Perfectly Oblivious (Parallel) RAM Revisited, and Improved Constructions

T-H. Hubert Chan
Department of Computer Science, University of Hong Kong, Hong Kong

Elaine Shi
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Wei-Kai Lin
Department of Computer Science, Cornell University, Ithaca, NY, USA

Kartik Nayak
Department of Computer Sciences, Duke University, Durham, NC, USA

Abstract

Oblivious RAM (ORAM) is a technique for compiling any RAM program to an oblivious counterpart, i.e., one whose access patterns do not leak information about the secret inputs. Similarly, Oblivious Parallel RAM (OPRAM) compiles a *parallel* RAM program to an oblivious counterpart. In this paper, we care about ORAM/OPRAM with *perfect security*, i.e., the access patterns must be *identically distributed* no matter what the program's memory request sequence is. In the past, two types of perfect ORAMs/OPRAMs have been considered: constructions whose performance bounds hold *in expectation* (but may occasionally run more slowly); and constructions whose performance bounds hold *deterministically* (even though the algorithms themselves are randomized).

In this paper, we revisit the performance metrics for perfect ORAM/OPRAM, and show novel constructions that achieve asymptotical improvements for all performance metrics. Our first result is a new perfectly secure OPRAM scheme with $O(\log^3 N/\log\log N)$ *expected* overhead. In comparison, prior literature has been stuck at $O(\log^3 N)$ for more than a decade.

Next, we show how to construct a perfect ORAM with $O(\log^3 N/\log\log N)$ *deterministic* simulation overhead. We further show how to make the scheme parallel, resulting in an perfect OPRAM with $O(\log^4 N/\log\log N)$ *deterministic* simulation overhead. For perfect ORAMs/OPRAMs with deterministic performance bounds, our results achieve *subexponential* improvement over the state-of-the-art. Specifically, the best known prior scheme incurs more than $\sqrt{N}$ deterministic simulation overhead (Raskin and Simkin, Asiacrypt'19); moreover, their scheme works only for the sequential setting and is *not* amenable to parallelization.

Finally, we additionally consider perfect ORAMs/OPRAMs whose performance bounds hold with high probability. For this new performance metric, we show new constructions whose simulation overhead is upper bounded by $O(\log^3 / \log\log N)$ except with negligible in N probability, i.e., we prove high-probability performance bounds that match the expected bounds mentioned earlier.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases perfect oblivious RAM, oblivious PRAM

Digital Object Identifier 10.4230/LIPIcs.ITC.2021.8

Related Version *Full Version*: `https://ia.cr/2020/604`

Funding *T-H. Hubert Chan*: This work was partially supported by the Hong Kong RGC under the grants 17200418 and 17201220.
Elaine Shi: This work was partially supported by NSF CNS-1601879, an ONR YIP award, and a Packard Fellowship.
Wei-Kai Lin: This work was supported by a DARPA Brandeis award.
Kartik Nayak: This work was partially supported by NSF Award 2016393.

Acknowledgements Author ordering is randomized.

1 Introduction

Oblivious RAM (ORAM) is an algorithmic construction that provably obfuscates a (parallel) program's access patterns. It was first proposed in the ground-breaking work by Goldreich and Ostrovsky [22, 21], and its parallel counterpart Oblivious Parallel ORAM (OPRAM) was proposed by Boyle et al. [9]. ORAM and OPRAM are fundamental building blocks for enabling various forms of secure computation on sensitive data, e.g., either through trusted-hardware [35, 19, 30, 28] or relying on cryptographic multi-party computation [24, 29]. Since the initial proposal, ORAM and OPRAM have attracted much interest from various communities, and there has been a line of work dedicated to understanding their asymptotic and concrete efficiencies. It is well-known [22, 21, 27] that any O(P)RAM scheme must incur at least a logarithmic *overhead* (also known as *simulation overhead*) in (parallel) runtime relative to the insecure counterpart. On the other hand, ORAM/OPRAM schemes with poly-logarithmic overhead have been known [22, 21, 23, 26, 36, 37, 38, 32], and the very recent exciting work of Asharov et al. [5] showed how to match the logarithmic lower bound in the sequential ORAM setting, assuming the existence of one-way functions and a computationally bounded adversary.[1] Throughout this paper, we use the standard notion of *simulation overhead* originally suggested by Goldreich and Ostrovsky [22, 21]: if the original RAM/PRAM's (parallel) runtime is T and the corresponding ORAM/OPRAM's (parallel) runtime is χT, we say that the ORAM/OPRAM has simulation overhead χ.

Motivation for perfectly secure ORAMs/OPRAMs. With the exception of very few works, most of the literature has focused on either *computationally* secure [22, 21, 23, 26, 11, 32, 5] or *statistically* secure [2, 36, 37, 15, 38] ORAMs. A computationally secure (or statistically secure, resp.) ORAM guarantees that for any two request sequences of the same length, the access patterns incurred are computationally (or statistically resp.) indistinguishable. Most known computationally secure or statistically secure schemes [22, 21, 36, 37, 38, 9, 13] suffer from a small failure probability that is *negligible in the ORAM's size henceforth denoted N* while achieving $\mathsf{poly}\log N$ overhead. If the ORAM/OPRAM's size is large, say, $N \geq \lambda$ for some desired security parameter λ, then the failure probability would also be negligible in the security parameter. Unfortunately, for small choices of N (e.g., $N = \mathsf{poly}\log\lambda$), these schemes actually give polylogarithmic overhead in security parameter λ (and not in N) to achieve a $\mathsf{negl}(\lambda)$ security failure probability – note that a $\mathsf{poly}\log\lambda$ overhead equals to $N^{\Theta(1)}$ for this parameter regime, and thus the dependence on N is undesirable. Even though at first sight, it seems like we might not care about the parameter regime when N is much smaller than λ; as it turns out, such a small-N ORAM/OPRAM (with polylogarithmic in N overhead) was needed in many scenarios, such as in the construction of searchable encryption schemes [18], oblivious algorithms [36, 32, 4, 5] including notably, the recent OptORAMa work [5] that constructed an optimal ORAM.

The study of *perfectly* secure ORAMs/OPRAMs is partly motivated by the aforementioned mismatch. Moreover, recall also that *perfect security* has long been a topic of interest in the multi-party computation and zero-knowledge proof literature [25, 20], and its theoretical importance widely-accepted. Historically, perfect security is viewed as attractive since 1) the security holds in any computational model even if quantum computers or other forms of computers can be built; and 2) perfectly secure schemes often have clean compositional properties. Therefore, another natural application of perfectly secure ORAM/OPRAM is to construct efficient perfectly secure, RAM-model MPC.

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

Does there exist a perfectly secure ORAM/OPRAM with $o(\log^3 N)$ overhead? Despite the sustained and lively progress in understanding the asymptotic overhead of computationally and statistically secure ORAMs/OPRAMs, our understanding of perfectly secure ORAMs/OPRAMs has been somewhat stuck. In general, few results are known in the perfect security regime: in 2011, Damgård et al. [17] first showed a perfectly secure ORAM scheme with $O(\log^3 N)$ expected simulation overhead and $O(\log N)$ server space *blowup*, where the space blowup is the ratio between the server space consumed by the scheme compared to the insecure space N. Recently, Chan et al. [12] show an improved and simplified construction that removed the $\log N$ server space blowup; and moreover, they showed how to extend the approach to the parallel setting resulting in a perfectly secure OPRAM scheme with $O(\log^3 N)$ expected overhead. There is no known super-logarithmic lower bound for perfect security, and thus we do not understand yet whether the requirement of perfect security would inherently incur more overhead than computationally secure ORAMs. Therefore, an exciting and extremely challenging open direction is to understand the exact asymptotic complexity of perfectly secure ORAMs and OPRAMs, that is, to seek a matching upper- and lower-bound. This is a very ambitious goal and in this paper, we aim to take the next natural step forward. Since all prior upper bounds seem stuck at $O(\log^3 N)$, we ask the following natural question: *does there exist an ORAM/OPRAM with $o(\log^3 N)$ asymptotic overhead?*

The large gap between expected and deterministic performance bounds. To achieve perfect security, the prior perfect ORAM/OPRAM constructions pay a price: specifically their algorithms are Las Vegas, and the stated $O(\log^3 N)$ overhead is in an *expected* sense. Their ORAMs can occasionally run longer than $O(\log^3 N)$ time if certain unlucky events happen (where the unlucky events are identically distributed for all inputs so that the scheme remains perfectly secure). Moreover, the smaller the choice of N, the more likely that the ORAM can run much longer than the expectation.

Raskin et al. [34] (Asiacrypt'19) recently pointed out that this issue was somewhat shoved under the rug in prior works on perfect ORAMs/OPRAMs, and they were the first to ask how to construct perfect ORAMs with *deterministic* performance bounds. To avoid confusion, note that all ORAM schemes with non-trivial efficiency must be randomized algorithms; however, their performance bounds can be made deterministic (i.e., deterministic performance bounds does not mean that the algorithm is deterministic). Raskin et al. showed a perfectly secure ORAM with a deterministic simulation overhead $O(\sqrt{N}\frac{\log N}{\log\log N})$ (assuming $O(1)$ client-side storage[2]). While conceptually interesting, in comparison with the $O(\log^3 N)$ schemes [17, 12], the price to obtain deterministic bounds seems high. Therefore, another natural question is, *does there exist perfectly secure ORAMs/OPRAMs with deterministic polylogarithmic overhead?*

1.1 Our Results and Contributions

Our contributions are two-fold. First, following Raskin et al., we make another effort at systematizing the performance metrics for perfect ORAM/OPRAMs. Besides expected and deterministic performance bounds, we additionally consider the notion of *high-probability* performance bounds (explained below). Second, we show novel perfect ORAM/OPRAM constructions with asymptotical performance improvements for all three types of performance metrics: expected, high-probability, and deterministic.

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

Revisiting the performance metrics for perfectly secure ORAMs/OPRAMs. We consider the following performance bounds for ORAM/OPRAMs:

1. *Expected performance bounds.* Suppose that the original RAM/PRAM runs in (parallel) time T, and the corresponding ORAM/OPRAM runs in *expected* (parallel) time $\chi \cdot T$, then we say that the ORAM/OPRAM satisfies expected simulation overhead (or expected overhead) χ.
2. *High-probability performance bounds.* Suppose that the original RAM/PRAM runs in (parallel) time T, and the corresponding ORAM/OPRAM runs in (parallel) time $\chi \cdot T$ with $1 - \delta$ probability where δ is suitably small (e.g., negligibly small in some security parameter), then, we say that ORAM/OPRAM satisfies simulation overhead (or overhead) χ with probability $1 - \delta$. We stress that the failure probability δ describes the probability that the ORAM/OPRAM exceeds the performance bounds, it does *not* describe security failure since the ORAM/OPRAM is perfectly secure. This is a natural, intermediate notion that is not as stringent as deterministic performance bounds and yet gives a strong guarantee. This notion may permit asymptotically better schemes than insisting on deterministic performance bounds.
3. *Deterministic performance bounds.* Suppose that the original RAM/PRAM runs in (parallel) time T, and the corresponding ORAM/OPRAM runs in (parallel) time $\chi \cdot T$ with probability 1, then we say that the ORAM/OPRAM satisfies deterministic simulation overhead χ.

Asymptotically better constructions for all performance metrics. We show novel perfect ORAM/OPRAM constructions that achieve asymptotical performance improvements across the board.

- First, for expected performance, we overcome the $\log^3 N$ barrier that the literature has been stuck at for the past decade. Our perfect ORAM/OPRAM scheme has $O(\log^3 N / \log\log N)$ expected overhead.
- Second, for high-probability performance bounds, previously, there were no documented schemes with this type of performance bounds to the best of our knowledge. We show new perfectly secure ORAM/OPRAMs that achieve $O(\log^3 N / \log\log N)$ simulation overhead with probability $1 - \mathsf{negl}(N)$.[3]
- Finally, we construct perfect ORAM/OPRAMs with deterministic, poly-logarithmic simulation overhead. Our result achieves a sub-exponential performance improvement relative to prior art [34].

Our results are summarized in the following theorems, and moreover, Table 1 gives an explicit comparison of our results with prior work.

▶ **Theorem 1** (Informal: perfect OPRAM with expected performance bounds)**.** *There exists a perfectly secure OPRAM scheme that consumes only $O(1)$ blocks of client private cache and $O(N)$ blocks of server-space; moreover the scheme achieves $O(\log^3 N / \log\log N)$ expected simulation overhead.*

▶ **Theorem 2** (Informal: perfect OPRAM with high-probability performance bounds)**.** *There exists a perfectly secure OPRAM scheme that consumes only $O(1)$ blocks of client private cache and $O(N)$ blocks of server-space; moreover the scheme achieves $O\Big(\frac{1}{\log\log N} \cdot (\log^3 N + \log^2 N \cdot \log^2 \log(1/\delta) + \log N \cdot \log^3 \log(1/\delta))\Big)$ simulation overhead with probability $1 - \delta$.*

[3] Note that our formal theorem statement gives a parametrized version where the performance failure probability may be any free parameter.

Table 1 Comparison of our results with prior work. For simplicity, the high-probability bounds are parameterized to $1 - \mathsf{negl}(N)$ failure probability.

		Space	ORAM overhead	OPRAM overhead
Schemes with **expected** performance bounds	Damgård et al. [17]	$O(N \log N)$	$O(\log^3 N)$	N/A
	Chan et al. [12]	$O(N)$	$O(\log^3 N)$	$O(\log^3 N)$
	This work	$O(N)$	$O(\log^3 N / \log\log N)$	$O(\log^3 N / \log\log N)$
Schemes with **high-probability** performance bounds	**This work**	$O(N)$	$O(\log^3 N / \log\log N)$	$O(\log^3 N / \log\log N)$
Schemes with **deterministic** performance bounds	Raskin et al. [34]	$O(N)$	$O(\sqrt{N} \cdot \frac{\log N}{\log\log N})$	N/A
	This work	$O(N)$	$O(\log^3 N / \log\log N)$	$O(\log^4 N / \log\log N)$

▶ **Theorem 3** (Informal: perfect OPRAM with deterministic performance bounds)**.** *There exists a perfectly secure ORAM scheme that achieves $O(\log^3 N / \log\log N)$ simulation overhead with probability 1. For the parallel setting: there exists a perfectly secure OPRAM scheme that achieves $O(\log^4 N / \log\log N)$ simulation overhead with probability 1.*

For both the ORAM/OPRAM schemes above, we need only $O(1)$ blocks of client private cache and $O(N)$ blocks of server-space.

1.2 Technical Highlight

Getting an $O(\log^3 N / \log\log N)$ deterministic overhead ORAM. To improve the overhead of perfectly secure ORAMs to $O(\log^3 N / \log\log N)$, our techniques are inspired by the rebalancing trick of Kushilevitz et al. [26] (SODA'12), and yet departs significantly from Kushilevitz et al. Namely, the existing perfect ORAM/OPRAM constructions of Chan et al. [12] consist of an "online fetch phase" and an "offline maintain phase," the fetch phase takes a logical request (as an input to the ORAM) and answers to the request using a data structure, and then the maintain phase reshuffles the data structure. We observe that the maintain and fetch phases suffer from an imbalance; specifically, the offline maintain phase costs $O(\log^3 N)$ per request whereas the online fetch phase costs only $O(\log^2 N)$. A natural idea is to modify the scheme and rebalance the costs of the offline maintain phase and the online fetch phase, such that both phases would cost only $O(\log^3 N / \log\log N)$. Unfortunately, existing techniques such as Kushilevitz et al. completely fail for rebalancing *perfect* ORAMs/OPRAMs – we defer the technical reasons to the full version [14].

Our starting point is the perfect ORAM construction by Chan et al. [12] in which the maintain phase costs $O(\log^3 N)$ and the fetch phase costs only $O(\log^2 N)$. Specifically, their construction consists of $D = O(\log N)$ number of ORAMs such that except for the last ORAM which stores the actual data blocks, every other ORAM serves as a (recursive) index structure into the next ORAM – for this reason, these D ORAMs are also called D recursion depths; and all of the recursion depths jointly realize an implicit logical index structure that is in fact isomorphic to a binary tree (which has a branching factor of 2).

First, we show how to use a *fat-block* trick to increase the branching factor and hence reduce the number of recursion depths by a $\log\log N$ factor. Specifically, we increase the implicit index structure's branching factor from 2 (i.e., storing two pointers or position labels in the next recursion depth) to $\log N$. Thus a fat-block is a bundle of *logarithmically* many normal blocks and hence each fat-block can store logarithmically many pointers. While this reduces the recursion depth by a $\log\log N$ factor, the fetch phase now costs a logarithmic factor more per recursion depth (since obliviously accessing a fat-block is a logarithmic factor more costly than accessing a normal block).

The primary challenge is to realize the maintain phase such that the amortized *per-depth* maintain-phase cost preserves the asymptotics, despite the fat-block now being logarithmically fatter. To accomplish this we rely on the following two key insights:

1. *Exploit residual randomness.* First, we rely on an elegant observation first made in the PanORAMa work [32] in the context of computationally secure ORAMs. Here we make the same observation for perfectly secure ORAMs. At the core of Chan et al.'s ORAM construction is a data structure called an oblivious "one-time-memory" (OTM). When an OTM is initialized, all elements in it are randomly permuted (and the randomness concealed from the adversary) – note that in our setting, each element is a fat-block. The critical observation is that after accessing a subset of the elements in this OTM data structure, the remaining unvisited elements still appear in a random order. By exploiting such residual randomness, when we would like to build a new OTM consisting of the remaining unvisited elements, we can avoid performing expensive oblivious permutation (which would take time $O(n \log n)$ to sort n elements) and instead rely on linear-time operations.
2. *Exploit sparsity.* In the construction of Chan et al., the D ORAMs at all recursion depths must perform a "coordinated shuffle" operation during the maintain phase. An important step in this coordinated shuffle is (for each recursion depth) to inform the parent depth the locations of its fat-blocks after the reshuffle. In Chan et al., two adjacent recursion depths perform such "communication" through oblivious sorting, thus incurring $O(n \log n)$ cost per-depth to rebuild a data structure of size n.
   Our key observation is that the fat-blocks contained in each OTM data structure in each recursion depth are sparsely populated. In fact, most entries in the fat-blocks are irrelevant and only a $1/\log N$ fraction of them are populated. Thus, we employ an oblivious tight compaction [5] to compress away the wasted space, where tight compaction is a degenerated sorting that sorts elements tagged with 0/1 keys. After this compression, the OTM becomes logarithmically smaller and we can apply an oblivious sort.

Finally, we stress that to get an ORAM with *deterministic* performance bounds, we will need to instantiate our ORAM with building blocks that give deterministic performance. We defer the details to later technical sections.

Parallelizing the scheme. To parallelize our ORAM scheme, we encounter several additional technicalities. Some of the core algorithmic building blocks can be parallelized; however, to preserve the asymptotical total work of the sequential versions, the only known parallel counterparts give Las Vegas-type performance bounds. This would be fine if we only wanted an OPRAM with *expected* $O(\log^3 N/\log\log N)$ overhead. However, more work is needed to get an OPRAM whose simulation overhead is $O(\log^3 N/\log\log N)$ *with high probability*. Finally, to get an OPRAM whose simulation overhead holds deterministically, we need to replace some of the oblivious parallel building blocks with ones with deterministic performance bounds – here we lose an extra logarithmic factor in total work. We defer a more detailed exposition of these technicalities to later sections.

2 Technical Overview

We start with an informal and intuitive exposition of our main technical ideas. For simplicity, most of the section describes how to get the sequential ORAM result with *deterministic* $O(\log^3 N/\log\log N)$ simulation overhead. Then, in Section 2.4, we sketch the additional steps and technicalities needed to parallelize the scheme to get the expected, high-probability, and deterministic performance bounds for OPRAM. The full formal details will be deferred to the technical sections later.

2.1 Background on Perfect ORAM

The goal of an ORAM scheme is to simulate to the client a memory array of N blocks, where each block consists of $\Omega(\log N)$ bits. In the simulated memory, each block is indexed by a *logically address* in $\{0, 1, \ldots, N-1\}$, and the client can read or write a block using a logical address. The ORAM is allowed to use client-side storage of only $O(1)$ number of blocks as well as server storage, where the server supports only fetch or store the content of blocks (but no computation). By perfect security, we require that the sequence of accessed blocks on the server (also called the *access pattern*) be identically distributed for any sequence of read/write accesses to the memory simulated by ORAM. Such settings are standard in previous works [17, 12, 34].

In a recent work, Chan et al. [12] propose a perfectly secure ORAM with $O(\log^3 N)$ simulation overhead. At a high level, their scheme is inspired by the hierarchical ORAM paradigm by Goldreich and Ostrovsky [22, 21], but Chan et al. replace the "oblivious hashing" (which has a negligible statistical imperfectness) with perfectly secure data structures (including a "one-time-memory" as well as a"position map"). In this way, they also remove the pseudo-random function (PRF) in Goldreich and Ostrovsky's construction [22, 21].

2.1.1 Position-based Hierarchical ORAM

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

Hierarchical levels. The server-side data structure is organized into $\log N+1$ levels numbered $0, 1, \ldots, \log N$ where level i is either 1. *empty*, in which case it stores no blocks; or 2. *full*, in which case the level stores 2^i real blocks plus 2^i dummy blocks in a randomly permuted fashion (we also say that the level has *capacity* 2^i). Each block, whose logical addresses range from 0 to $N-1$, resides in exactly one of the levels at a random position within the level.

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

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

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

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

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

Chan et al. show how to achieve the above obliviously – even though the client has only $O(1)$ blocks of client storage – through oblivious sorting (using the AKS sorting network [1]). The cost of rebuilding a level of capacity n is dominated by the oblivious sorting on $O(n)$ blocks, which has a cost of $O(n \log n)$.

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

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

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

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

2.1.2 Recursive Position Map

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

As Chan et al. show, in the hierarchical ORAM context such a recursion idea does not work in a straightforward blackbox manner,[4] but needs a special "coordinated rebuild" technique which we now explain. Henceforth, suppose that each block's logical address addr

[4] Roughly speaking, it is because each logical access on ORAM_{i+1} would have incurred too many accesses on ORAM_i, and then the cost of such recursion would have been too expensive.

is $\log N$ bits long, and we use the notation $\mathsf{addr}^{\langle d \rangle}$ to denote the address addr, written in binary format, truncated to the most significant d bits.

- *Fetch phase (straightforward):* To fetch a block at some logical address addr, the client looks up logical address $\mathsf{addr}^{\langle d \rangle}$ in each ORAM_d for $d = 0, 1, \ldots D$ sequentially. Since the block at logical address $\mathsf{addr}^{\langle d \rangle}$ in ORAM_d stores the position labels for the two blocks at logical addresses $\mathsf{addr}^{\langle d \rangle} \| 0$ $\mathsf{addr}^{\langle d \rangle} \| 1$ in ORAM_{d+1}, the client is always able to find out the position of the block in the next recursion depth before it performs a lookup there.
- *Maintain phase (coordinated rebuild):* The maintain phase needs special treatment such that the rebuilds at all recursion depths are coordinated. Specifically, whenever the data ORAM_D is rebuilding the level ℓ, each other recursion depth ORAM_d would be rebuilding level $\min(\ell, d)$ in a coordinated fashion – note that each ORAM_d has only d levels.
  The main goal of the coordination is for each ORAM_d to pass the blocks' updated position labels back to the parent depth ORAM_{d-1}. More specifically, recall that when ORAM_d rebuilds a level ℓ, all real blocks in the level would now be placed at a new random position. When these new positions have been decided, ORAM_d must inform the corresponding metadata blocks in ORAM_{d-1} the new position labels. The coordinated rebuild is possible due to the following invariant which is not hard to observe (recall that $\mathsf{addr}^{\langle d \rangle}$ is the block that stores the position labels for the block $\mathsf{addr}^{\langle d+1 \rangle}$ in ORAM_{d+1}):
  - For every addr, the block at address $\mathsf{addr}^{\langle d \rangle}$ in ORAM_d is always stored at a smaller or equal level relative to the level of the block at address $\mathsf{addr}^{\langle d+1 \rangle}$ in ORAM_{d+1}.

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

2.1.3 Analysis

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

To asymptotically improve the overhead, one promising idea is to somehow balance the fetch and maintain phases. This idea has been explored in computationally secure ORAMs first by Kushilevitz et al. [26] and later improved in subsequent works [11]. Unfortunately as we explain the full version [14], Kushilevitz et al.'s rebalancing trick is not compatible with known perfect ORAMs. Thus we need fundamentally new techniques for realizing such a rebalancing idea.

2.2 Building Blocks

Before we introduce our new algorithms, we describe two important oblivious algorithms as building blocks that were discovered in very recent works [33, 5].

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

In a very recent work called OptORAMa [5], the authors show how to accomplish tight compaction obliviously in $O(m)$ time. Their algorithm can be expressed as a linear-sized circuit (of constant fan-in and fan-out), consisting only of boolean gates and swap gates, where a boolean gate can perform boolean computations on two input bits; and a swap gate takes in a bit and two balls, and decides to either swap or not swap the two balls.

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

2.3 A New Rebalancing Trick for Perfectly Secure ORAMs

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

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

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

Our hope is that now with the smaller recursion depth, we can accomplish the maintain phase in amortized $O(\log^3 N / \log\log N)$ cost. Recall that each level ℓ in a metadata ORAM_d now contains 2^ℓ fat-blocks. The crux is to rebuild a level containing 2^ℓ fat-blocks in cost that is linear in the level's total size, that is, $2^\ell \cdot \chi$. Note that if we naïvely used oblivious sorting on fat-blocks (like in Section 2.1.1) to accomplish this, the cost would have been $2^\ell \cdot \chi \cdot \log(2^\ell)$ which is more expensive than previous scheme and undesirable.

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

- *Sparsity:* First, observe that each level in a metadata ORAM is *sparsely populated*: although the entire level, say, level ℓ, has the capacity to store $2^\ell \cdot \chi$ position labels, the level is rebuilt after every 2^ℓ requests. Thus in fact only 2^ℓ of these position label entries are populated.
- *Residual randomness:* The second important observation is that the unvisited fat-blocks contained in any level appear in a random order where the randomness of the permutation is hidden from the adversary – note that a similar observation was first made in the PanORAMa work [32] by Patel et al.

  More specifically, suppose that to start with, a level contains n fat-blocks including some reals and some dummies, and all of these n fat-blocks have been randomly permuted (where the randomness of the permutation is hidden from the adversary). As the client

visits fat-blocks in a level, the adversary learns which blocks are visited. Now, among all the unvisited blocks, there are both real and dummy blocks and all these blocks are equally likely to appear in any order w.r.t. the adversary's view.

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

1. *Merge.* The first step of the rebuild is to merge consecutively full levels into the next empty level (or the largest level). After this merge step, this new level is marked full and every smaller level is marked empty.
2. *Permute.* After the above merge step, the resulting array containing n fat-blocks must be randomly permuted (and their positions after the permutation will then be passed to the parent depth next).
3. *Update.* After the permutation step, each real fat-block in the level (at a recursion depth d) whose logical address is addr must receive up to χ updated positions from the child recursion depth, i.e., the fat-block at logical address addr wants to learn where the fat-blocks at logical addresses $\mathsf{addr}||0$, $\mathsf{addr}||1$, …, $\mathsf{addr}||(\chi - 1)$ newly reside in the child depth $d + 1$.
4. *Create dummy metadata.* Finally, create a dummy metadata array to accompany this level: the dummy metadata array containing n entries where each entry is $O(\log N)$ bits (note that an entry is a normal block, not a fat-block). This array should store the positions of all *dummy* fat-blocks contained in the level in a randomly permuted order.

Realizing "merge + permute". We first explain how to accomplish the "merge + permute" steps. For simplicity we focus on explaining the case where consecutive full levels are merged into the next empty level (since it would be fine if the merging into the largest level *alone* is done naïvely using oblivious sort on all fat-blocks). Here it is important to rely on the residual randomness property mentioned earlier. Suppose the levels to be merged contain $1, 2, 4, 8, \ldots, n/2$ fat-blocks respectively. Recall that in all of these levels to be merged, the unvisited blocks appear in a random order w.r.t. the adversary's view. Thus, we can simply do $O(\log n)$ cascading merges using **Intersperse** (see Section 2.2), every time merging two arrays each containing 2^i fat-blocks into an array containing 2^{i+1} fat-blocks, and the overall cost is $O(n)$.

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

Recall that the problem we need to solve boils down to the following. There is a destination array D consisting $O(n \cdot \chi)$ position entries among which $O(n)$ entries are going to be updated, and we override terminologies "real" and "dummy" (opposed to previously denoted real or dummy fat-blocks) and say the to-be-updated $O(n)$ entries are *real* while all remaining entries are *dummy*. Additionally, there is a source array S consisting of $O(n)$ entries (which can be real or dummy). In both the source S and the destination D, each real entry is of the form (k, v) where k denotes a key and v denotes a payload value; further, in each array D or S, every real entry must have a distinct key. Now, we would like to route each real entry $(k, v) \in S$ to the corresponding entry with the same key in the destination array D.

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

First, we rely on oblivious tight compaction to compact the destination array D, resulting in a compacted array $\widetilde{D}$ consisting of only $O(n)$ entries. Moreover, recall that oblivious tight compaction can be viewed as a *circuit* consisting of boolean gates and swap gates. When we compact the destination array D, each swap gate remembers the routing decision since later it will be useful to run this circuit in the reverse direction. After the compaction, we can now afford to pay the cost of oblivious sorting. Hence, each entry in the source S can route itself to each entry in the compacted destination $\widetilde{D}$ – this can be accomplished through a standard technique called oblivious routing [9, 13], which has a cost of $O(n \log n)$. Now, by running the aforementioned tight compaction circuit in the reverse direction, we can route each element of the compacted destination $\widetilde{D}$ back into the original destination array D.

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

Obliviously create dummy metadata array. Finally, obliviously creating the dummy metadata array is easy: this can be accomplished by writing down $O(\log N)$ bits of metadata per fat-block, and then performing a combination of oblivious random permutation and oblivious sort on the resulting metadata array. To get deterministic simulation overhead for our ORAM, we will need to use an oblivious random permutation algorithm with deterministic performance bounds – fortunately, this is known due to Asharov et al. [5].

Summary. In the above, we took care to make sure that all oblivious building blocks used give deterministic performance bounds. At this moment, we derive a perfect ORAM scheme with $O(\log^3 N / \log \log N)$ *deterministic* overhead. This warmup result already improves upon Chan et al. [12] who showed $O(\log^3 N)$ *expected* simulation overhead, as well as Raskin [34] who showed roughly $\sqrt{N}$ *deterministic* simulation overhead.

2.4 Parallelizing the Scheme

So far, for simplicity we have focused on the sequential case. To obtain our OPRAM result, we need to make the above scheme parallel. To this end, we will rely on the OPRAM techniques by Chan et al. [12], that is, the fetch phase is still performed sequentially, but the maintain phase is realized using parallel and oblivious sorting, tight compaction, and random permutation. One main challenge here is that we will need a parallel counterpart for the **Intersperse** algorithm. Note that Asharov et al. [5]'s **Intersperse** algorithm gives deterministic performance bounds and perfect security, but is inherently sequential. We devise a new parallel **Intersperse** algorithm that preserves the same asymptotical total work as the sequential version of Asharov et al. [5] (for fat-blocks); however, the algorithm gives Las-Vegas-type performance. This is fine if we only aim for an $O(\log^3 / \log \log N)$-*expected*-overhead OPRAM, but it will not work if we want matching high probability performance bounds. Observe that our parallel **Intersperse** algorithm's performance bounds are more concentrated around the mean for larger input sizes. In our OPRAM, the **Intersperse** algorithm needs to be applied to input arrays containing $1, 2, 4, \ldots, N/\log N$ fat blocks. Therefore, to get $O(\log^3 / \log \log N)$ overhead with $1 - \mathsf{negl}(N)$ probability, our idea is to apply the Las-Vegas **Intersperse** algorithm only to sufficiently large instances, and for small instances, we apply a variant which gives deterministic performance bounds but is a logarithmic factor more expensive. We prove that this bi-modal approach gives an OPRAM scheme whose simulation overhead is $O(\log^3 / \log \log N)$ with $1 - \mathsf{negl}(N)$ probability.

Finally, to get an OPRAM with deterministic performance bounds, we need to replace the **Intersperse** building block entirely with one that gives deterministic performance bounds, but is a logarithmic factor more expensive. This explains why our OPRAM with deterministic performance bounds has an extra logarithmic factor.

2.5 Open Questions

Our paper raises several interesting open questions. First, for constructions with deterministic performance bounds, currently our OPRAM scheme has a logarithmic factor higher simulation overhead than the sequential ORAM – this extra logarithmic factor stems from the parallel perfect oblivious permutation building block we use [3]. One open question is whether we can get rid of this extra logarithmic factor.

In our paper, we adopt the standard notion of simulation overhead originally defined by Goldreich and Ostrovsky [22, 21]. This standard notion is naturally *amortized* over the multiple steps of the RAM/PRAM, since it takes the ratio of the total runtime of the ORAM/OPRAM and that of the original RAM/PRAM. In comparison, some earlier works consider an even stronger notion, often referred to as *worst-case* overhead [31, 36, 26, 11, 34]: a worst-case overhead of χ requires that *every* (parallel) step of the original RAM/PRAM is simulated by at most χ steps in the compiled ORAM/OPRAM. While some previous ORAM/OPRAM constructions are amenable to a standard deamortization trick [31, 26] to achieve worst-case overhead that match the amortized, our constructions are not compatible with standard deamortization techniques [26] for a similar reason why PanORAMa [32] and OptORAMa [5] are also not compatible with standard deamortization. It is due to the "residual randomness" technique: after the residual randomness of an element is used to facilitate a random permutation, if the same element is then accessed due to the deamortization, then such residual randomness is revealed and the random permutation is no longer random in the adversarial view, which is insecure. An interesting future direction is whether we can achieve worst-case overhead that match the amortized bounds claimed in our paper.

Our paper focuses on the *theoretical* understanding of the asymptotic complexity of perfectly secure ORAMs/OPRAMs. Our asymptotic constant is huge due to using AKS sorting network [1] and the linear tight compaction [6]. The constants are similar to earlier works [17, 12] that also use AKS sorting. A standard way to replace the huge constant with another logarithmic factor is to replace both AKS sorting and tight compaction with bitonic sorter [8] (notice that the standard bitonic sort takes $O(n \log^2 n)$ work, but to sort only 0/1 elements, i.e. tight compaction, a bitonic sort augmented with counting takes only $O(n \log n)$ work). An interesting question is whether we can achieve the performance bounds claimed in this paper, but without the use of expander graphs with large constants.

Last but not the least, for perfectly secure ORAM/OPRAM (and in fact even for statistically secure ORAM/OPRAM), we still do not have matching upper- and lower-bounds. Therefore, a more challenging direction is to close this obvious gap in our understanding.

2.6 Roadmap of Subsequent Formal Sections

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

3 Preliminaries

3.1 Definitions

3.1.1 Parallel Random-Access Machines

We review the concepts of a parallel random-access machine (PRAM) and an oblivious parallel random-access machine (OPRAM). The definitions in this section are borrowed from Chan et al. [12]. Although we give definitions only for the parallel case, we point out that this is without loss of generality, since a sequential RAM can be thought of as a special case PRAM with one CPU.

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

We consider a PRAM model where the number of CPUs is fixed to be some parameter $m \leq N$[5]. Each CPU has a state that stores $O(1)$ blocks. In each step, each CPU executes a next instruction circuit denoted Π, and then interacts with memory. Circuit Π can perform word-level operations including addition, subtraction, and bit-wise boolean operations in unit time and then updates the CPU state. Further, CPUs interact with memory through request instructions $\vec{I}^{(t)} := (I_i^{(t)} : i \in [m])$. Specifically, at time step t, CPU i's instruction is of the form $I_i^{(t)} := (\mathsf{read}, \mathsf{addr})$, or $I_i^{(t)} := (\mathsf{write}, \mathsf{addr}, \mathsf{data})$ where the operation is performed on the memory block with address addr and the block content data.

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

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

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

CPU-to-CPU communication. In the remainder of the paper, we sometimes describe our algorithms using CPU-to-CPU communication. For our OPRAM algorithm to be oblivious, the inter-CPU communication pattern must be oblivious too. We stress that such inter-CPU communication can be emulated using shared memory reads and writes. Therefore, when we express our performance metrics, we assume that all inter-CPU communication is implemented with shared memory reads and writes.

[5] If $N < m$, the oblivious simulation can be achieved by assigning at most one address to each CPU and then performing oblivious routing [9], which takes only $O(\log m)$ overhead.

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

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

where Π denotes the next instruction circuit, N denotes the total memory size (in terms of number of blocks), T denotes the PRAM's total runtime, and m denotes the number of CPUs.

Finally, in this paper, we consider PRAMs that are *stateful* and can evaluate a sequence of inputs, carrying states in between, where each input can be stored in a single memory block.

3.1.2 Oblivious Parallel Random-Access Machines

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

Randomized PRAM. A *randomized PRAM* is a PRAM where the CPUs are allowed to generate private random numbers. Concretely, we assume that the next instruction circuit Π can sample a uniform random number from $[a]$ for any positive integer $a \leq 2^w$ in unit time (recall that w is the memory word size in bits), where the assumption is needed by the oblivious random permutation (e.g., [3]). For simplicity, we assume that a randomized PRAM has a priori known, deterministic runtime, and that the CPU activation pattern in each time step is also fixed a priori and publicly known.

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

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

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

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

We remark that for statistical and computational security, some earlier works [11, 13] presented an adaptive, composable security notion. The perfectly oblivious counterpart of their adaptive, composable notion is equivalent to our notion defined above. In particular, our notion implies security against an adaptive adversary who might choose the input sequence inp adaptively over time after having observed partial access patterns of PRAM.

We say that OPRAM is a *perfectly oblivious simulation* of PRAM iff OPRAM is perfectly oblivious, and moreover $\mathsf{OPRAM}(\mathsf{inp})$ is identically distributed as $\mathsf{PRAM}(\mathsf{inp})$ for any input inp.

Metrics. We will use the standard notion of *simulation overhead* to characterize an OPRAM's performance [22, 21, 9]. If a PRAM that consumes m CPUs and completes in T parallel steps can be obliviously simulated by an OPRAM that completes in $\gamma \cdot T$ steps also with m CPUs, then we say that the simulation overhead is γ. Moreover, supposing that the OPRAM is randomized (by a random tape that is independent from the PRAM), and letting the OPRAM completes in T' steps with m CPUs where T' is a random variable, we say that the *expected* simulation overhead is γ if $\mathbb{E}[T'] = \gamma T$, and we say that the simulation overhead is γ *with probability* $1-\delta$ if $\Pr[T' \leq \gamma T] \geq 1-\delta$. We additionally say the simulation overhead γ is *deterministic* if it is γ with probability 1, which coincides with the standard simulation overhead.

More generally, suppose that an ample (i.e., unbounded) number of CPUs are available: in this case if algorithm can be completed in T parallel steps consuming $m_1, m_2, \ldots, m_T$ CPUs in each step respectively, then we say that the algorithm can be completed in T *depth* and $W := \sum_{t \in [T]} m_t$ *total work*. Similar to that of simulation overhead, when the total work and depth are random variables, we quantify the total work and depth using expected, with probability, or deterministic, where deterministic is sometimes omitted.

Therefore, for an OPRAM, if the original PRAM (taking T parallel steps and using m CPUs) can be obliviously simulated in W' total work and $T' = O(W'/m)$ depth then the OPRAM has simulation overhead W'/Tm.

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

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

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

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

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

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

3.2 Oblivious Algorithm Building Blocks

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

3.2.1 Oblivious Sort

Oblivious sorting can be accomplished through a sorting network such as the famous construction by Ajtai, Komlós, and Szemerédi [1]. We restate this result in the context of PRAM algorithms:

▶ **Theorem 4** (Oblivious sorting [1]). *There exists a deterministic, oblivious algorithm that sorts an array of n elements consuming $O(\beta \cdot n \log n)$ total work and $O(\log n)$ depth where $\beta \geq 1$ denotes the number of memory words it takes to represent each element.*

3.2.2 Oblivious Random Permutation

Let ORP be an algorithm that upon receiving an input array X, outputs a permutation of X. Let $\mathcal{F}_{\text{perm}}$ denote an ideal functionality that upon receiving the input array X, outputs a perfectly random permutation of X. We say that ORP is a *perfectly oblivious* random permutation, iff it is a perfect oblivious simulation of the functionality $\mathcal{F}_{\text{perm}}$. Recall that for any integer $m \in [n]$, each CPU of the PRAM can sample an integer uniformly at random from $[m]$ in unit time.

Sequential ORP algorithm with deterministic performance bounds. Recently, a sequential oblivious algorithm is developed to perform such permutation in $O(n \log n)$ total work [5, Section 6.4].

▶ **Theorem 5** (A sequential ORP algorithm [5]). *Let $\beta \geq 1$ denote the number of memory words it takes to represent each element. There exists an oblivious random permutation construction that completes in deterministic $O(\beta \cdot n \log n)$ total work.*

Parallel ORP algorithm with deterministic performance bounds. Alonso and Schott [3] construct a parallel random permutation algorithm that takes $O(n \log^2 n)$ total work and $O(\log^2 n)$ depth to randomly permute n elements. Although achieving obliviousness was not a goal of their paper, it turns out that their algorithm is also perfectly oblivious, giving rise to the following theorem:

▶ **Theorem 6** (Alonso-Schott ORP). *There is a perfectly oblivious algorithm that permutes an array of n elements in deterministic $O(\beta \cdot n \log n + n \log^2 n)$ total work and $O(\log^2 n)$ depth where $\beta \geq 1$ denotes the number of memory words for representing each element.*

Parallel, Las Vegas ORP algorithm. A few recent works [10, 5] describe another perfectly oblivious random permutation algorithm which is asymptotically more efficient but the algorithm is Las Vegas, i.e., the algorithm satisfies perfect obliviousness and correctness, but

with a small probability the algorithm may run longer than the stated bound.[6] Below, we restate this result in the form that we desire in this paper – the specific theorem stated below arises from the improved analysis of Asharov et al. [5, Theorem 4.3] where we replace the "quadratic oblivious random permutation" with Alonso-Schott ORP; for the performance bounds, we state an *expected* version and a *high-probability* version. Notice that the replaced ORP incurs an $O(\log^2 n)$ depth with probability $o(1)$ but not in expectation.

▶ **Theorem 7** (A Las Vegas ORP algorithm)**.** *Let $\beta \geq 1$ denote the number of memory words it takes to represent each element. There exists a Las Vegas perfectly oblivious random permutation construction that completes in expected $O(\beta \cdot n \log n)$ total work and expected $O(\log n)$ depth. Furthermore, except with $n^{-\Omega(\sqrt{n})}$ probability, the algorithm completes in $O(\beta \cdot n \log n)$ total work and $O(\log^2 n)$ depth.*

Note that the above theorem gives a high-probability performance bound for sufficiently large n. Later in our OPRAM construction, we will adopt ORP for problems of different sizes – we will use Theorem 7 for sufficiently large instances and use Theorem 6 for small instances.

3.2.3 Oblivious Routing

Oblivious routing [9] 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 no more than C times in the src array where $C = O(1)$ is a known constant; however, each (non-dummy) key can appear any number of times in dst.
- *Outputs:* We would like to output an array $\mathsf{Out} := \{v'_{i,j}\}_{i\in[n'],j\in[C]}$ where $(v'_{i,1}, \ldots, v'_{i,C})$ contains all the values contained in src whose keys match k'_i (padded with $\perp$ to length C).

▶ **Theorem 8** (Oblivious routing [9, 13, 10])**.** *There exists a perfectly oblivious routing algorithm that accomplishes the above task in $O(\log(n + n'))$ depth and $O(\beta \cdot (n + n') \log(n + n'))$ total work where $\beta \geq 1$ denotes the number of words it takes to represent each element.*

3.2.4 Oblivious Tight Compaction

As mentioned in Section 2.2, tight compaction is the following task: given an input array containing n elements where each element is tagged with a bit indicating whether it is real or dummy, produce an output array containing also n elements such that all real elements in the input appear in the front and all dummies appear at the end. We will use the parallel oblivious tight compaction of Asharov et al. [6] running in linear work and logarithmic depth.

▶ **Theorem 9** (Oblivious tight compaction [6])**.** *There exists a deterministic, oblivious tight compaction algorithm that compacts an array of n elements in total work $O(\beta \cdot n)$ and depth $O(\log n)$ where $\beta \geq 1$ denotes the number of words it takes to represent each element.*

[6] Using more depth but only unbiased random bits, Czumaj [16] shows a Las Vegas switching network to achieve the same abstraction.

We point out that Asharov et al.'s oblivious parallel compaction algorithm [6] works in the so-called *indivisibility* model, that is, the payload of the elements are moved around as opaque strings.

3.3 Parallel Intersperse

Oblivious intersperse is an abstraction that which can be used to mix two input arrays such that the mixing is uniformly at random in the adversarial view. The abstraction was originated in PanORAMa [32] and then formally defined and realized in OptORAMa [5]. In this section, we define intersperse for completeness and then state the sequential and parallel realizations that run in expected, high probability, or deterministic performance bounds.

3.3.1 Definition

Informally, in the definition of OptORAMa, the **Intersperse** algorithm receives the concatenation of the two input arrays and only the sum of their lengths is public but not each array's individual length where each input array is shuffled uniformly at random, and then **Intersperse** is required to output a uniformly shuffled array consisting of all input elements. More specifically, **Intersperse** has the following syntax.

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

We now define the security notion required for **Intersperse**. We require that when we run **Intersperse** on two input arrays $\mathbf{I}_0$ and $\mathbf{I}_1$ that are both randomly shuffled (based on a secret permutation), the resulting array will be randomly shuffled (based on a secret permutation) too. More formally stated, we require that **Intersperse** is a perfect oblivious simulation of the following $\mathcal{F}_{\text{shuffle}}(\mathbf{I}_0, \mathbf{I}_1)$ functionality provided that the two input arrays are randomly shuffled. Henceforth we assume that the bit-width of each element in the input arrays is a publicly known parameter that the scheme is implicitly parametrized with.

$\mathcal{F}_{\text{shuffle}}(\mathbf{I}_0 \| \mathbf{I}_1, n_0, n_1)$:
1. Choose a permutation $\pi : [n] \to [n]$ uniformly at random where $n := |\mathbf{I}_0| + |\mathbf{I}_1|$.
2. Let $\mathbf{I}$ be the concatenation of $\mathbf{I}_0 \| \mathbf{I}_1$.
3. Initialize an array $\mathbf{B}$ of size n. Assign $\mathbf{B}[i] := \mathbf{I}[\pi(i)]$ for every $i = 1, \ldots, n$.
4. **Output:** The array $\mathbf{B}$.

The recent work OptORAMa [5, Claim 6.3] showed how to construct an **Intersperse** algorithm in linear time, i.e., $O(n)$; however, their algorithm is inherently sequential (see the following warmup). A manuscript by Asharov et al. [7] considered how to devise a parallel version of **Intersperse** in an attempt to make OptORAMa parallel; but their parallel **Intersperse** algorithm achieves only statistical security.[7] Later in this section we will describe a variant of the parallel intersperse that is perfectly secure but consumes more cost.

[7] The algorithm of Asharov et al. [7] may abort and fail with a negligible probability, and such negligible event reveals some information about the input (n, n_0) so that it is only statistically secure.

3.3.2 Warmup: A Sequential, Linear-Work Intersperse Algorithm

Asharov et al. [5] used the following method to construct a sequential **Intersperse** algorithm:

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

   $\mathcal{F}_{\text{SampleAux}}(n, n_0)$ – **Sample Auxiliary Array**
   - **Input:** Two numbers $n, n_0 \in \mathbb{N}$ such that $n_0 \leq n$.
   - **The functionality:** Sample an array Aux of n bits uniformly at random conditioned on having n_0 zeros and $n - n_0$ ones. Output Aux.

2. Next, we route elements 1-to-1 from $\mathbf{I}_0$ to zeros in Aux and 1-to-1 route elements from $\mathbf{I}_1$ to ones in Aux. This can be accomplished by running oblivious tight compaction circuit (Theorem 9) to pack all the 0s in Aux in the front. During the process, all swap gates remember their routing decisions. Now, we can run the oblivious tight compaction circuit in reverse and on the input array $\mathbf{I}_0||\mathbf{I}_1$. It is not hard to see that in the outcome, every 0 position in Aux would receive an element from $\mathbf{I}_0$ and every 1 position in Aux would receive an element from $\mathbf{I}_1$.

Asharov et al. [5, Claim 6.3] proved that the above algorithm indeed realizes the **Intersperse** abstraction as defined above. Moreover, they show how to implement the above idea obliviously in linear-time, resulting in the following theorem:

▶ **Theorem 10** (Sequential, linear-time **Intersperse** [5])**.** *There exists an algorithm that perfectly obliviously simulates $\mathcal{F}_{\text{shuffle}}$ for two randomly shuffled input arrays. Moreover, the algorithm completes in deterministic $O(\beta n)$ total work where n denotes the sum of the lengths of the two input arrays, and $\beta \geq 1$ denotes the number of memory words required to represent each element.*

3.3.3 Parallel Intersperse Algorithms

We need a parallel version of the **Intersperse** algorithm. In Asharov et al. [5]'s **Intersperse** construction, while the oblivious tight compaction building block can be replaced with a parallel realization of tight compaction (Theorem 9), unfortunately they adopt a highly sequential procedure for generating the Aux array. To get a parallel algorithm, it suffices to devise a parallel procedure for generating such an Aux array. More formally, we would like to devise an algorithm that obliviously simulates the functionality $\mathcal{F}_{\text{SampleAux}}(n, n_0)$.

A naïve algorithm with deterministic performance. A naïve algorithm is the following: simply write down exactly n_0 number of 0s and $n - n_0$ number of 1s, apply an oblivious random permutation to permute the array, and output the result. If we use Theorem 6 to instantiate this naïve algorithm, we obtain the following theorem:

▶ **Theorem 11** (Naïve parallel algorithm for sampling Aux)**.** *For any $n_0 \leq n$, there exists an algorithm that perfectly obliviously simulates $\mathcal{F}_{\text{SampleAux}}(n, n_0)$; moreover, for sampling an* Aux *array of length n, the algorithm completes in deterministic $O(n \log^2 n)$ total work and $O(\log^2 n)$ depth.*

This immediately gives rise to the following corollary for **Intersperse** due to the result of Asharov et al. [5] and parallel tight compaction (Theorem 9):

▶ **Corollary 12** (Naïve parallel **Intersperse**). *There exists an algorithm that perfectly obliviously simulates $\mathcal{F}_{\text{shuffle}}$ for two randomly shuffled input arrays. Moreover, the algorithm completes in deterministic $O(\beta n + n \log^2 n)$ total work and $O(\log^2 n)$ depth where n denotes the sum of the lengths of the two input arrays, and $\beta \geq 1$ denotes the number of memory words required to represent each element.*

A more efficient Las Vegas algorithm. To obliviously simulates the functionality $\mathcal{F}_{\text{SampleAux}}(n, n_0)$ with better performance, we use the Las Vegas version of oblivious random permutation, Theorem 7, which gives the following theorem:

▶ **Theorem 13** (Las Vegas parallel algorithm for sampling Aux). *For any $n_0 \leq n$, there exists a Las Vegas algorithm that perfectly obliviously simulates $\mathcal{F}_{\text{SampleAux}}(n, n_0)$. Except with probability $n^{-\Omega(\sqrt{n})}$, the algorithm completes in $O(n \log n)$ total work and $O(\log^2 n)$ depth. Furthermore, the above stated performance bounds also apply in expectation.*

Now due to the work of Asharov et al. [5] and parallel tight compaction (Theorem 9), we have the following corollary.

▶ **Corollary 14** (Parallel **Intersperse**). *Let $\beta \geq 1$ be the number of words used to represent an element. There is an* **Intersperse** *algorithm that is a perfectly oblivious simulation of $\mathcal{F}_{\text{shuffle}}$ on two randomly shuffled input arrays; moreover, except with $n^{-\Omega(\sqrt{n})}$ probability, the algorithm completes in $O(\beta n + n \log n)$ total work and $O(\log^2 n)$ depth. Moreover, the stated performance bounds also apply in expectation.*

We defer the formal construction and proof to the full version [14]

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. An O(N Log N) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.
2. Miklós Ajtai. Oblivious rams without cryptographic assumptions. In *STOC*, 2010.
3. Laurent Alonso and René Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 159(1):15–28, 1996.
4. Gilad Asharov, Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Locality-preserving oblivious ram. In *Eurocrypt*, 2019.
5. Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 403–432, Cham, 2020. Springer International Publishing.
6. Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Oblivious Parallel Tight Compaction. In *1st Conference on Information-Theoretic Cryptography (ITC 2020)*, volume 163 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:23, 2020.
7. Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Optimal oblivious parallel RAM. Cryptology ePrint Archive, Report 2020/1292, 2020. URL: `https://eprint.iacr.org/2020/1292`.
8. K. E. Batcher. Sorting Networks and Their Applications. In *AFIPS '68 (Spring)*, 1968.
9. 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.
10. T-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In *Asiacrypt*, 2017.

11 T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *Asiacrypt*, 2017.

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

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

14 T-H. Hubert Chan, Elaine Shi, Wei-Kai Lin, and Kartik Nayak. Perfectly oblivious (parallel) RAM revisited, and improved constructions. Cryptology ePrint Archive, Report 2020/604, 2020. URL: `https://eprint.iacr.org/2020/604`.

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

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

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

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

19 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.

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

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

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

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

24 S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

25 Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 406–425. Springer, 2011.

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

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

28 Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Not.*, 50(4):87–101, 2015.

29 Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376, 2015.

30 Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

31 Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *ACM Symposium on Theory of Computing (STOC)*, pages 294–303, 1997.

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

33 Enoch Peserico. Deterministic oblivious distribution (and tight compaction) in linear time. *CoRR*, abs/1807.06719, 2018. `arXiv:1807.06719`.

34 Michael Raskin and Mark Simkin. Perfectly secure oblivious ram with sublinear bandwidth overhead. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019*, pages 537–563, 2019.

35 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.

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

37 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.

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