagerup. Fast and optimal simulations between CRCW prams. In *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, pages 45–56, 1992.

[21] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525. Springer Berlin Heidelberg, 2014.

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

[23] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating Efficient RAM-model Secure Computation. In *S & P*, May 2014.

[24] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[25] Kartik Nayak and Jonathan Katz. An oblivious parallel ram with $o(\log^2 n)$ parallel runtime blowup. Cryptology ePrint Archive, Report 2016/1141, 2016. `http://eprint.iacr.org/2016/1141`.

[26] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.

[27] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[28] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious ram. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, Washington, D.C., 2015. USENIX Association.

[29] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

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

[31] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[32] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (S & P)*, 2013.

[33] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

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

[35] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *CCS*, 2014.

[36] Peter Williams and Radu Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[37] Peter Williams and Radu Sion. Round-optimal access privacy on outsourced storage. In *ACM Conference on Computer and Communication Security (CCS)*, 2012.

[38] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, pages 139–148, 2008.

A Algorithmic Details for Building Blocks

Oblivious aggregation for a sorted array. Oblivious aggregation [3, 26] is the following primitive where given a somewhat sorted array of (key, value) pairs, each representative element for a key will learn some aggregation function computed over all pairs with the same key.

- *Input:* An array $\mathsf{Inp} := \{k_i, v_i\}_{i \in [n]}$ of possibly dummy (key, value) pairs, where all pairs with the same key appear in consecutive locations. Henceforth we refer to all elements with the same key as the same *group*. We say that index $i \in [n]$ is a *representative* for its group if i is the leftmost element of the group.
- *Output:* Let Aggr be a publicly known, commutative and associative aggregation function and we assume that its output range can be described by $O(1)$ number of blocks. The goal of oblivious aggregation is to output the following array:

$$\mathsf{Outp}_i := \begin{cases} \mathsf{Aggr}\left(\{(k,v) | (k,v) \in \mathsf{Inp} \text{ and } k = k_i\}\right) & \text{if } i \text{ is a representative} \\ \perp & \text{o.w.} \end{cases}$$

Boyle et al. [3] and Nayak et al. [26] showed the following fact.

Fact 7 (Oblivious aggregation [3, 26])**.** *Oblivious aggregation for a sorted array of length n can be accomplished in $O(\log n)$ parallel steps with n CPUs.*

Below we describe some extensions of the oblivious aggregation algorithm, for the special case when each group's size is not too large.

Fact 8. *For any $k > 1$, suppose that each group size in the input array is bounded by k, then oblivious aggregation can be completed for the input array in $O(\log k)$ parallel steps consuming n CPUs.*

Proof. Run the same algorithm as Boyle et al. [3] and Nayak et al. [26], but stop early. More specifically, in their oblivious aggregation algorithm, in each iteration i, every entry of the array learns the aggregation result of the size 2^i region to its right. The general oblivious aggregation algorithm [3, 26] needs to be run for $\log n$ iterations where each iteration takes $O(1)$ parallel steps consuming n CPUs.

Since our group size is bounded by k, it suffices to stop the algorithm at iteration $\log k$. Therefore, we can complete oblivious aggregation in $O(\log k)$ parallel steps with n CPUs. □

We can further improve the algorithm for the case of bounded group size using the above fact as a stepping stone.

Lemma 11 (Oblivious aggregation on a sorted array for bounded groups)**.** *For any $k > 1$, suppose that each group size in the input array is bounded by k, oblivious aggregation can be completed for the input array in $O(\log k)$ parallel steps with $O(\frac{n}{\log k})$ CPUs.*

Proof. We can construct the following algorithm:

- First, divide the array into $\frac{n}{\log k}$ segments each of size $\log k$. Assign one CPU to each segment — through a linear scan of the segment, a single CPU can perform oblivious aggregation on each segment in $O(\log k)$ steps. The result is $n' := \frac{n}{\log k}$ output arrays denoted $X_1, X_2, \ldots, X_{n'}$ respectively — henceforth assume that each element in these arrays is tagged with its key (i.e., assume that the aggregation function always tags the outcome with the key).

- Perform partial oblivious aggregation on the array $Y := (X_1[1], X_2[1], \ldots, X_{n'}[1])$ — since we know at most k elements of Y share the same key, this oblivious aggregation can be completed in $O(\log k)$ parallel steps consuming $O(n')$ CPUs due to Fact 8. The result is an array denoted Z of length n'.

- Construct Out as follows — this step can be accomplished obliviously in $O(k)$ steps with $\frac{n}{k}$ CPUs.

$$\mathsf{Out}[i] := \begin{cases} \mathsf{Aggr}(X[i], Z[\lceil \frac{i}{\log k} \rceil + 1]) & \text{if } i \text{ is a representative and } Z[\lceil \frac{i}{\log k} \rceil + 1] \text{ has the same key} \\ X[i] & \text{if } i \text{ is a representative and } Z[\lceil \frac{i}{\log k} \rceil + 1] \text{ has a different key} \\ \perp & \text{o.w.} \end{cases}$$

  In the above, if $Z[\lceil \frac{i}{\log k} \rceil + 1]$ exceeds the array boundary to the right, we treat it in the same way as $Z[\lceil \frac{i}{\log k} \rceil + 1]$ having a different key. Also, note that the above works because Aggr is assumed to be commutative and associative.

□

Oblivious propagation for a sorted array. Oblivious propagation [26] is the opposite of aggregation. Given an array of possibly dummy (key, value) pairs where all elements with the same key appear consecutively, we say that an element is the *representative* if it is the leftmost element with its key. Oblivious propagation aims to achieve the following: for each key, propagate the representative's value to all other elements with the same key. Nayak et al. [26] show that such oblivious propagation can be achieved in $O(\log n)$ steps consuming n CPUs where n is the size of the input array.

Oblivious routing. Oblivious routing [3] is the following primitive where n source CPUs wish to route data to n' destination CPUs based on the key.

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

  We assume that each (non-dummy) key appears only once in the src array, however, each (non-dummy) key can appear multiple times in dst.

- *Outputs:* We would like to output two arrays: 1) an array $\mathsf{Out} := \{v'_i\}_{i \in [n']}$ such that for each $i \in [n']$, it holds that either $(k'_i, v'_i) \in \mathsf{src}$, or $k'_i \notin \mathsf{src}$ in which case $v'_i := \perp$, or $k'_i = v'_i = \perp$; and 2) an additional array Remain of length n that holds all remaining elements of src that have not been routed to the destination array (padded with dummies).

Oblivious routing can be accomplished in $O(\log(n + n'))$ steps consuming $n + n'$ CPUs. We describe the algorithm below which is a variant of what was described in Boyle et al. [3].

1. Let X be the concatenation of the src and dst arrays, where each element from src is tagged with `src`, and each element from dst is tagged with `dst` as well as its offset within the dst array. The array X can be constructed in $O(1)$ parallel steps with $n + n'$ CPUs.

2. Oblivious sort X based on the key, such that the same keys appear adjacent to each other, and for the same key, an element tagged with `src` appears before any element tagged with `dst`. This can be completed in $O(\log(n + n'))$ parallel steps with $n + n'$ CPUs.

3. Invoke an instance of the oblivious propagation algorithm such that for each key, the value contained in the leftmost element is propagated to all other elements in X with the same key. If the leftmost element is not from the src array, then the value propagated is assumed to be $\perp$. This can be completed in $O(\log(n+n'))$ parallel steps with $n+n'$ CPUs.

4. Output the Out array as follows. In parallel, for each element of the resulting array X: if the element is tagged with $\mathtt{dst}$, write down this entry; else write down $\perp$ — this forms a new array Y. Recall that all entries from dst are also tagged with its offset within dst. Now, oblivious sort Y in increasing order of this offset value, pushing all dummies elements to the end. Output $\mathsf{Out} := Y[1:n']$.

5. Output the Remain array as follows. In parallel, for each element of the resulting array X: if the element is tagged with $\mathtt{src}$, and the element to its right does not have the same key (or this is the end of the array), write down this entry; else write down $\perp$ — this forms a new array Z. Oblivious sort Z pushing all dummy elements to the end. Output $\mathsf{Remain} := Z[1:n]$.

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 pair $(g, \mathsf{priority})$:
  - $g \in [B] \cup \{\perp\}$ denotes a destined bin number, and if $g = \perp$, it means that the element is not eligible for any bin;
  - $\mathsf{priority}$ will be used to break ties if more than Z balls are destined for the same bin. A higher value of $\mathsf{priority}$ is preferred over a smaller one.
- *Outputs:* An array $\mathsf{Out}[1 : BZ]$ of length $B \cdot Z$ containing real and dummy elements, and an array Remain (of the same length as In) containing the remainder elements. Henceforth we say that $\mathsf{Out}[(g-1)B+1 : gB]$ is the g-th bin of the output array where $g \in [B]$. The outputs must satisfy the following:
  - All real elements in In appear exactly once in the union of Out and Remain; and further all real elements in Out and Remain must come from In.
  - All real elements in the g-th bin of Out must be some element of the input array In tagged with the bin number g.
  - If more than Z elements of In are destined for bin g, then the Z elements with the highest priority land in bin g; if fewer than Z elements of In are destined for bin g, then all of them must land in bin g of the output array.

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 $(\mathtt{filler}, g, \mathsf{priority} = -\infty)$ 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 elements with higher priority in the front, and place filler elements after real elements.

3. By invoking an instance of of the oblivious propagation algorithm, each element in the array finds the leftmost element in its own group. Now for each element in the array in parallel, if its offset within its own group is greater than Z, assign it with the tag `excess`; otherwise, assign it with the tag `normal`.

4. Oblivious sort the resulting array placing all elements tagged with `excess` and all dummies at the end. Elements tagged with `excess` should appear before the dummies.

5. Truncate the resulting array: the first $B \cdot Z$ elements form Out, and the next $|\mathsf{In}|$ elements form the array Remain.

6. For every element in Out and Remain: if it is a filler, replace with a dummy; further, remove temporary tags such as `excess` and `normal` that are needed only internally by this algorithm.