Oblivious Hashing Revisited, and Applications to Asymptotically Efficient ORAM and OPRAM

T-H. Hubert Chan[1], Yue Guo[2], Wei-Kai Lin[2], and Elaine Shi[2]

[1]The University of Hong Kong
[2]Cornell University
hubert@cs.hku.hk, yg393@cornell.edu, {wklin, elaine}@cs.cornell.edu

September 21, 2017

Abstract

Oblivious RAM (ORAM) is a powerful cryptographic building block that allows a program to provably hide its access patterns to sensitive data. Since the original proposal of ORAM by Goldreich and Ostrovsky, numerous improvements have been made. To date, the best asymptotic overhead achievable for general block sizes is $O(\log^2 N/\log\log N)$, due to an elegant scheme by Kushilevitz et al., which in turn relies on the oblivious Cuckoo hashing scheme by Goodrich and Mitzenmacher.

In this paper, we make the following contributions: we first revisit the prior $O(\log^2 N/\log\log N)$-overhead ORAM result. We demonstrate the somewhat incompleteness of this prior result, due to the subtle incompleteness of a core building block, namely, Goodrich and Mitzenmacher's oblivious Cuckoo hashing scheme.

Even though we do show how to patch the prior result such that we can fully realize Goodrich and Mitzenmacher's elegant blueprint for oblivious Cuckoo hashing, it is clear that the extreme complexity of oblivious Cuckoo hashing has made understanding, implementation, and proofs difficult. We show that there is a conceptually simple $O(\log^2 N/\log\log N)$-overhead ORAM that dispenses with oblivious Cuckoo hashing entirely.

We show that such a conceptually simple scheme lends to further extensions. Specifically, we obtain the first $O(\log^2 N/\log\log N)$ Oblivious Parallel RAM (OPRAM) scheme[1], thus not only matching the performance of the best known sequential ORAM, but also achieving super-logarithmic improvements in comparison with known OPRAM schemes.

1 Introduction

Oblivious RAM [18,19,38], originally proposed in the seminal work by Goldreich and Ostrovsky [18, 19], is a powerful cryptographic primitive that provably obfuscates a program's access patterns to sensitive data. Since Goldreich and Ostrovsky's original work [18, 19], numerous subsequent works have proposed improved constructions, and demonstrated a variety of ORAM applications in both theoretical contexts (e.g., multiparty computation [22, 27], Garbled RAMs [17, 28]) as well as in secure hardware and software systems (e.g., secure processors [14, 15, 29, 37], and cloud outsourcing [21,36,39,40,45]). To hide access patterns, an ORAM scheme typically involves reading, writing, or shuffling multiple blocks for every data request. Suppose that on average, for each data

[1]In a companion paper, Chan and Shi obtain the same asymptotical result in the tree-based framework [8].

request, an ORAM scheme must read/write X blocks. In this paper, we refer to X as the overhead (or the total work blowup) of the ORAM scheme.

Goldreich and Ostrovsky [18, 19] showed that, roughly speaking, any "natural" ORAM scheme that treats each block as an "opaque ball" must necessarily suffer from at least logarithmic overhead. The recent Circuit ORAM [43] work demonstrated an almost matching upper bound for large enough blocks. Let N denote the total memory size. Circuit ORAM showed the existence of a *statistically* secure ORAM scheme that achieves $O(\alpha \log N)$ overhead for N^{ϵ}-bit blocks for any constant $\epsilon > 0$ and any super-constant function $\alpha = \omega(1)$ (where the ORAM's statistical failure probability is $N^{-\alpha}$). To date, the existence of an almost logarithmic ORAM scheme is only known for large blocks. For general block sizes, the state of affairs is different: the best known construction (asymptotically speaking) is a *computationally* secure scheme by Kushilevitz et al. [26], which achieves $O(\frac{\log^2 N}{\log \log N})$ overhead assuming block sizes[2] of $\Omega(\log N)$. We note that all known ORAM schemes assume that a memory block is at least large enough to store its own address, i.e., at least $\Omega(\log N)$ bits long. Therefore, henceforth in this paper, we use the term "general block size" to refer to a block size of $\Omega(\log N)$.

Although most practical ORAM implementations (in the contexts of secure multi-party computation, secure processors, and storage outsourcing) opted for tree-based ORAM constructions [38, 42, 43] due to tighter practical constants, we note that hierarchical ORAMs are nonetheless of much theoretical interest: for example, when the CPU has $O(\sqrt{N})$ private cache, hierarchical ORAMs can achieve $O(\log N)$ simulation overhead while a comparable result is not known in the tree-based framework. Recent works [3, 7] have also shown how hierarchical ORAMs can achieve asymptotically better locality and IO performance than known tree-based approaches.

Our contributions. In this paper, we make the following contributions:

- **Revisit $O(\log^2 N/\log\log N)$ ORAMs.** We revisit how to construct a *computationally* secure ORAM with $O(\frac{\log^2 N}{\log \log N})$ overhead for general block sizes. First, we show why earlier results along this front [21, 26] are somewhat incomplete due to the incompleteness of a core building block, oblivious Cuckoo hashing, that is proposed and described by Goodrich and Mitzenmacher [21]. Next, besides fixing and restating the earlier results regarding the existence of an $O(\log^2 N/\log\log N)$ ORAM, perhaps more compellingly, we show how to obtain an ORAM with the same asymptotic overhead, but in a conceptually much simpler manner, completely obviating the need to perform oblivious Cuckoo hashing [21] which is the center of complexity in the earlier result [26].

- **New results on efficient OPRAMs.** Building on our new ORAM scheme, we next present the first Oblivious Parallel RAM (OPRAM) construction that achieves $O(\frac{\log^2 N}{\log \log N})$ simulation overhead. To the best of our knowledge, our OPRAM scheme is the first one to asymptotically match the best known sequential ORAM scheme for general block sizes. Moreover, we achieve a super-logarithmic factor improvement over earlier works [5, 9] and over the concurrent work by Nayak et al. [32] (see further clarifications in Section 1.3).

We stress that our conceptual simplicity and modular approach can open the door for possible improvements. For example, our OPRAM results clearly demonstrate the benefits of having a conceptually clean hierarchical ORAM framework: had we tried to make (a corrected variant of) Kushilevitz et al. [26] into an OPRAM, it is not clear whether we could have obtained the same

[2] This $O(\frac{\log^2 N}{\log \log N})$ result for computational security was later matched in the tree-based ORAM framework [8, 13] although tree-based ORAMs were initially investigated for the case of statistical security.

performance. In particular, achieving $O(\log^2 N/\log\log N)$ worst-case simulation overhead requires deamortizing a parallel version of their oblivious cuckoo hash rebuilding algorithm, and moreover, work and depth have to be deamortized at the same time — and we are not aware of a way to do this especially due to the complexity of their algorithm.

1.1 Background on Oblivious Hashing and Hierarchical ORAMs

In this paper, we consider the hierarchical framework, originally proposed by Goldreich and Ostrovsky [18, 19], for constructing ORAM schemes. At a high level, this framework constructs an ORAM scheme by having exponentially growing levels of capacity $1, 2, 4, \ldots, N$ respectively, where each smaller level can be regarded as a "stash" for larger levels. Each level in the hierarchy is realized through a core abstraction henceforth called *oblivious hashing* in the remainder of this paper. Since oblivious hashing is the core abstraction we care about, we begin by explicitly formulating oblivious hashing as the following problem:

- *Functional abstraction.* Given an array containing n possibly dummy elements where each non-dummy element is a (key, value) pair, design an efficient algorithm that builds a hash table data structure, such that after the building phase, each element can be looked up by its key consuming a small amount of time and work. In this paper, we will assume that all non-dummy elements in the input array have distinct keys.

- *Obliviousness.* The memory access patterns of both the building and lookup phases do not leak any information (to a computationally bounded adversary) about the initial array or the sequence of lookup queries Q — as long as *all non-dummy queries in Q are distinct.* In particular, obliviousness must hold even when Q may contain queries for elements *not contained* in the array in which case the query should return the result $\perp$. The correct answer to a dummy query is also $\perp$ by convention.

Not surprisingly, the performance of a hierarchical ORAM crucially depends on the core building block, oblivious hashing. Here is the extent of our knowledge about oblivious hashing so far:

- Goldreich and Ostrovsky [18, 19] show an oblivious variant of normal balls-and-bins hashing that randomly throws n elements into n bins. They show that obliviously building a hash table containing n elements costs $O(\alpha n \log n \log \lambda)$ work, and each query costs $O(\alpha \log \lambda)$ work. If α is any super-constant function, we can attain a failure probability $\mathsf{negl}(\lambda)$. This leads to an $O(\alpha \log^3 N)$-overhead ORAM scheme, where N is the total memory size[3].

- Subsequently, Goodrich and Mitzenmacher [21] show that the Cuckoo hashing algorithm can be made oblivious, incurring $O(n \log n)$ total work for building a hash table containing n elements, and only $O(1)$ query cost (later we will argue why their oblivious hashing scheme is somewhat incomplete). This leads to an ORAM scheme with $O(\log^2 N)$-overhead.

- Kushilevitz et al. [26] in turn showed an elegant reparametrization trick atop the Goodrich and Mitzenmacher ORAM, thus improving the overhead to $O(\frac{\log^2 N}{\log\log N})$. Since Kushilevitz et al. [26] crucially rely on Goodrich and Mitzenmacher's oblivious Cuckoo hashing scheme, incompleteness of the hashing result in some sense carries over to their $O(\frac{\log^2 N}{\log\log N})$ overhead ORAM construction.

[3]Henceforth in this paper, we use n to denote the size of a hash table and λ to denote its security parameter. For our ORAM construction, we use N to denote both the logical memory size as well as the ORAM's security parameter. This distinction is necessary since the ORAM will employ hash tables of varying n.

1.2 Technical Roadmap

Revisit oblivious Cuckoo hashing. Goodrich and Mitzenmacher [21]'s blueprint for obliviously building a Cuckoo hash table is insightful and elegant. They express the task of Cuckoo hash table rebuilding as a MapReduce task (with certain nice properties), and they show that any such MapReduce algorithm has an efficient oblivious instantiation.

Fundamentally, their construction boils down using a sequence of oblivious sorts over arrays of (roughly) exponentially decreasing lengths. To achieve full privacy, it is necessary to hide the true lengths of these arrays during the course of the algorithm. Here, Goodrich and Mitzenmacher's scheme description and their proof appear inconsistent: their scheme seems to suggest padding each array to the maximum possible length for security — however, this would make their scheme $O(\log^3 N)$ overhead rather than the claimed $O(\log^2 N)$. On the other hand, their proof appears only to be applicable, if the algorithm reveals the true lengths of the arrays — however, as we argue in detail in Appendix A, the array lengths in the cuckoo hash rebuilding algorithm contain information about the size of each connected component in the cuckoo graph. Thus leaking array lengths can lead to an explicit attack that succeeds with non-negligible probability: at a high level, this attack tries to distinguish two request sequences, one repeatedly requesting the same block whereas the other requests disctinct blocks. The latter request sequence will cause the cuckoo graph in the access phase to resemble the cuckoo graph in the rebuild phase, whereas the former request sequence results in a fresh random cuckoo hash graph for the access phase (whose connected component sizes are different than the rebuild phase with relatively high probability).

As metioned earlier, the incompleteness of oblivious Cuckoo hashing also makes the existence proof of an $O(\log^2 N/\log\log N)$-overhead ORAM somewhat incomplete.

Is oblivious Cuckoo hashing necessary for efficient hierarchical ORAM? Goodrich and Mitzenmacher's oblivious Cuckoo hashing scheme is extremely complicated. Although we do show in our Appendix A that the incompleteness of Goodrich and Mitzemacher's construction and proofs can be patched, thus correctly and fully realizing the elegant blueprint they had in mind — the resulting scheme nonetheless suffers from large constant factors, and is unsuitable for practical implementation. Therefore, a natural question is, *can we build efficient hierarchical ORAMs without oblivious Cuckoo hashing?*

Our first insight is that perhaps oblivious Cuckoo hashing scheme is an overkill for constructing efficient hierarchical ORAMs after all. As initial evidence, we now present an almost trivial modification of the original Goldreich and Ostrovsky oblivious balls-and-bins hashing scheme such that we can achieve an $O(\alpha\log^2 N)$-overhead ORAM for any super-constant function α.

Recall that Goldreich and Ostrovsky [18, 19] perform hashing by hashing n elements into n bins, each of $O(\alpha\log\lambda)$ capacity, where λ is the security parameter. A simple observation is the following: instead of having n bins, we can have $\frac{n}{\alpha\log\lambda}$ bins — it is not hard to show that each bin's occupancy will still be upper bounded by $O(\alpha\log\lambda)$ except with $\mathsf{negl}(\lambda)$ probability. In this way, we reduce the size of the hash table by a $\log\lambda$ factor, and thus the hash table can be obliviously rebuilt in logarithmically less time. Plugging in this new hash table into Goldreich and Ostrovsky's ORAM construction [18,19], we immediately obtain an ORAM scheme with $O(\alpha\log^2 N)$ overhead.

This shows that through a very simple construction we can almost match Goodrich and Mitzenmacher's ORAM result [21]. This simple scheme does not quite get us to where we aimed to be, but we will next show that oblivious Cuckoo hashing is likewise an overkill for constructing $(\frac{\log^2 N}{\log\log N})$-overhead ORAMs.

Conceptually simple $(\frac{\log^2 N}{\log\log N})$-overhead ORAM. Recall that a hierarchical ORAM's overhead is impacted by two cost metrics of the underlying oblivious hashing scheme, i.e., the cost of building the hash-table, and the cost of each lookup query. Goodrich and Mitzenmacher's oblivious Cuckoo hashing scheme [21] minimizes the lookup cost to $O(1)$, but this complicates the building of the hash-table.

Our key insight is that in all known hashing-based hierarchical ORAM constructions [18, 19, 21, 26], the resulting ORAM's cost is dominated by the hash-table rebuilding phase, and thus it may be okay if the underlying hashing scheme is more expensive in lookup. More specifically, to obtain an $O(\frac{\log^2 N}{\log\log N})$ ORAM, we would like to apply Kushilevitz et al. [26]'s reparametrized version of the hierarchical ORAM. Kushilevitz et al. [26] showed that their reparametrization technique works when applied over an oblivious Cuckoo hashing scheme. We observe that in fact, Kushilevitz et al. [26]'s reparametrization technique is applicable for a much broader parameter range, and concretely for any oblivious hashing scheme with the following characteristics:

- It takes $O(n \log n)$ total work to build a hash table of n elements — in other words, the per-element building cost is $O(\log n)$.
- The lookup cost is asymptotically smaller than the per-element building cost — specifically, $O(\log^\epsilon \lambda)$ lookup cost suffices where $\epsilon \in (0.5, 1)$ is a suitable constant.

This key observation allows us to relax the lookup time on the underlying oblivious hashing scheme. We thus propose a suitable oblivious hashing scheme that is conceptually simple. More specifically, our starting point is a (variant of a) two-tier hashing scheme first described in the elegant work by Adler et al. [1]. In a two-tier hashing scheme, there are two hash tables denoted H_1 and H_2 respectively, each with $\frac{n}{\log^\epsilon \lambda}$ bins of $O(\log^\epsilon \lambda)$ capacity, where $\epsilon \in (0.5, 1)$ is a suitable constant. To hash n elements (non-obliviously), we first throw each element into a random bin in H_1. For all the elements that overflow its bin capacity, we throw them again into the second hash table H_2. Stochastic bounds show that the second hash table H_2 does not overflow except with $\mathsf{negl}(\lambda)$ probability. Clearly, the lookup cost is $O(\log^\epsilon \lambda)$; and we will show that the hash table building algorithm can be made oblivious through $O(1)$ number of oblivious sorts.

New results on oblivious parallel RAM. The conceptual simplicity of our ORAM scheme not only makes it easier to understand and implement, but also lends to further extensions. In particular, we construct a computationally secure OPRAM scheme that has $O(\log^2 N/\log\log N)$ overhead — to the best of our knowledge, this is the first OPRAM scheme that matches the best known sequential ORAM in performance for general block sizes. Concretely, the hierarchical lookup phase can be parallelized using the standard conflict resolution (proposed by Boyle et al. [5]) as this phase is read-only. In the rebuild phase, our two-tier oblivious hashing takes only $O(1)$ number of oblivious sort and linear scan that marks excess elements, which can be parallelized with known algorithms, i.e. range prefix sum.

As mentioned earlier, our modular approach and conceptual simplicity turned out to be a crucial reason why we could turn our ORAM scheme into an OPRAM — it is not clear whether (a corrected version of) Kushilevitz et al. [26] is amenable to the same kind of transformation achieving the same overhead due to complications in deamortizing their cuckoo hash rebuilding algorithm. Thus we argue that our conceptually simple framework can potentially lend to other possible applications and improvements.

1.3 Related Work

ORAMs. ORAM was first proposed in a seminal work by Goldreich and Ostrovsky [18, 19] who showed a computationally secure scheme with $O(\alpha \log^3 N)$ overhead for general block sizes and for any super-constant function $\alpha = \omega(1)$. Subsequent works improve the hierarchical ORAM [21, 26] and show that $O(\frac{\log^2 N}{\log \log N})$ overhead can be attained under computational security — our paper points out several subtleties and the incompleteness of the prior results; additionally, we show that it is possible to obtain such an $O(\frac{\log^2 N}{\log \log N})$ overhead in a conceptually much simpler manner.

Besides the hierarchical framework, Shi et al. [38] propose a tree-based paradigm for constructing ORAMs. Numerous subsequent works [10, 42, 43] improved tree-based constructions. With the exception of a few works [13], the tree-based framework was primarily considered for the construction of *statistically secure* ORAMs. The performance of tree-based ORAMs depend on the block size, since with a larger block size we can reduce the number of recursion levels in these constructions. The recent Circuit ORAM work [43] shows that under block sizes as large as N^ϵ for any arbitrarily small constant ϵ, we can achieve $\alpha \log N$ bandwidth overhead for an arbitrary super-constant function $\alpha = \omega(1)$ — this also shows the (near) tightness of the Goldreich-Ostrovsky lower bound [18, 19] showing that any ORAM scheme must necessarily incur logarithmic overhead. Note that under block sizes of at least $\log^{1+\epsilon} N$ for an arbitrarily small constant ϵ, Circuit ORAM [43] can also attain $O(\frac{\log^2 N}{\log \log N})$ overhead and it additionally achieves statistical security rather than computational.

OPRAMs. Since modern computing architectures such as cloud platforms and multi-core architectures exhibit a high degree of parallelism, it makes sense to consider the parallel counterpart of ORAM. Oblivious Parallel ORAM (OPRAM) was first proposed by Boyle et al. [5], who showed a construction with $O(\alpha \log^4 N)$ overhead for any super-constant function α. Boyle et al.'s result was later improved by Chen et al. [9], who showed how to achieve $O(\alpha \log^3 N)$ overhead with poly-logarithmic CPU private cache — their result also easily implies an $O(\alpha \log^3 N \log \log N)$ overhead OPRAM with $O(1)$ CPU private cache, the setting that we focus on in this paper for generality.

A concurrent and independent manuscript by Nayak et al. [32] further improves the CPU-memory communication by extending Chen et al.'s OPRAM [9]. However, their scheme still requires $O(\alpha \log^3 N \log \log N)$ CPU-CPU communication which was the dominant part of the overhead in Chen et al. [9]. Therefore, under a general notion of overhead that includes both CPU-CPU communication and CPU-memory communication, Nayak et al.'s scheme still has the same asymptotic overhead[4] as Chen et al. [9] which is more than a logarithmic factor more expensive in comparison with our new OPRAM construction.

In a companion paper, Chan et al. [8] showed how to obtain statistically secure and computationally secure OPRAMs in the tree-based ORAM framework. Specifically, they showed that for general block sizes, we can achieve statistically secure OPRAM with $O(\log^2 N)$ simulation overhead and computationally secure OPRAM with $O(\log^2 N / \log \log N)$ simulation overhead. For the computationally secure setting, Chan et al. [8] achieves the same asymptotical overhead as this paper, but the two constructions follow different paradigms so we believe that they are both of value. In another recent work, Chan et al. [6] proposed a new notion of depth for OPRAMs where the OPRAM is allowed to have more CPUs than the original PRAM to further parallelize the computation. In this paper, an OPRAM's simulation overhead is defined as its runtime blowup assuming that the OPRAM consumes the same number of CPUs as the PRAM.

[4]The title of their paper [32] suggests $O(\log^2 N)$ overhead, since they did not account for the cost of CPU-CPU communication when describing the overhead.

Non-oblivious techniques for hashing. Many hashing schemes [4,11,16,24,30] were considered in the (parallel) algorithms literature. Unfortunately, most of them are *not* good candidates for constructing efficient ORAM and OPRAM schemes since there is no known efficient and oblivious counterpart for the algorithm. Below we review some representative works in the hashing literature.

Multiple choice hashing [31], commonly referred to as "the power of choice 2" is a hashing scheme that reduces collision by having each element inspect 2 (or in general, $d = O(1)$) random bins, and pick the bin with the smallest load. An ingenious analysis by Mitzenmacher shows that each bin's load cannot exceed $c \log \log n$ with high probability. Unfortunately, the power of choice 2 is not a great candidate for constructing ORAM and OPRAMs, since the algorithm for building the hash table heavily relies on dynamic memory accesses and we know of no straightforward way implement it as an efficient oblivious algorithm.

Perfect hashing or static hashing [11, 16] is another approach for hashing such that lookups can be performed in constant time. The idea of perfect hashing is to select a hash function that satisfies nice properties, e.g., one that reduces collisions. Perfect hashing and similar approaches are not natural candidates for constructing oblivious hashing schemes, since the selection of the hash function biases the distribution: for example, suppose a hash function with fewer collisions is preferred, then if one sees a visit to a hash bin looking for a real item, then another request for a real item will less likely hit the same hash bin. Further, although some static hashing schemes [16] also adopt a two-tier hash structure, their constructions are different in nature from ours despite the name collision.

The parallel algorithms community have also proposed several elegant and highly non-trivial parallel hashing algorithms [4, 24, 30]. Amazingly, these ingenious works have shown that it takes only $O(n)$ work and $O(\log^* n)$ time to preprocess n elements, such that lookup queries can be supported in constant time. Unfortunately, to the best of our knowledge, known parallel hashing algorithms are fundamentally non-oblivious as well in a similar spirit as why perfect hashing [16] is non-oblivious.

2 Definitions and Building Blocks

2.1 Parallel Random Access Machines

We define a Parallel Random Access Machine (PRAM) and an Oblivious Parallel Random Access Machine (OPRAM) in a similar fashion as Boyle et al. [5] as well as Chan and Shi [8]. Some of the definitions in this section are borrowed verbatim from Boyle et al. [5]. or Chan and Shi [8].

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.

Parallel Random Access Machine (PRAM). 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 D 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 $I_i^{(t)} := (\mathtt{read}, \mathsf{addr})$, or $I_i^{(t)} := (\mathtt{write}, \mathsf{addr}, \mathsf{data})$

where the operation is performed on the memory block with address addr and the block content $\mathsf{data} \in \{0,1\}^D \cup \{\perp\}$.

If $I_i^{(t)} = (\mathtt{read}, \mathsf{addr})$ then the CPU i should receive the contents of $\mathsf{mem}[\mathsf{addr}]$ at the beginning of time step t. Else if $I_i^{(t)} = (\mathtt{write}, \mathsf{addr}, \mathsf{data})$, 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 be well-defined. In this paper, we assume the following:

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

We note that a CRCW-PRAM with a parametrizable conflict resolution rule is among the most powerful CRCW-PRAM model, whereas CREW is a much weaker model. Our results are stronger if we allow the underlying PRAM to be more powerful but 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 [23].

CPU-to-CPU communication. In the remainder of the paper, we sometimes describe our algorithms using CPU-to-CPU communication. For our OPRAM algorithm to be oblivious, the inter-CPU communication pattern must be oblivious too. We stress that such inter-CPU communication can be emulated using shared memory reads and writes. Therefore, when we express our performance metrics, we assume that all inter-CPU communication is implemented with shared memory reads and writes. In this sense, our performance metrics already account for any inter-CPU communication, and there is no need to have separate metrics that characterize inter-CPU communication. In contrast, some earlier works [9] adopt separate metrics for inter-CPU communication.

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

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

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 P_t denotes the set of CPUs to be activated in each time step $t \in [T]$, where $m_t := |P_t|$.

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

2.2 Oblivious Parallel Random-Access Machines

Randomized PRAM. A *randomized PRAM* is a special PRAM where the CPUs are allowed to generate private random numbers. For simplicity, we assume that a randomized PRAM has a priori known, deterministic runtime, 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 of inputs $(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$, we define the notation $\mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$ as follows:

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

Oblivious PRAM (OPRAM) Scheme. To define oblivious PRAM (OPRAM) scheme, we will consider stateful algorithms. A stateful algorithm can be activated multiple times over time, each time receiving some input and returning some output; moreover, the algorithm stores persistent state in between multiple activations. Oblivious PRAM scheme is a stateful algorithm that obliviously simulates an ideal logical memory that always returns the last value written when several addresses are requested. More formally, recall that a PRAM algorithm sends the instruction $\vec{I}^{(t)} := (I_i^{(t)} : i \in [m_t])$ at each time step t, where m_t denotes the number of CPUs and CPU i's instruction is of the form $I_i^{(t)} := (\mathtt{read}, \mathsf{addr})$, or $I_i^{(t)} := (\mathtt{write}, \mathsf{addr}, \mathsf{data})$. Let $\mathcal{F}_{\text{mem}}$ denote the ideal logical memory such that on receiving the request instructions $\vec{I}^{(t)}$, for each $i \in [m_t]$, $\mathcal{F}_{\text{mem}}$ outputs the last value (for all step $t' < t$) written to addr in its state; or if nothing has been written to addr, it outputs 0; additionally, if $I_i^{(t)} = (\mathtt{write}, \mathsf{addr}, \mathsf{data})$, $\mathcal{F}_{\text{mem}}$ writes data to addr in its state. We define an adaptively secure, composable notion for OPRAM scheme below.

Definition 1 (Adaptively secure OPRAM scheme)**.** We say that a stateful algorithm OPRAM is an oblivious PRAM scheme iff there exists a p.p.t. simulator Sim, such that for any non-uniform p.p.t. adversary $\mathcal{A}$, $\mathcal{A}$'s view in the following two experiments, $\mathsf{Expt}_{\mathcal{A}}^{\text{real},\mathsf{OPRAM}}$ and $\mathsf{Expt}_{\mathcal{A},\mathsf{Sim}}^{\text{ideal},\mathcal{F}_{\text{mem}}}$ are computationally indistinguishable:

$\mathsf{Expt}_{\mathcal{A}}^{\text{real},\mathsf{OPRAM}}(1^\lambda)$:	$\mathsf{Expt}_{\mathcal{A},\mathsf{Sim}}^{\text{ideal},\mathcal{F}_{\text{mem}}}(1^\lambda)$:
$\mathsf{out}_0 = X_0 = \perp$	$\mathsf{out}_0 = X_0 = \perp$
For $r = 1, 2, \ldots \mathsf{poly}(\lambda)$:	For $r = 1, 2, \ldots \mathsf{poly}(\lambda)$:
$\vec{I}^{(r)} \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_{r-1}, X_{r-1})$	$\vec{I}^{(r)} \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_{r-1}, X_{r-1})$
$\mathsf{out}_r \leftarrow \mathsf{OPRAM}(\vec{I}^{(r)})$,	$\mathsf{out}_r \leftarrow \mathcal{F}_{\text{mem}}(\vec{I}^{(r)})$,
$X_r := \mathsf{Addresses}[\mathsf{OPRAM}](\vec{I}^{(r)})$	$X_r \leftarrow \mathsf{Sim}(1^\lambda, N, r)$

Note that here we handle correctness and obliviousness in a single definition. Also note that the output out_r of ideal logical memory is a vector that responds the instruction $\vec{I}^{(r)}$, but X_r denotes the addresses that OPRAM incurs to fulfill the instruction $\vec{I}^{(r)}$, which can involve several CPUs and take several time steps.

Oblivious simulation and simulation overhead. In the above definition, an OPRAM scheme simulates the ideal logic parallel memory. For convenience, we often adopt two intermediate metrics in our descriptions, namely, *total work blowup* and *parallel runtime blowup*. We say that an OPRAM scheme has a total work blowup of x and a parallel runtime blowup of y, iff for every step t in which the instruction $\vec{I}^{(t)}$ requested by m_t CPUs, the OPRAM can complete this step with $x \cdot m_t$ total work and in y parallel steps — if the OPRAM is allowed to consume any number of CPUs (possibly greater than m_t).

Fact 1. *If there exists an* OPRAM_0 *scheme with x total work blowup and y parallel runtime blowup such that $x \geq y$, then there exists an* OPRAM_1 *scheme that has $O(x)$ simulation overhead when consuming the same number of CPUs as the original PRAM that sends instructions at every step.*

Proof. Let OPRAM_0 the OPRAM that has x total work blowup and y parallel runtime blowup, where OPRAM_0 may consume as many CPUs as it wants. We now construct an OPRAM scheme denoted OPRAM_1 that consumes the same number of CPUs as the original PRAM. Effectively OPRAM_1 will simulate OPRAM_0 with possibly fewer CPUs than OPRAM_0 for any PRAM step. Suppose that for a time step t, the PRAM consumes m_t CPUs: OPRAM_1 simulates the t-th PRAM step as follows:

- If OPRAM_0 consumes at most m_t CPUs for simulating the t-th PRAM step, then OPRAM_1 does the same as what OPRAM_0 would have done for simulating the t-th PRAM step. Clearly, OPRAM_1 can simulate the t-th PRAM step in $O(y)$ number of parallel steps where $y \leq x$.

- The non-trivial case is that OPRAM_0 uses more than m_t CPUs. Observe that one parallel step using m CPUs (where $m > m_t$) has total work m; when simulated by m_t CPUs, this total work can be finished in $O(m/m_t)$ parallel steps. Denote W as the portion of total work performed by OPRAM_0 with more than m_t CPUs. By the above argument, this portion can be simulated with m_t CPUs in $O(\frac{W}{m_t})$ parallel steps. Hence, the total number of parallel steps by OPRAM_1 (with only m_t CPUs) is at most $y + O(\frac{W}{m_t}) \leq y + x = O(x)$, where first inequality follows by the definition of x.

□

2.3 Oblivious Hashing Scheme

Without loss of generality, we define only the parallel version, since the sequential version can be thought of the parallel version subject to executing on a single CPU.

An oblivious (parallel) hashing scheme is a stateful (parallel) algorithm that obliviously simulates an ideal hashing functionality denoted $\mathcal{F}_{\mathrm{ht}}$, consisting of the following activation points:

- $\perp \leftarrow \mathcal{F}_{\mathrm{ht}}(\mathsf{Build}, S)$: stores the input set $S = \{(k_i, v_i) \mid \mathsf{dummy}\}_{i \in [n]}$, where each element is either a dummy denoted dummy or a (key, value) pair denoted (k_i, v_i). For an input set S to be *valid*, we require that any two non-dummy elements in S must have distinct keys.

- $v \leftarrow \mathcal{F}_{\mathrm{ht}}(\mathsf{Lookup}, k)$: given a (possibly dummy) query k, outputs a value v. Let set S be the set stored in the last Build request (or empty set if there is no such set). Given S and query k, v is chosen in a way that resepects the following:

  - If $k = \mathsf{dummy}$ (i.e., if k is a dummy query) or if $k \notin S$, then $v = \perp$.
  - Else, it must hold that $(k, v) \in S$.

When we consider a parallel oblivious hashing scheme, we assume that it is executed on a Concurrent Read, Exclusive Write PRAM.

A legitimate sequence of requests is denoted as $\vec{I} = (I_1, \ldots, I_{\mathsf{poly}(\lambda)})$, where the first request I_1 must be of the form $(\mathsf{Build}, _)$, and every other request I_i for $i > 1$ must be of the form $(\mathsf{Lookup}, _)$. We say $\vec{I}$ is *non-recurrent* if all non-dummy Lookup queries in $\vec{I}$ ask for distinct keys.

Definition 2 (Adaptively secure oblivious hashing scheme)**.** We say that HT is an oblivious (parallel) hashing scheme if it obliviously simulates an ideal hashing functionality $\mathcal{F}_{\rm ht}$ in the following sense: there exists a p.p.t. simulator Sim, such that for any non-uniform p.p.t. adversary $\mathcal{A}$, $\mathcal{A}$'s view in the following two experiments, $\mathsf{Expt}_{\mathcal{A}}^{\rm real,\mathsf{HT}}$ and $\mathsf{Expt}_{\mathcal{A},\mathsf{Sim}}^{\rm ideal,\mathcal{F}_{\rm ht}}$ are computationally indistinguishable as long as $\mathcal{A}$ produces non-recurrent (legitimate) request sequences in both experiments with probability 1 (below n denotes the size of the set contained in the Build request, i.e., the first request in a legitimate sequence):

$\mathsf{Expt}_{\mathcal{A}}^{\rm real,\mathsf{HT}}(1^\lambda)$:	$\mathsf{Expt}_{\mathcal{A},\mathsf{Sim}}^{\rm ideal,\mathcal{F}_{\rm ht}}(1^\lambda)$:
$\mathsf{out}_0 = X_0 = \perp$	$\mathsf{out}_0 = X_0 = \perp$
For $r = 1, 2, \ldots \mathsf{poly}(\lambda)$:	For $r = 1, 2, \ldots \mathsf{poly}(\lambda)$:
$I_r \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_{r-1}, X_{r-1})$	$I_r \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_{r-1}, X_{r-1})$
$\mathsf{out}_r \leftarrow \mathsf{HT}(1^\lambda, I_r)$,	$\mathsf{out}_r \leftarrow \mathcal{F}_{\rm ht}(I_r)$,
$X_r := \mathsf{Addresses}[\mathsf{HT}](I_r)$	$X_r \leftarrow \mathsf{Sim}(1^\lambda, n, r)$

Notations. For convenience, for our real-world hashing algorithms, we will write $\mathsf{T} \leftarrow \mathsf{Build}(1^\lambda, S)$ in place of $\mathsf{HT}(1^\lambda, \mathsf{Build}, S)$ where T is the data structure created by the real-world Build algorithm upon receiving S; similarly we will write $\mathsf{Lookup}(1^\lambda, \mathsf{T}, k)$ in place of $\mathsf{HT}(1^\lambda, \mathsf{Lookup}, k)$.

Definition 3 ($(W_{\rm build}, T_{\rm build}, W_{\rm lookup}, T_{\rm lookup})$-parallel oblivious hashing scheme)**.** Let $W_{\rm build}(\cdot,\cdot)$, $W_{\rm lookup}(\cdot,\cdot)$, $T_{\rm build}(\cdot,\cdot)$, and $T_{\rm lookup}(\cdot,\cdot)$ be functions in n and λ. We say that (Build, Lookup) is a $(W_{\rm build}, T_{\rm build}, W_{\rm lookup}, T_{\rm lookup})$*-parallel oblivious hashing scheme*, iff (Build, Lookup) satisfies adaptive security as defined above; and moreover, the scheme achieves the following performance:

- Building a hash table with n elements takes $n \cdot W_{\rm build}(n, \lambda)$ total work and $T_{\rm build}(n, \lambda)$ time with all but $\mathsf{negl}(\lambda)$ probability. Note that $W_{\rm build}(n, \lambda)$ is the *per-element* amount of work required for preprocessing.
- A lookup query takes $W_{\rm lookup}(n, \lambda)$ total work and $T_{\rm lookup}(n, \lambda)$ time.

As a special case, we say that (Build, Lookup) is a $(W_{\rm build}, W_{\rm lookup})$*-oblivious hashing scheme*, if it is a $(W_{\rm build}, _, W_{\rm lookup}, _)$-parallel oblivious hashing scheme for any choice of the wildcard field "_" — in other words, in the sequential case, we do not care about the scheme's parallel runtime, and the scheme's total work is equivalent to the runtime when running on a single CPU.

[**Read-only lookup assumption.**] When used in ORAM, observe that elements are inserted in a hash table in a batch only in the Build algorithm. Moreover, we will assume that the Lookup algorithm is read-only, i.e., it does not update the hash table data structure T, and no state is carried across between multiple invocations of Lookup.

A note on the security parameter. Since later in our application, we will need to apply oblivious hashing to different choices of n (including possibly small choices of n), throughout the description of the oblivious hashing scheme, we distinguish the security parameter denoted λ and the size of the set to be hashed denoted n.

2.4 Building Blocks

Duplicate suppression. Informally, duplicate suppression is the following building block: given an input array X consisting of (key, value) pairs and possibly dummy elements where each key can have multiple occurrences, the algorithm outputs a duplicate-suppressed array where only one occurrence of each key is preserved, and a preference function $\mathsf{priority}$ is used to choose which one.

More formally, $\mathsf{SuppressDuplicate}(X, n', \mathsf{priority})$ is a parallel algorithm that takes in the following inputs:

- an array X where each element is either of the form (k, v) or a dummy denoted $\perp$,
- a correct upper bound n' on the number of real elements in X which also serves as the target output length, and
- a priority function $\mathsf{priority}$ which defines a total ordering on all elements with the same key.

$\mathsf{SuppressDuplicate}(X, n', \mathsf{priority})$ outputs an array Y of length n', such that

- No two elements in Y have the same key k except for dummy elements;
- All non-dummy elements in Y come from X, i.e., for any $(k, v) \in Y$, it holds that $(k, v) \in X$; and
- If multiple elements in X have the same k, the most preferred element is preserved in Y. More formally, for every non-dummy element $(k, v) \in X$, there exists a $(k, v') \in Y$ such that either $v' = v$ or (k, v') is preferred over (k, v) according to the priority function $\mathsf{priority}$.

Earlier works have [5, 18, 19] proposed an algorithm that relies on oblivious sorting to achieve duplicate suppression in $O(n \log n)$ work and $O(\log n)$ parallel runtime where $n := |X|$.

Oblivious select. $\mathsf{Select}(X, k, \mathsf{priority})$ takes in an array X where each element is either of the form (k, v) or a dummy denoted $\perp$, a query k, and a priority function $\mathsf{priority}$ which defines a total ordering on all elements with the same key; and outputs a value v such that $(k, v) \in X$ and moreover there exists no $(k, v') \in X$ such that v' is preferred over v for the key k by the priority function $\mathsf{priority}$.

Oblivious select can be accomplished using a simple tree-based algorithm [8] in $O(\log n)$ parallel runtime and $O(n)$ total work where $n = |X|$.

Oblivious multicast. Oblivious multicast is the following building block. Given the following inputs:

- a source array $X := \{(k_i, v_i) \mid \mathsf{dummy}\}_{i \in [n]}$ where each element is either of the form (k, v) or a dummy denoted dummy, and further all real elements must have a distinct k; and
- a destination array $Y := \{k'_i\}_{i \in [n]}$ where each element is a query k' (possibly having duplicates).

the oblivious multicast algorithm outputs an array $\mathsf{ans} := \{v_i\}_{i \in [n]}$ such that if $k'_i \notin X$ then $v_i := \perp$; else it must hold that $(k'_i, v_i) \in X$.

Boyle et al. [5] propose an algorithm based on $O(1)$ oblivious sorts that achieves oblivious multicast in $O(\log n)$ parallel runtime and $O(n \log n)$ total work.

Range prefix sum. We will rely on a parallel range prefix sum algorithm which offers the following abstraction: given an input array $X = (x_1, \ldots, x_n)$ of length n where each element of X is of the form $x_i := (k_i, v_i)$, output an array $Y = (y_1, \ldots, y_n)$ where each y_i is defined as follows:

- Let $i' \leq i$ be the smallest index such that $k_{i'} = k_{i'+1} = \ldots = k_i$;
- $y_i := \sum_{j=i'}^{i} v_j$.

In the GraphSC work, Nayak et al. [33] provide an oblivious algorithm that computes the range prefix sum in $O(\log n)$ parallel runtime and $O(n \log n)$ total work — in particular, their paper [33] defines a building block called "longest prefix sum" which is a slight variation of the range prefix sum abstraction we need. It is easy to see that Nayak et al.'s algorithm for longest prefix sum can be modified in a straightforward manner to compute our notion of range prefix sum.

3 Oblivious Two-Tier Hashing Scheme

In this section, we present a simple oblivious two-tier hashing scheme. Before we describe our scheme, we make a couple important remarks that the reader should keep in mind:

- Note that our security definition implies that the adversary can only observe the memory access patterns, and we require simulatability of the memory access patterns. Therefore our scheme description does not explicitly encrypt data. When actually deploying an ORAM scheme, all data must be encrypted if the adversary can also observe the contents of memory.

- In our oblivious hashing scheme, we use λ to denote the security parameter, and use n to denote the hash table's size. Our ORAM application will employ hash tables of varying sizes, so n can be small. Observe that an instance of hash table building can fail with $\mathsf{negl}(\lambda)$ probability; when this happens in the context of ORAM, the hash table building is restarted. This ensures that the ORAM is always correct, and the security parameter is related to the running time of the ORAM.

- For small values of n, we need special treatment to obtain $\mathsf{negl}(\lambda)$ security failure probability — specifically, we simply employ normal balls-and-bins hashing for small values of n. Instead of having the ORAM algorithm deal with this issue, we wrap this part inside the oblivious hashing scheme, i.e., the oblivious hashing scheme will automatically decide whether to employ normal hashing or two-tier hashing depending on n and λ.

  This modular approach makes our ORAM and OPRAM algorithms conceptually simple and crystallizes the security argument as well.

The goal of this section is to give an oblivious hashing scheme with the following guarantee.

Theorem 1 (Parallel oblivious hashing). *For any constant $\epsilon > 0.5$, for any $\alpha(\lambda) := \omega(1)$, there exists a $(W_{\text{build}}, T_{\text{build}}, W_{\text{lookup}}, T_{\text{lookup}})$-parallel oblivious hashing scheme where*

$$W_{\text{build}} = O(\log n), \qquad T_{\text{build}} = O(\log n),$$

$$W_{\text{lookup}} = \begin{cases} O(\alpha \log \lambda) & \text{if } n < e^{3 \log^{\epsilon} \lambda} \\ O(\log^{\epsilon} \lambda) & \text{if } n \geq e^{3 \log^{\epsilon} \lambda} \end{cases}, \quad T_{\text{lookup}} = O(\log \log \lambda)$$

3.1 Construction: Non-Oblivious and Sequential Version

For simplicity, we first present a non-oblivious and sequential version of the hashing algorithm, and we can use this version of the algorithm for the purpose of our stochastic analysis. Later in Section 3.2, we will show how to make the algorithm both oblivious and parallel. Henceforth, we fix some $\epsilon \in (0.5, 1)$.

Case 1: $n < e^{3\log^{\epsilon}\lambda}$**.** When n is sufficiently small relative to the security parameter λ, we simply apply normal hashing (i.e., balls and bins) in the following manner. Let each bin's capacity $Z(\lambda) = \alpha \log \lambda$, for any $\alpha = \omega(1)$ superconstant function in λ.

For building a hash table, first, generate a secret PRF key denoted $\mathsf{sk} \xleftarrow{\$} \{0,1\}^{\lambda}$. Then, store the n elements in $B := \lceil 5n/Z \rceil$ bins each of capacity Z, where each element $(k, _)$ is assigned to a pseudorandom bin computed as follows:

$$\text{bin number} := \mathsf{PRF}_{\mathsf{sk}}(k).$$

Due to a simple application of the Chernoff bound, the probability that any bin overflows is negligible in λ as long as Z is superlogarithmic in λ.

To look up an element with the key k, compute the bin number as above and read the entire bin.

Case 2: $n \geq e^{3\log^{\epsilon}\lambda}$**.** This is the more interesting case, and we describe our two-tier hashing algorithm below.

- **Parameters and data structure.** Suppose that our memory is organized into two hash tables named H_1 and H_2 respectively, where each hash table has $B := \lceil \frac{n}{\log^{\epsilon}\lambda} \rceil$ bins, and each bin can store at most $Z := 5\log^{\epsilon}\lambda$ blocks.
- $\mathsf{Build}(1^{\lambda}, \{(k_i, v_i) \mid \mathsf{dummy}\}_{i \in [n]})$:
  a) Generate a PRF key $\mathsf{sk} \xleftarrow{\$} \{0,1\}^{\lambda}$.
  b) For each element $(k_i, v_i) \in S$, try to place the element into the bin numbered $\mathsf{PRF}_{\mathsf{sk}}(1||k_i)$ in the first-tier hash table H_1. In case the bin is full, instead place the element in the overflow pile henceforth denoted Buf.
  c) For each element (k, v) in the overflow pile Buf, place the element into the bin numbered $\mathsf{PRF}_{\mathsf{sk}}(2||k)$ in the second-tier hash table H_2.
  d) Output $\mathsf{T} := (\mathsf{H}_1, \mathsf{H}_2, \mathsf{sk})$.
- $\mathsf{Lookup}(\mathsf{T}, k)$: Parse $\mathsf{T} := (\mathsf{H}_1, \mathsf{H}_2, \mathsf{sk})$ and perform the following.
  a) If $k = \perp$, i.e., this is a dummy query, return $\perp$.
  b) Let $i_1 := \mathsf{PRF}_{\mathsf{sk}}(1||k)$. If an element of the form (k, v) is found in $\mathsf{H}_1[i_1]$, return v. Else, let $i_2 := \mathsf{PRF}_{\mathsf{sk}}(2||k)$, look for an element of the form (k, v) in $\mathsf{H}_2[i_2]$ and return v if found.
  c) If still not found, return $\perp$.

Overflow event. If in the above algorithm, an element happens to choose a bin in the second-tier hash table H_2 that is full, we say that a bad event called overflow has happened. When a hash building is called in the execution of an ORAM, recall that if an overflow occurs, we simply discard all work thus far and restart the build algorithm from the beginning.

In Section 3.4, we will prove that indeed, overflow events occur with negligible probability. Therefore, henceforth in our ORAM presentation, *we will simply pretend that* overflow *events never happen during hash table building.*

Remark 1. Since the oblivious hashing scheme is assumed to retry from scratch upon overflows, we guarantee perfect correctness and computational security failure (due to the use of a PRF). Similarly, our resulting ORAM and OPRAM schemes will also have perfect correctness and computational security. Obviously, the algorithms may execute longer if overflows and retries take place — henceforth in the paper, whenever we say that an algorithm's total work or runtime is bounded by x, we mean that it is bounded by x except with negligible probability over the randomized execution.

3.2 Construction: Making it Oblivious

Oblivious Building. To make the building phase oblivious, it suffices to have the following Placement building block.

Let B denote the number of bins, let Z denote each bin's capacity, and let R denote the maximum capacity of the overflow pile. Placement is the following building block. Given an array $\mathsf{Arr} = \{(\mathsf{elem}_i, \mathsf{pos}_i) \mid \mathsf{dummy}\}_{i\in[n]}$ containing n possibly dummy elements, where each non-dummy element elem_i is tagged with a pseudo-random bin number $\mathsf{pos}_i \in [B]$, output B arrays $\{\mathsf{Bin}_i\}_{i\in[B]}$ each of size exactly Z and an overflow pile denoted Buf of size exactly R. The placement algorithm must output a valid assignment if one exists. Otherwise if no valid assignment exists, the algorithm should abort outputting hash-failure.

We say that an assignment is valid if the following constraints are respected:

i) Every non-dummy $(\mathsf{elem}_i, \mathsf{pos}_i) \in \mathsf{Arr}$ exists either in some bin or in the overflow pile Buf.

ii) For every Bin_i, every non-dummy element in Bin_i is of the form $(_, i)$. In other words, non-dummy elements can only reside in their targeted bin or the overflow pile Buf.

iii) For every Bin_i, if there exists a dummy element in Bin_i, then no element of the form $(_, i)$ appears in Buf. In other words, no elements from each bin should overflow to Buf unless the bin is full.

[*Special case*]. A special case of the placement algorithm is when the overflow pile's targeted capacity $R = 0$. This special case will be used when we create the second-tier hash table.

Below, we show that using standard oblivious sorting techniques [2], Placement can be achieved in $O(n \log n)$ total work:

1. For each $i \in [B]$, add Z copies of filler elements $(\diamond, i)$ where $\diamond$ denotes that this is a filler element. These filler elements are there to make sure that each bin is assigned at least Z elements. Note that filler elements and dummy elements are treated differently.

2. Oblivious sort all elements by their bin number. For elements with the same bin number, break ties by placing real elements to the left of filler elements.

3. In a single linear scan, for each element that is not among the first Z elements of its bin, tag the element with the label "`excess`".

4. Oblivious sort all elements by the following ordering function:
   - All dummy elements must appear at the very end;
   - All non-excess elements appear before excess elements;
   - For two non-excess elements, the one with the smaller bin number appears first (breaking ties arbitrarily).
   - For excess elements, place real elements to the left of filler elements.

Oblivious lookups. It remains to show how to make lookup queries oblivious. To achieve this, we can adopt the following simple algorithm:

- If the query $k \neq \perp$: compute the first-tier bin number as $i_1 := \mathsf{PRF}_{\mathsf{sk}}(1||k)$. Read the entire bin numbered i_1 in the first-tier hash table H_1. If found, read an entire random bin in H_2; else compute $i_2 := \mathsf{PRF}_{\mathsf{sk}}(2||k)$ and read the entire bin numbered i_2 in the second-tier hash table H_2. Finally, return the element found or $\perp$ if not found.

- If the query $k = \perp$, read an entire random bin in H_1, and an entire random bin in H_2. Both bin numbers are selected freshly and independently at random. Finally, return $\perp$.

3.3 Construction: Making it Parallel

To make the aforementioned algorithm parallel, it suffices to make the following observations:

i) Oblivious sorting of n elements can be accomplished using a sorting circuit [2] that involves $O(n \log n)$ total work and $O(\log n)$ parallel runtime.

ii) Step 3 of the oblivious building algorithm involves a linear scan of the array marking each excessive element that exceeds its bin's capacity.

   This linear scan can be implemented in parallel using the oblivious "range prefix sum" algorithm in $O(n \log n)$ total work and $O(\log n)$ parallel runtime. We refer the reader to Section 2.4 for a definition of the range prefix sum algorithm.

iii) Finally, observe that the oblivious lookup algorithm involves searching in entire bin for the desired block. This can be accomplished obliviously and in parallel through our "oblivious select" building block defined in Section 2.4. Since each bin's capacity is $O(\log^\epsilon n)$, the oblivious select algorithm can be completed in $O(\log \log n)$ parallel runtime and tight total work.

Remark 2 (The case of small n)**.** So far, we have focused our attention on the (more interesting) case when $n \geq e^{3 \log^\epsilon \lambda}$. When $n < e^{3 \log^\epsilon \lambda}$, we rely on normal hashing, i.e., balls and bins. In this case, hash table building can be achieved through a similar parallel oblivious algorithm that completes in $O(n \log n)$ total work and $O(\log n)$ parallel runtime; further, each lookup query completes obliviously in $O(\alpha \log \lambda)$ total work and $O(\log \log \lambda)$ parallel runtime.

Performance of our oblivious hashing scheme. In summary, the resulting algorithm achieves the following performance:

- Building a hash table with n elements takes $O(n \log n)$ total work and $O(\log n)$ parallel runtime with all but $\mathsf{negl}(\lambda)$ probability, regardless of how large n is.

- Each lookup query takes $O(\log^\epsilon \lambda)$ total work when $n \geq e^{3 \log^\epsilon \lambda}$ and $O(\alpha \log \lambda)$ total work when $n < e^{3 \log^\epsilon \lambda}$ where $\alpha(\lambda) = \omega(1)$ can be any super-constant function. Further, regardless of how large n is, each lookup query can be accomplished in $O(\log \log \lambda)$ parallel runtime.

3.4 Overflow Analysis

We give the overflow analysis of the two-tier construction in Section 3.1. We use the following variant of Chernoff Bound.

Fact 2 (Chernoff Bound for Binomial Distribution). *Let X be a random variable sampled from a binomial distribution (with any parameters). Then, for any $k \geq 2E[X]$, $Pr[X \geq k] \leq e^{-\frac{k}{6}}$.*

Utilization of first-tier hash. Recall that the number of bins is $B := \left\lceil \frac{n}{\log^{\epsilon} \lambda} \right\rceil$. For $i \in [B]$, let X_i denote the number of items that are sent to bin i in the first-tier hash. Observe that the expectation $E[X_i] = \frac{n}{B} \geq \log^{\epsilon} \lambda$.

Overflow from first-tier hash. For $i \in [B]$, let $\widehat{X}_i$ be the number of items that are sent to bin i in the first-tier but have to be sent to the overflow pile because bin i is full. Recall that the capacity of a bin is $Z := 5 \log^{\epsilon} \lambda$. Then, it follows that $\widehat{X}_i$ equals $X_i - Z$ if $X_i > Z$, and 0 otherwise.

Tail bound for overflow pile. We next use the standard technique of moment generating function to give a tail inequality for the number $\sum_i \widehat{X}_i$ of items in the overflow pile. For sufficiently small $t > 0$, we have

$$E[e^{t\widehat{X}_i}] \leq 1 + \sum_{k \geq 1} \Pr[X_i = Z + k] \cdot e^{tk} \leq 1 + \sum_{k \geq 1} \Pr[X_i \geq Z + k] \cdot e^{tk} \leq 1 + \frac{\exp(-\frac{Z}{6})}{e^{\frac{1}{6} - t} - 1},$$

where the last inequality follows from Fact 2 and a standard computation of a geometric series. For the special case $t = \frac{1}{12}$, we have $E[e^{\frac{\widehat{X}_i}{12}}] \leq 1 + 12 \exp(-\frac{Z}{6})$.

Lemma 1 (Tail Inequality for Overflow Pile). *For $k \geq 288 B e^{-\frac{Z}{6}}$, $\Pr[\sum_{i \in [B]} \widehat{X}_i \geq k] \leq e^{-\frac{k}{24}}$.*

Proof. Fix $t := \frac{1}{12}$. Then, we have $\Pr[\sum_{i \in [B]} \widehat{X}_i \geq k] = \Pr[t \sum_{i \in [B]} \widehat{X}_i \geq tk] \leq e^{-tk} \cdot E[e^{t \sum_{i \in [B]} \widehat{X}_i}]$, where the last inequality follows from the Markov's inequality.

As argued in [12], when n balls are thrown independently into n bins uniformly at random, then the numbers X_i's of balls received in the bins are negatively associated. Since $\widehat{X}_i$ is a monotone function of X_i, it follows that the $\widehat{X}_i$'s are also negatively associated. Hence, it follows that $E[e^{t \sum_{i \in [B]} \widehat{X}_i}] \leq \prod_{i \in [B]} E[e^{t\widehat{X}_i}] \leq \exp(12 B e^{-\frac{Z}{6}})$.

Finally, observing that $k \geq 288 B e^{-\frac{Z}{6}}$, we have $\Pr[\sum_{i \in [B]} \widehat{X}_i \geq k] \leq \exp(12 B e^{-\frac{Z}{6}} - \frac{k}{12}) \leq e^{-\frac{k}{24}}$, as required. □

In view of Lemma 1, we consider $N := 288 B e^{-\frac{Z}{6}}$ as an upper bound on the number of items in the overflow pile. The following lemma gives an upper bound on the probability that a particular bin overflows in the second-tier hash.

Lemma 2 (Overflow Probability in the Second-Tier Hash). *Suppose the number of items in the overflow pile is at most $N := 288 B e^{-\frac{Z}{6}}$, and we fix some bin in the second-tier hash. Then, the probability that this bin receives more than Z items in the second tier hash is at most $e^{-\frac{Z^2}{6}}$.*

Proof. Observe that the number of items that a particular bin receives is stochastically dominated by a binomial distribution with N items and probability $\frac{1}{B}$. Hence, the probability that it is at least Z is at most $\binom{N}{Z} \cdot (\frac{1}{B})^Z \leq (\frac{Ne}{Z})^Z \cdot (\frac{1}{B})^Z \leq e^{-\frac{Z^2}{6}}$, as required. □

Corollary 1 (Negligible Overflow Probability). Suppose the number n of items is chosen such that both $B e^{-\frac{Z}{6}}$ and Z^2 are $\omega(\log \lambda)$, where $B := \left\lceil \frac{n}{\log^{\epsilon} \lambda} \right\rceil$ and $Z := \lceil 5 \log^{\epsilon} \lambda \rceil$. Then, the probability that the overflow event happens in the second-tier hash is negligible in λ.

Proof. Recall that $B = \lceil \frac{n}{\log^{\epsilon} \lambda} \rceil$, where $n \geq e^{3 \log^{\epsilon} \lambda}$ in Theorem 1. By choosing $N = 288Be^{-\frac{Z}{6}}$, from Lemma 1, the probability that there are more than N items in the overflow pile is $\exp(-\Theta(N))$, which is negligible in λ.

Given that the number of items in the overflow pile is at most N, according to Lemma 2, the probability that there exists some bin that overflows in the second-tier hash is at most $Be^{-\frac{Z^2}{6}}$ by union bound, which is also negligible in λ, because we assume $B \leq \text{poly}(\lambda)$. □

3.5 Obliviousness

If there is no overflow, for any valid input, Build accesses fixed addresses. Also, given queries are non-recurrent, Lookup fetches a fresh pseudorandom bin for each dummy or non-dummy request. Hence, the simulator is just running Build and Lookup with all dummy requests.

Proof. Concretely, consider the following simulator $\mathsf{Sim}(1^{\lambda}, n, r)$:

- If $r = 1$, recall that the first request is always $(\mathsf{Build}, 1^{\lambda}, S)$. Emulate normal Build with input as $(1^{\lambda}, S')$, where S' is a multiset consisting of n dummy entries. Output the access pattern of the emulation.
- If $r > 1$, recall the request must be $(\mathsf{Lookup}, 1^{\lambda}, k_r)$. Emulate by performing $\mathsf{Lookup}(1^{\lambda}, \mathsf{dummy})$, and output the access pattern of the emulation.

It remains to show that the access pattern of $\mathsf{Expt}^{\text{ideal},\mathcal{F}_{\text{ht}}}_{\mathcal{A},\mathsf{Sim}}(1^{\lambda})$ is computationally indistinguishable from $\mathsf{Expt}^{\text{real},\mathsf{HT}}_{\mathcal{A}}(1^{\lambda})$ if the PRF scheme satisfies pseudorandomness.

Hybrid 1. First, we consider a variant of the real-world algorithm that does not retry upon overflow events, and that the hash bins for each key is chosen based on a random function rather than a pseudorandom function. In this case, the access pattern of experiment in Hybrid 1 and the access pattern of $\mathsf{Expt}^{\text{ideal},\mathcal{F}_{\text{ht}}}_{\mathcal{A},\mathsf{Sim}}(1^{\lambda})$ are identically distributed, as long as the query sequence satisfies the *non-recurrence* property.

Hybrid 2. In Hybrid 2, the experiment is almost identical to Hybrid 1, but we replace the truly random function with a pseudorandom function selected at random from a family of pseudorandom functions. Assuming the security of the pseudorandom function, the addresses in Hybrid 2 are computationally indistingushable from the addresses in Hybrid 1.

Hybrid 3. Now we proceed to modify the algorithm in Hybrid 2 so that upon an overflow event, we retry until success. Due to Corollary 1, the probability over overflow is bounded by $\mathsf{negl}(\lambda)$. It holds that the addresses of the real-world algorithm in Hybrid 3 are statistically close to those of Hybrid 2.

Notice that in the above, adjacent hybrids are indistinguishable even when the adversary can adaptively choose queries. The proof can be completed by observing that Hybrid 3 is the same as the real-world experiment $\mathsf{Expt}^{\text{real},\mathsf{HT}}_{\mathcal{A}}(1^{\lambda})$. □

4 Modular Framework for Hierarchical ORAM

4.1 Preliminary: Hierarchical ORAM from Oblivious Hashing

Goldreich and Ostrovsky [18,19] were the first to define Oblivious RAM (ORAM) and they provide an elegant solution to the problem which was since referred to as the "hierarchical ORAM". Goldreich and Ostrovsky [18, 19] describe a special-case instantiation of a hierarchical ORAM where they adopt an oblivious variant of naïve hashing. Their scheme was later extended and improved by several subsequent works [21, 26, 44].

In this section, we will present a generalized version of Goldreich and Ostrovsky's hierarchical ORAM framework. Specifically, we will show that Goldreich and Ostrovsky's core idea can be interpreted as the following: take any oblivious hashing scheme satisfying the abstraction defined in Section 2.3, we can construct a corresponding ORAM scheme that makes blackbox usage of the oblivious hashing scheme.

From our exposition, it will be clear why such a modular approach is compelling: it makes both the construction and the security proof simple. In comparison, earlier hierarchical ORAM works do not adopt this modular approach, and their conceptual complexity could sometimes confound the security proof [35].

Data structure. There are $\log N + 1$ *levels* numbered $0, 1, \ldots, L$ respectively, where $L := \lceil \log_2 N \rceil$ is the maximum level. Each level is a hash table denoted $\mathsf{T}_0, \mathsf{T}_1, \ldots, \mathsf{T}_L$ where T_i has capacity 2^i. At any time, each table T_i can be in two possible states, *available* or *full*. Available means that this level is currently empty and does not contain any blocks, and thus one can rebuild into this level. Full means that this level currently contains blocks, and therefore an attempt to rebuild into this level will effectively cause a cascading merge.

ORAM operations. Upon any memory access request $(\texttt{read}, \mathsf{addr})$ or $(\texttt{write}, \mathsf{addr}, \mathsf{data})$, perform the following procedure. For simplicity, we omit writing the security parameter of the algorithms, i.e., let $\mathsf{Build}(\cdot) := \mathsf{Build}(1^N, \cdot)$, and let $\mathsf{Lookup}(\cdot) := \mathsf{Lookup}(1^N, \cdot)$.

1. $\mathsf{found} := \texttt{false}$.
2. For each $\ell = 0, 1, \ldots L$ in increasing order,
   - If not found, $\mathsf{fetched} := \mathsf{Lookup}(\mathsf{T}_\ell, \mathsf{addr})$: if $\mathsf{fetched} \neq \bot$, let $\mathsf{found} := \texttt{true}$, $\mathsf{data}^* := \mathsf{fetched}$.
   - Else $\mathsf{Lookup}(\mathsf{T}_\ell, \bot)$.
3. Let $\mathsf{T}^\emptyset := \{(\mathsf{addr}, \mathsf{data}^*)\}$ if this is a $\texttt{read}$ operation; else let $\mathsf{T}^\emptyset := \{(\mathsf{addr}, \mathsf{data})\}$. Now perform the following hash table rebuilding:
   - Let ℓ be the smallest level index such that T_ℓ is marked available. If all levels are marked full, then $\ell := L$. In other words, ℓ is the target level to be rebuilt.
   - Let $S := \mathsf{T}^\emptyset \cup \mathsf{T}_0 \cup \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_{\ell-1}$; if all levels are marked full, then additionally let $S := S \cup \mathsf{T}_L$. Further, tag each non-dummy element in S with its level number, i.e., if a non-dummy element in S comes from T_i, tag it with the level number i.
   - $\mathsf{T}_\ell := \mathsf{Build}(\mathsf{SuppressDuplicate}(S, 2^\ell, \mathsf{pref}))$, and mark T_ℓ as full. Further, let $\mathsf{T}_0 = \mathsf{T}_1 = \ldots = \mathsf{T}_{\ell-1} := \emptyset$ and their status bits set to available. Here we adopt the following priority function pref:

When two or more real blocks with the same address (i.e., key) exist, the one with the smaller level number is preferred (and the algorithm maintains the invariant that no two blocks with the same address and the same level number should exist).

4. Return data^*.

Deamortization. In the context of hierarchical ORAM, a hash table of capacity n is rebuilt every n memory requests, and we typically describe the ORAM's overhead in terms of the amortized cost per memory request. As one may observe, every now and then, the algorithm needs to rebuild a hash table of size N, and thus a small number of memory requests may incur super-linear cost to complete.

A standard deamortization technique was described by Ostrovsky and Shoup [34] to evenly spread the cost of hash table rebuilding over time, and this deamortization framework only blows up the total work of the ORAM scheme by a small constant factor; the details are in Appendix C. In the rest of the paper, we assume that every instance of hash table used in an ORAM scheme is rebuilt in the background using this deamortization technique without explicitly mentioning so. Further, the stated costs in the theorems are applicable to worst-case performance (not just amortized).

Obliviousness. To show obliviousness of the above construction, we make the following observations.

Fact 3 (Non-recurrence condition is preserved)**.** *In the above ORAM construction, it holds that for every hash table instance, all lookup queries it receives satisfy the non-recurrence condition.*

Proof. Due to our ORAM algorithm, every 2^ℓ operations, the old instance of hash table T_ℓ is destroyed and a new hash table instance is created for T_ℓ. It suffices to prove the non-recurrence condition in between every two rebuilds for T_ℓ. Suppose that after T_ℓ is rebuilt in some step, now we focus on the time steps going forward until the next rebuild. Consider when a block block^* is first found in T_ℓ where $\ell \in [L]$, block^* is entered into $\mathsf{T}^\emptyset$. Due to the definition of the ORAM algorithm, until the next time T_ℓ is rebuilt, block^* exists in some $\mathsf{T}_{\ell'}$ where $\ell' < \ell$. Due to the way the ORAM performs lookups — in particular, we would look up a dummy element in T_ℓ if block^* is found in a smaller level — we conclude that until T_ℓ is rebuilt, no lookup query will ever be issued again for block^* to T_ℓ. Note this holds even when queries are adaptively choosen. □

Lemma 3 (Obliviousness)**.** *Suppose that the underlying hashing scheme satisfies correctness and obliviousness as defined in Section 2.3, then it holds that the above ORAM scheme satisfies obliviousness as defined in Section 2.2.*

Proof. Suppose that for every hash table instance, if all of its non-dummy lookup queries are distinct, then due to the obliviousness definition of our oblivious hashing building block (see Section 2.3), there exists a simulator that knowing only the size of the memory and the number of accesses can simulate the joint distribution of all memory accesses. By Fact 3, we have such simulator $\mathsf{Sim}_{\mathrm{ht}}$. Hence, to construct the ORAM simulator, it suffices to run $\mathsf{Sim}_{\mathrm{ht}}$ for each Build or Lookup on each level T_i. □

Theorem 2 (Hierarchical ORAM from oblivious hashing)**.** *Assume the existence of one-way functions and a* $(W_{\mathrm{build}}, W_{\mathrm{lookup}})$*-oblivious hashing scheme. Then, there exists an ORAM scheme that*

achieves the following blowup for block sizes of $\Omega(\log N)$ *bits:*

$$\textit{ORAM's blowup} := \max\left(\sum_{\ell=0}^{\log N} W_{\text{build}}(2^\ell, N),\ \sum_{\ell=0}^{\log N} W_{\text{lookup}}(2^\ell, N)\right) + O(\log^2 N)$$

This theorem is essentially proved by Goldreich and Ostrovsky [18,19] — however, they proved it only for a special case. We generalize their hierarchical ORAM construction and express it modularly to work with any oblivious hashing scheme as defined in Section 2.3.

Remark 3. We point out that due to the way we define our oblivious hashing abstraction, each instance of oblivious hash table will independently generate a fresh PRF key during Build, and this PRF key is stored alongside the resulting hash table data structure in memory. Throughout this paper, we assume that each PRF operation can be evaluated in $O(1)$ runtime on top of our RAM. *We stress that this implicit assumption (or equivalent) was made by all earlier ORAM works [18, 19, 21, 26] that rely on a PRF for security.*

4.2 Preliminary: Improving Hierarchical ORAM by Balancing Reads and Writes

Subsequent to Goldreich and Ostrovsky's ground-breaking result [18, 19], Kushilevitz et al. [26] propose an elegant optimization for the hierarchical ORAM framework such that under some special conditions to be specified later, they can shave a (multiplicative) $\log\log N$ factor off the total work for a hierarchical ORAM scheme. Similarly, Kushilevitz et al. [26] describe a special-case instantiation of an ORAM scheme based on oblivious Cuckoo hashing which was proposed by Goodrich and Mitzenmacher [21].

In this section, we observe that the Kushilevitz et al.'s idea can be generalized. For the sake of exposition, we will first ignore the smaller ORAM levels that employ normal hashing in the following discussion, i.e., we assume that the smaller levels that employ normal hashing will not be a dominating factor in the cost. Now, imagine that there is an oblivious hashing scheme such that for sufficiently large n, the per-element cost for preprocessing is more expensive than the cost of a lookup by a $\log^\delta n$ factor for some constant $\delta > 0$. In other words, imagine that there exists a constant $\delta > 0$ such that the following condition is met for sufficiently large n:

$$\frac{W_{\text{build}}(n, \lambda)}{W_{\text{lookup}}(n, \lambda)} \geq \log^\delta n.$$

If the underlying oblivious hashing scheme satisfies the above condition, then Kushilevitz et al. [26] observes that Goldreich and Ostrovsky's hierarchical ORAM construction is suboptimal in the sense that the cost of fetch phase is asymptotically smaller than the cost of the rebuild phase. Hence, the resulting ORAM's total work will be dominated by the rebuild phase, which is then determined by the building cost of the underlying hashing scheme, i.e., $W_{\text{build}}(n, \lambda)$.

Having observed this, Kushilevitz et al. [26] propose the following modification to Goldreich and Ostrovsky's hierarchical ORAM [18, 19]. In Goldreich and Ostrovsky's ORAM, each level is a factor of 2 larger than the previous level — henceforth the parameter 2 is referred to the *branching factor*. Kushilevitz et al. [26] proposes to adopt a branching factor of $\mu := \log N$ instead of 2, and this would reduce the number of levels to $O(\log N/\log\log N)$ — in this paper, we will adopt a more general choice of $\mu := \log^\phi N$ for a suitable positive constant ϕ. To make this idea work, they allow up to $\mu - 1$ simultaneous hash table instances for any ORAM level. If for all levels below ℓ, all instances of hash tables are full, then all levels below ℓ will be merged into a new hash table residing at level $\ell + 1$. The core idea here is to *balance the cost of the fetch phase and the rebuild*

phase by having a larger branching factor; and as an end result, we could shave a $\log\log N$ factor from the ORAM's total work.

We now elaborate on this idea more formally.

Data structure. Let $\mu := \log^{\phi} N$ for a suitable positive constant ϕ to be determined later. There are $O(\log N/\log\log N)$ *levels* numbered $0, 1, \ldots, L$ respectively, where $L = \lceil \log_{\mu} N \rceil$ denotes the maximum level. Except for level L, for every other $\ell \in \{0, 1, \ldots, L-1\}$: the ℓ-th level contains up to $\mu - 1$ hash tables each of capacity μ^{ℓ}. Henceforth we use the notation T_ℓ to denote level ℓ, and T_ℓ^i to denote the i-th hash table within level ℓ. The largest level L contains a single hash table of capacity N denoted T_L^0. Finally, every level $\ell \in \{0, 1, \ldots, L\}$ has a counter c_ℓ initialized to 0. Effectively, for every level $\ell \neq L$, if $c_\ell = \mu - 1$, then the level is considered full; else the level is considered available.

ORAM operations. Upon any memory access query $(\texttt{read}, \mathsf{addr})$ or $(\texttt{write}, \mathsf{addr}, \mathsf{data})$, perform the following procedure.

1. $\mathsf{found} := \texttt{false}$.

2. For each $\ell = 0, 1, \ldots L$ in increasing order, for $\tau = c_\ell - 1, c_\ell - 2 \ldots 0$ in decreasing order:

   If not found: $\mathsf{fetched} := \mathsf{Lookup}(\mathsf{T}_\ell^\tau, \mathsf{addr})$; if $\mathsf{fetched} \neq \perp$, let $\mathsf{found} := \texttt{true}$, $\mathsf{data}^* := \mathsf{fetched}$. Else $\mathsf{Lookup}(\mathsf{T}_\ell^\tau, \perp)$.

3. Let $\mathsf{T}^{\emptyset} := \{(\mathsf{addr}, \mathsf{data}^*)\}$ if this is a $\texttt{read}$ operation; else let $\mathsf{T}^{\emptyset} := \{(\mathsf{addr}, \mathsf{data})\}$. Now, perform the following hash table rebuilding.

   - Let ℓ be the smallest level index such that its counter $c_\ell < \mu - 1$. If no such level index exists, then let $\ell := L$. In other words, we plan to rebuild a hash table in level ℓ.
   - Let $S := \mathsf{T}^{\emptyset} \cup \mathsf{T}_0 \cup \mathsf{T}_1 \cup \ldots, \cup \mathsf{T}_{\ell-1}$; and if $\ell = L$, additionally, let $S := S \cup \mathsf{T}_L^0$ and let $c_L = 0$. Further, in the process, tag each non-dummy element in S with its level number and its hash table number within the level. For example, if a non-dummy element in S comes from T_i^τ, i.e., the τ-th table in the i-th level, tag it with (i, τ).
   - Let $\mathsf{T}_\ell^{c_\ell} := \mathsf{Build}(\mathsf{SuppressDuplicate}(S, \mu^\ell, \mathsf{pref}))$, and let $c_\ell := c_\ell + 1$.

     Here we adopt the following priority function pref: when two or more blocks with the same address (i.e., key) exist, the one with the *smaller* level number is preferred; if there is a tie in level number, the one with the *larger* hash table number is preferred.
   - Let $\mathsf{T}_0 = \mathsf{T}_1 = \ldots = \mathsf{T}_{\ell-1} := \emptyset$ and set $c_0 = c_1 = \ldots = c_{\ell-1} := 0$.

4. Return data^*.

Goldreich and Ostrovsky's ORAM scheme [18, 19] is a special case of the above for $\mu = 2$.

Deamortization. The deamortization technique of Ostrovsky and Shoup [34] (described in Appendix C) applies in general to hierarchical ORAM schemes for which each level is some data structure that is rebuilt regularly. Therefore, it can be applied to our scheme as well, and thus the work of rebuilding hash tables is spread evenly across memory requests.

Obliviousness. The obliviousness proof is basically identical to that presented in Section 4.1, since the only change here from Section 4.1 is that the parameters are chosen differently due to Kushilevitz et al.'s elegant idea [26].

Theorem 3 (Hierarchical ORAM variant.)**.** *Assume the existence of one-way functions and a* $(W_{\text{build}}, W_{\text{lookup}})$*-oblivious hashing scheme. Then, there exists an ORAM scheme that achieves the following blowup for block sizes of* $\Omega(\log N)$ *bits where* $L = O(\log N/\log\log N)$*:*

$$\textit{ORAM's blowup} := \max\left(\sum_{\ell=0}^{L} W_{\text{build}}(\mu^\ell, N),\ \log^\phi N \cdot \sum_{\ell=0}^{L} W_{\text{lookup}}(\mu^\ell, N)\right) + O(L\log N)$$

We note that Kushilevitz et al. [26] proved a special case of the above theorem, we now generalize their technique and describe it in the most general form.

4.3 Conceptually Simpler ORAM for Small Blocks

In the previous section, we presented a hierarchical ORAM scheme, reparametrized using Kushilevitz et al. [26]'s technique, consuming any oblivious hashing scheme with suitable performance characteristics as a blackbox.

To obtain a conceptually simple ORAM scheme with $O(\log^2 N/\log\log N)$ overhead, it suffices to plug in the oblivious two-tier hashing scheme described earlier in Section 3.

Corollary 2 (Conceptually simpler ORAM for small blocks)**.** There exists an ORAM scheme with $O(\log^2 N/\log\log N)$ runtime blowup for block sizes of $\Omega(\log N)$ bits.

Proof. Using the simple oblivious two-tier hashing scheme in Section 3 with $\epsilon = \frac{3}{4}$, we can set $\phi = \frac{1}{4}$ in Theorem 3 to obtain the result. □

4.4 IO Efficiency and the Case of Large CPU Cache

Besides the ORAM's runtime, we often care about its IO performance as well, where IO-cost is defined as the number of cache misses as in the standard external-memory algorithms literature. When the CPU has a large amount of private cache, e.g., N^ϵ blocks where $\epsilon > 0$ is an arbitrarily small constant, several works have shown that oblivious sorting $n \leq N$ elements can be accomplished with $O(n)$ IO operations [7, 20, 21]. Thus, a direct corollary is that for the case of N^ϵ CPU cache, we can construct a computationally secure ORAM scheme with $O(\log N)$ IO-cost (by using the basic hierarchical ORAM construction with $O(\log N)$ levels with an IO-efficient oblivious sort).

5 Asymptotically Efficient OPRAM

In this section, we show how to construct an $O(\frac{\log^2 N}{\log\log N})$ OPRAM scheme. To do this, we will show how to parallelize our new $O(\frac{\log^2 N}{\log\log N})$-overhead ORAM scheme. Here we benefit tremendously from the conceptual simplicity of our new ORAM scheme. In particular, as mentioned earlier, our oblivious two-tier hashing ($\mathsf{Build}, \mathsf{Lookup}$) algorithms have efficient parallel realizations. We will now present our OPRAM scheme. For simplicity, we first present a scheme assuming that the number of CPUs in each step of the computation is fixed and does not change over time. In this case, we show that parallelizing our earlier ORAM construction boils down to parallelizing the (Build and Lookup) algorithms of the oblivious hashing scheme. We then extend our construction to support the case when the number of CPUs varies over time.

5.1 Intuition

Warmup: uniform number of CPUs. We first describe the easier case of uniform m, i.e., the number of CPUs in the PRAM does not vary over time. Further, we will consider the simpler case when the branching factor $\mu := 2$.

- *Data structure.* Recall that our earlier ORAM scheme builds an exponentially growing hierarchy of oblivious hash tables, of capacities $1, 2, 4, \ldots, N$ each. Here, we can do the same, but we can start the level of hierarchy at capacity $m = 2^i$ (i.e., skip the smaller levels).
- *OPRAM operations.* Given a batch of m simultaneous memory requests, suppose that all addresses requested are distinct — if not, we can run a standard conflict resolution procedure as described by Boyle et al. [5] incurring only $O(\log m)$ parallel steps consuming m CPUs. We now need to serve these requests in parallel. In our earlier ORAM scheme, each request has two stages: 1) reading one block from each level of the exponentially growing hierarchy; and 2) perform necessary rebuilding of the levels. It is not hard to see that the fetch phase can be parallelized easily — particularly, observe that the fetch phase is read-only, and thus having m CPUs performing the reads in parallel will not lead to any write conflicts.

  It remains to show how to parallelize the rebuild phase. Recall that in our earlier ORAM scheme, each level has a status bit whose value is either *available* or *full*. Whenever we access a single block, we find the available (i.e., empty) level ℓ and merge all smaller levels as well as the updated block into level ℓ. If no such level ℓ exists, we simply merge all levels as well as the updated block into the largest level.

  Here in our OPRAM construction, since the smallest level is of size m, we can do something similar. We find the smallest available (i.e., empty) level ℓ, and merge all smaller levels as well as the possibly updated values of the m fetched blocks into level ℓ. If no such level ℓ exists, we simply merge all levels as well as possibly updated values of the m fetched blocks into the largest level. Rebuilding a level in parallel effectively boils down to rebuilding a hash table in parallel (which boils down to performing $O(1)$ number of oblivious sorts in parallel) — which we have shown to be possible earlier in Section 3.

Varying number of CPUs. Our definitions of PRAM and OPRAMs allow the number of CPUs to vary over time. In this case, oblivious simulation of a PRAM is more sophisticated. First, instead of truncating the smaller levels whose size are less than m, here we have to preserve all levels — henceforth we assume that we have an exponentially growing hierarchy with capacities $1, 2, 4, \ldots, N$ respectively. The fetch phase is simple to parallelize as before, since the fetch phase does not make modifications to the data structure. We now describe a modified rebuild phase when serving a batch of $m = 2^\gamma$ requests: note that in the following, γ is a level that matches the current batch size, i.e., the number of CPUs in the present PRAM step of interest:

(a) Suppose level γ is marked available. Then, find the first available (i.e., empty) level ℓ greater than γ. Merge all levels below γ and the updated values of the newly fetched m blocks into level ℓ.

    If no such level ℓ exists, then merge all blocks and the updated values of the newly fetched m blocks into the largest level L.

(b) Suppose level γ is marked as full. Then, find the first available (i.e., empty) level ℓ greater than γ. Merge all levels below or equal to γ (but not the updated values of the m fetched blocks) into level ℓ; rebuild level γ to contain the updated values of the m fetched blocks.

Similarly, if no such level ℓ exists, then merge all blocks and the updated values of the newly fetched m blocks into the largest level L.

One way to view the above algorithm is as follows: let us view the concatenation of all levels' status bits as a binary counter (where full denotes 1 and available denotes 0). If a single block is accessed like in the ORAM case, the counter is incremented, and if a level flips from 0 to 1, this level will be rebuilt. Further, if there would be a carry-over to the $(L+1)$-st level, then the largest level L is rebuilt. However, now m blocks may be requested in a single batch — in this case, the above procedure for rebuilding effectively can be regarded as incrementing the counter by some value v where $v \leq 2m$ — in particular, the value v is chosen such that only $O(1)$ levels must be rebuilt by the above rule.

We now embark on describing the full algorithm — specifically, we will describe for a general choice of the branching factor μ that is not necessarily 2. Further, our description supports the case of varying number of CPUs.

5.2 Detailed Algorithm

Data structure. Same as in Section 4.2. Specifically, there are $O(\log N/\log\log N)$ *levels* numbered $0, 1, \ldots, L$ respectively, where $L = \lceil \log_\mu N \rceil$ denotes the maximum level. Except for level L, for every other $\ell \in \{0, 1, \ldots, L-1\}$: the ℓ-th level contains up to $\mu - 1$ hash tables each of capacity μ^ℓ. Henceforth, we use the notation T_ℓ to denote level ℓ. Moreover, for $0 \leq i < \mu - 1$, we use T_ℓ^i to denote the i-th hash table within level ℓ. The largest level L contains a single hash table of capacity N denoted T_L^0. Finally, every level $\ell \in \{0, 1, \ldots, L\}$ has a counter c_ℓ initialized to 0.

We say that a level $\ell < L$ is *available* if its counter $c_\ell < \mu - 1$; otherwise, $c_\ell = \mu - 1$, and we say that the level $\ell < L$ is *full*. For the largest level L, we say that it is available if $c_L = 0$; else we say that it is full. Note that for the case of general $\mu > 2$, available does not necessarily mean that the level's empty.

OPRAM operations. Upon a batch of m memory access requests $Q := \{\mathsf{op}_p\}_{p \in [m]}$ where each op_p is of the form $(\texttt{read}, \mathsf{addr}_p)$ or $(\texttt{write}, \mathsf{addr}_p, \mathsf{data}_p)$, perform the following procedure. Henceforth we assume that $m = 2^\gamma$ where γ denotes the level whose capacity matches the present batch size.

1. **Conflict resolution.** $Q' := \mathsf{SuppressDuplicate}(Q, m, \mathsf{PRAM.priority})$, i.e., perform conflict resolution on the batch of memory requests Q, and obtain a batch Q' of the same size but where each distinct address appears only once — suppressing duplicates using the PRAM's priority function priority, and padding the resulting set with dummies to length m.

2. **Fetch phase.** For each $\mathsf{op}_i \in Q'$ in parallel where $i \in [m]$, parse $\mathsf{op}_i = \bot$ or $\mathsf{op}_i = (\texttt{read}, \mathsf{addr}_i)$ or $\mathsf{op}_i = (\texttt{write}, \mathsf{addr}_i, \mathsf{data}_i)$:

   (a) If $\mathsf{op}_i = \bot$, let $\mathsf{found} := \texttt{true}$; else let $\mathsf{found} := \texttt{false}$.

   (b) For each $\ell = 0, 1, \ldots L$ in increasing order, for $\tau = c_\ell - 1, c_\ell - 2 \ldots 0$ in decreasing order:
      - If not found: $\mathsf{fetched} := \mathsf{Lookup}(\mathsf{T}_\ell^\tau, \mathsf{addr}_i)$; if $\mathsf{fetched} \neq \bot$, let $\mathsf{found} := \texttt{true}$, $\mathsf{data}_i^* := \mathsf{fetched}$.
      - Else, $\mathsf{Lookup}(\mathsf{T}_\ell^\tau, \bot)$.

3. **Rebuild phase.** For each $\mathsf{op}_i \in Q'$ in parallel where $i \in [m]$: if op_i is a read operation add $(\mathsf{addr}_i, \mathsf{data}_i^*)$ to $\mathsf{T}^\emptyset$; else if op_i is a write operation, add $(\mathsf{addr}_i, \mathsf{data}_i)$ to $\mathsf{T}^\emptyset$; else add $\bot$ to $\mathsf{T}^\emptyset$.

Perform the following hash table rebuilding — recall that γ is the level whose capacity matches the present batch size:

(a) If level γ is full, then skip this step; else, perform the following:
Let $S := \mathsf{T}_0 \cup \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_{\gamma-1}$, and $\mathsf{T}_\gamma^{c_\gamma} := \mathsf{Build}(\mathsf{SuppressDuplicate}(S, \mu^\gamma, \mathsf{pref}))$ where pref prefers a block from a smaller level (i.e., the fresher copy) if multiple blocks of the same address exists.
Let $c_\gamma := c_\gamma + 1$, and for every $j < \gamma$, let $c_j := 0$.

(b)
- At this moment, if level γ is still available, then let $\mathsf{T}_\gamma^{c_\gamma} := \mathsf{Build}(\mathsf{T}^\emptyset)$, and $c_\gamma := c_\gamma + 1$.
- Else, if level γ is full, perform the following:
  Find the first available level $\ell > \gamma$ greater than γ that is available; if no such level ℓ exists, let $\ell := L$ and let $c_L := 0$.
  Let $S := \mathsf{T}^\emptyset \cup \mathsf{T}_0 \cup \ldots \cup \mathsf{T}_{\ell-1}$; if $\ell = L$, additionally include $S := S \cup \mathsf{T}_L$.
  Let $\mathsf{T}_\ell^{c_\ell} := \mathsf{Build}(\mathsf{SuppressDuplicate}(S, \mu^\ell, \mathsf{pref}))$, and let $c_\ell := c_\ell + 1$. For every $j < \ell$, reset $c_j := 0$.

Deamortization. The deamortization technique (described in Appendix C) of Ostrovsky and Shoup [34] applies here as well, and thus the work of rebuilding hash tables are spread evenly across memory requests.

Obliviousness. The obliviousness proof is basically identical to that presented in Section 4.1. Since we explicitly resolve conflict before serving a batch of m requests, we preserve the non-recurrence condition. The only remaining differences here in comparison with Section 4.1 is that 1) here we use a general branching factor of μ rather than 2 (as in Section 4.1); and 2) here we consider the parallel setting. It is clear that neither of these matter to the obliviousness proof.

Theorem 4 (OPRAM from oblivious parallel hashing)**.** *Assume the existence of one-way functions and a $(W_{\text{build}}, T_{\text{build}}, W_{\text{lookup}}, T_{\text{lookup}})$-oblivious hashing scheme. Then, there exists an ORAM scheme that achieves the following performance for block sizes of $\Omega(\log N)$ bits where $L = O(\frac{\log N}{\log \log N})$:*

$$\textit{total work blowup} := \max\left(\sum_{\ell=0}^{L} W_{\text{build}}(\mu^\ell, N),\ \log^\phi N \cdot \sum_{\ell=0}^{L} W_{\text{lookup}}(\mu^\ell, N)\right) + O(L \log N),$$

$$\textit{and para. runtime blowup} := \max\left(\{T_{\text{build}}(\mu^\ell, N)\}_{\ell \in [L]},\ \log^\phi N \cdot \sum_{\ell=0}^{L} T_{\text{lookup}}(\mu^\ell, N)\right) + O(L).$$

Proof. Basically, the proof is our explicit OPRAM construction from any parallel oblivious hashing scheme described earlier in this section. For total work and parallel runtime blowup, we basically take the maximum of the ORAM's fetch phase and rebuild phase. The additive term $O(L \log N)$ in the total work stems from additional building blocks such as parallel duplicate suppression and other steps in our OPRAM scheme; and same for the additive term $O(L)$ in the parallel runtime blowup. □

Using the simple oblivious hashing scheme in Section 3 with $\epsilon = \frac{3}{4}$, we can set $\phi = \frac{1}{4}$ to obtain the following corollary.

Corollary 3 (Asympototically efficient OPRAM for small blocks)**.** Assume that one-way functions exist. Then, there exists a computationally secure OPRAM scheme that achieves $O(\log^2 N / \log \log N)$ simulation overhead when the block size is at least $\Omega(\log N)$ bits.

Acknowledgments

We gratefully acknowledge Kartik Nayak and Ling Ren for numerous inspiring technical discussions during the early stage of the project, and these discussions were of critical help in leading to the two-tier oblivious hashing construction. We thank Kai-Min Chung, Rafael Pass, and Muthuramakrishnan Venkitasubramaniam for consistently supportive discussions. Elaine Shi would like to thank Emil Stefanov for discussions regarding the practical performance of the Goodrich-Mitzenmacher construction which partly inspired this work, and for countless inspiring discussions about ORAMs in general over the years.

This work is supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, CNS-1601879, CNS-1617676, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a DARPA Safeware grant (subcontractor under IBM), a Sloan Fellowship, Google Faculty Research Awards, a Baidu Research Award, and a VMWare Research Award.

References

[1] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Eilstrup Rasmussen. Parallel randomized load balancing. *Random Struct. Algorithms*, 13(2):159–188, 1998.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An O(N Log N) sorting network. In *STOC*, 1983.

[3] Gilad Asharov, Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Oblivious computation with data locality. 2017.

[4] Holger Bast and Torben Hagerup. Fast and reliable parallel hashing. In *SPAA*, 1991.

[5] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *TCC*, pages 175–204, 2016.

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

[7] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. Manuscript, 2017.

[8] T-H. Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *TCC*, 2017.

[9] Binyi Chen, Huijia Lin, and Stefano Tessaro. In *TCC*, pages 205–234, 2016.

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

[11] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, August 1994.

[12] Devdatt P. Dubhashi and Desh Ranjan. Balls and bins: A study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998.

[13] 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.

[14] 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.

[15] 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.

[16] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with O(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[17] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *Eurocrypt*, pages 405–422, 2014.

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

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

[20] Michael T. Goodrich. Data-oblivious External-memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data. In *SPAA*, 2011.

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

[22] 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.

[23] 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.

[24] Torben Hagerup. The log-star revolution. In *STACS*, pages 259–278, 1992.

[25] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009.

[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] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *S & P*, 2015.

[28] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In *Eurocrypt*, 2013.

[29] 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.

[30] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *STOC*, pages 307–316, 1991.

[31] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, October 2001.

[32] Kartik Nayak and Jonathan Katz. An oblivious parallel ram with $o(\log^2 n)$ parallel runtime blowup. `https://eprint.iacr.org/2016/1141`.

[33] 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.

[34] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

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

[36] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security Symposium*, pages 415–430, 2015.

[37] 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*, 2013.

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

[39] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *CCS*, 2013.

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

[41] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[42] 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.

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

[44] 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.

[45] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

A Oblivious Cuckoo Hashing Revisited

In this section, we revisit the elegant oblivious Cuckoo hashing idea by Goodrich and Mitzenmacher [21], and we clarify unspecified, but somewhat non-trivial details of their algorithm and proofs.

A.1 Revisiting Goodrich and Mitzenmacher's Oblivious Cuckoo Hashing Scheme

Background: oblivious Cuckoo hashing. Two concurrent and independent works, Pinkas and Reimann [35] and Goodrich and Mitzenmacher [21] both considered how to rely on a Cuckoo hash table to construct asymptotically efficient ORAMs. It was observed by others [21, 41] that the Pinkas and Reimann [35] construction did not obliviously build the Cuckoo hash table, and consequently their scheme is flawed in terms of security. Goodrich and Mitzenmacher [21] tackled the highly non-trivial challenge of obliviously rebuilding a Cuckoo hashing table by proposing an elegant blueprint. Goodrich and Mitzenmacher [21] showed that the task of building a Cuckoo hash table can be expressed as an efficient MapReduce algorithm. They then showed how to obliviously realize any MapReduce algorithm (where the reduce operation satisfies certain nice properties). Specifically, the map operation can be performed obliviously in a trivial fashion; whereas the reduce algorithm, which boils down to a group-by-key followed by an aggregate-by-key operation, can be implemented through an oblivious sort (implementing group-by-key), followed by a linear scan (implementing aggregate-by-key). Their observation is very elegant and powerful — in fact, they showed that not only a single algorithm, but also a broad class of algorithms, can be made oblivious efficiently. This powerful idea was extended in several subsequent works in the design of efficient oblivious algorithms [27, 33] and in designing programming frameworks for oblivious computation [27, 33].

We now explain Goodrich and Mitzenmacher's blueprint in more detail. In a Cuckoo hash table, each element is assigned to two bins. We may consider each element as an edge that connects the two vertices representing the two bins. This defines what is called a *Cuckoo graph*. The process of building the Cuckoo hash table can be regarded as performing breadth-first-search (BFS) on this Cuckoo graph, such that we can group the vertices by the connected components they are in. In the process of performing BFS, there is a way to assign each element (i.e. edge) to a bin (i.e. an incident vertex) — for the purpose of our discussion, the reader need not know how such an assignment can be made.

Such a BFS algorithm can be expressed efficiently in the MapReduce framework. The goal of the BFS is to let each edge know the vertex with the smallest index in its connected component. Roughly speaking, suppose that at the beginning of each iteration, each edge know some vertex in its connected component. Now, during the iteration, each edge in parallel contacts other edges that share a vertex. Throughout this process, an edge may discover another vertex with smaller index in its component, and hence, it will update this information. The process continues until convergence, i.e., every edge knows the vertex with the smallest index in its component.

Goodrich and Mitzenmacher [21] observe the following: 1) each iteration of the above BFS process can be expressed as $O(1)$ map and $O(1)$ reduce operations, and thus can be implemented through $O(1)$ oblivious sorts; and 2) convergence will happen after super-logarithmic number of iterations except with negligible probability, since the largest connected component can have at most super-logarithmic number of vertices (which also gives a bound on the diameter) except with negligible probability.

Array lengths: to reveal or not reveal? There is, however, one subtlety regarding the length of the array containing the edges that are in "active" components, i.e., those components whose edges have not reached consensus yet. To summarize, building a Cuckoo hashing table obliviously involves super-logarithmic number of MapReduce iterations. When performed in the cleartext, the number of edges in active components will decrease geometrically. For connected components that have reached consensus, the edges inside them no longer need to participate in future iterations. Hence, if a connected component has diameter at most k, it is active only within the first k iterations.

When we perform this procedure obliviously, however, the immediate question is whether the number of *active edges* (i.e., the edges in active components) can be revealed throughout the algorithm. If this number is revealed, then some information is leaked about the structure of the Cuckoo hashing graph. Below, we will demonstrate an explicit attack on the security of the ORAM scheme if this information is leaked to the adversary during the hash table rebuilding.

Goodrich and Mitzenmacher [21] did not explicitly address how to deal with edges in components that have reached consensus. On this front, their algorithm description and their proof of performance bounds appear inconsistent. Their algorithm description (Figure 1 in their paper) seems to suggest that all edges continue to participate until the end. If so, their algorithm would incur $O(\log^3 N)$ cost rather than the claimed $O(\log^2 N)$. Hence, we have to assume that inactive edges are pruned during the execution of BFS — but this will reveal the number of active edges over time during the course of the BFS algorithm. Indeed, Goodrich and Mitzenmacher's stochastic bounds and overhead analysis are also only applicable assuming that the number of inactive edges are pruned during the course of BFS. We now show an explict an attack on the resulting ORAM scheme, assuming that the number of active edges during the course of the BFS are revealed to the adversary.

An explicit attack when the number of active edges is revealed. Before we describe the attack, we first make a remark about one subtle difference regarding how Goodrich and Mitzenmacher [21] and Kushilevitz et al. [26] use oblivious Cuckoo hashing. Specifically, Goodrich and Mitzenmacher employ the technique of placing dummy elements and relying on dummy counters, whereas Kushilevitz et al. does not employ dummy counters. Our attack below directly applies to Kushilevitz et al.'s version of the algorithm, but at the end of our description, we make a remark on one might be able to extend this idea to attack Goodrich and Mitzenmacher's version.

Consider a specific hash table of capacity n in the hierarchical ORAM henceforth called the target hash table, and consider two request sequences:

- *Request sequence 0:* repeatedly ask for the same block whose identifier is 1. In this case, every time 1 will have been found in a smaller level, and thus we would make a dummy request to the target level.
- *Request sequence 1:* keep cycling through the requests $1, 2, \ldots, n$. In this case, consider the subsequent n requests to the target level immediately after the level is rebuilt: for any of these n requests, the requested block will not have been found in a smaller level, and thus we would look for a real block that exists in the target level.

It suffices to show that a polynomial-time adversary can distinguish between the two cases with non-negligible probability, based on its view in the course of the ORAM algorithm.

Recall that when the target level is being rebuilt, consider the rebuilding algorithm as a sampling experiment. The adversary would have observed the following random variables: the number of active edges over time during the course of the BFS algorithm.

Now during the lookup phase, if we have request sequence 0, the adversary observes two random bins (i.e., a fresh random edge) upon every request. If we have request sequence 1, the adversary observes an actual edge that was actually drawn in the rebuild phase.

The adversary's job is to decide which case it is. This translates to deciding whether the random bipartite graph being gradually unveiled in the lookup phase is the same as the random graph of the earlier rebulid phase (where the adversary has observed certain random variables of the graph as mentioned earlier). To show that the adversary has non-negligible advantage in distinguishing the two cases, it suffices observe the following — since the adversary can simply simulate the BFS algorithm on the random graph unveiled during the lookup phase, and compare the observed random variables with those observed in the rebuild phase:

Claim 1. For two random bipartite graphs with n edges generated independently, the probability that running the BFS algorithm on them will lead to exactly the same number of active edges after every iteration is at most some constant $c < 1$.

To summarize, the above constitutes an explicit attack on Kushilevitz et al.'s ORAM scheme [26] assuming that the Cuckoo hashing scheme prunes the number of inactive edges.

Inspired by this idea, it is conceivable that a similar attack would exist against Goodrich and Mitzenmacher's ORAM scheme as well — but the attack there will need to be somewhat more sophisticated, since Goodrich and Mitzenmacher employ the techniques of having dummy blocks and dummy counters, which provides some extent of obfuscation. We note that the Kushilevitz et al. [26] scheme cannot employ the same technique because in their scheme, each hash table must be accessed more times than Goodrich and Mitzenmacher, and since the dummy counter technique requires that the number of dummy elements match the number of accesses, there would be too many dummy elements which would make their scheme asymptotically more expensive.

B Oblivious Cuckoo Hashing Clarified

In this section, we clarify Goodrich and Mitzenmacher's elegant blueprint of obliviously performing Cuckoo hashing. We show that some non-trivial additional steps are needed in both the algorithm and the proofs to correctly implement their blueprint. We include this section for completeness — however, for constructing ORAMs and OPRAMs, we instead recommend the usage of conceptually simpler hashing schemes as described in the main body of the paper.

Intuition. As mentioned earlier, if we do not prune any edges throughout the hash-table rebuilding algorithm, the cost would be too high to obtain an $O(\log^2 N)$. We therefore would like to prune inactive edges during the process. Unfortunately, as argued earlier, we also cannot reveal the true number of active edges since this would result in leakage of the Cuckoo hashing graph, and thus lead to an explicit attack on the resulting ORAM's security.

Our idea is to prune inactive edges using an a priori fixed schedule that is not dependent on the concrete sample of the Cuckoo graph. This a priori fixed schedule will decrease geometrically — but possibly slower than the actual number of active edges. Specifically, the number of active edges left after the k-th iteration is roughly the number of edges residing in connected components of size k or larger — henceforth we use the random variable T_k to denote this quantity. Roughly speaking, we would like to prove statements of the following form: for a suitable constant $\beta \in (0, 1)$, as long as k is not too large, the following holds

$$\Pr[T_k \geq O(n\beta^k)] \leq \mathsf{negl}(\lambda)$$

In some sense, the above statement is a bit stronger than the stochastic bounds that Goodrich and Mitzenmacher actually proved [21], and their analysis does not trivially imply such a statement — in fact, to prove this, we need to modify their Martingale analysis in a non-blackbox manner (see Section B.1).

Roadmap. In the remainder of this section, we will do the following:

- In Section B.1, we will formalize and prove the above stochastic bounds regarding the Cuckoo hash graph;
- In Section B.2, we will describe a complete algorithm for obliviously building a Cuckoo hash table. Specifically, we will describe how exactly to prune arrays during the process of oblivious hash table rebuilding, an important detail that is seemingly unspecified in the earlier work. Finally, we will comment on how to leverage oblivious Cuckoo hashing to construct an ORAM.

B.1 Analysis of Cuckoo Graph

In this section, we prove a high-probability upper bound on how fast the array lengths decrease in the aforementioned BFS exploration of the random Cuckoo graph. Based on this bound, we will explicitly describe how to obliviously prune arrays during the BFS exploration in Section B.2 — importantly, we will prune only up to this a priori known upper bound, not to the true array lengths.

We first establish some notations and terminology. Let λ be the secure parameter such that the failure probability of the hashing scheme is $\mathsf{negl}(\lambda)$. For $n \geq \log^8 \lambda$, the cuckoo hashing scheme stores n distinct keys using the data structure T consisting of three arrays: two tables $\mathsf{T}_1, \mathsf{T}_2$ of size $m = (1+\epsilon)n$, and one stash S of size $O(\log \lambda)$, where $\epsilon > 0$ is a constant. The scheme uses two functions $\mathsf{PRF}_{\mathsf{sk}}(1||\cdot)$, $\mathsf{PRF}_{\mathsf{sk}}(2||\cdot)$ mapping from the key space to $[m]$, where PRF key $\mathsf{sk} \overset{\$}{\leftarrow} \{0,1\}^\lambda$ is freshly generated in each Build, and we use PRF_1 and PRF_2 in the following for short. For each key k, it is stored in one of the following locations: (a) $\mathsf{PRF}_1(k)$-th slot of T_1, (b) $\mathsf{PRF}_2(k)$-th slot of T_2, or (c) some slot of S, where we always prefer to store elements in T_1 and T_2 rather S. Each slot accommodates only one key. After building the hast table $\mathsf{T} = \{\mathsf{T}_1, \mathsf{T}_2, \mathsf{S}\}$, a straightforward Lookup can be done with total work $O(\log \lambda)$. Therefore, in the following, we analyze cuckoo hashing using a bipartite graph. Specifically, we describe the algorithm Build, which takes $O(n \log n)$ total work.

Recall that the cuckoo graph is formed in the following way. For some constant $\epsilon > 0$, n independently random edges are inserted into a bipartite graph, where each part has $m = (1+\epsilon)n$ vertices, in the following way: each edge's endpoints are sampled uniformly at random from each of the two parts. The following fact is proved in [25, Lemma 2.4].

Fact 4 (Every Vertex in Small Component)**.** *For any vertex v in the cuckoo graph, let C_v be the connected component containing v. Then, there exists a constant $\beta \in (0,1)$ (depending on the parameter ϵ chosen for the cuckoo graph) such that for any vertex v and integer $k \geq 2$, $\Pr[|C_v| \geq k] \leq \beta^k$.*

Lemma 4. *Consider the cuckoo graph defined above. For any (random) edge e, let C_e be the connected component containing the edge e. Then, there exists a constant $\beta \in (0,1)$ (depending on the parameter ϵ chosen for the cuckoo graph) such that for any vertex v and integer $k \geq 3$, $\Pr[|C_e| \geq k] \leq 2\beta^k$.*

Proof. We use Fact 4 for the existence of some $\widehat{\beta} \in (0,1)$ such that for every vertex v and integer k, it holds that the probability $\Pr[|C_v| \geq k] \leq \widehat{\beta}^k$. We set $\beta := \sqrt{\widehat{\beta}}$.

Fix some integer k and some edge e. We condition on the end-points x and y of edge e. We remove edge e from the cuckoo graph, and consider the randomness due to the remaining edges. For each vertex v, let $\widehat{C}_v$ be the component containing v in the remaining graph. Since the remaining graph has fewer (random) edges, by a standard coupling argument, it follows that $\Pr[|\widehat{C}_v| \geq k|e] \leq \widehat{\beta}^k$.

Since $|C_v| \geq k$ implies that either $\widehat{C}_x$ or $\widehat{C}_y$ contains at least $\lceil \frac{k}{2} \rceil$ vertices, we have $\Pr[|C_e| \geq k] = E_{x,y}[\Pr[|C_e| \geq k|e = \{x, y\}]] \leq E_{x,y}[\Pr[|\widehat{C}_x| \geq \lceil \frac{k}{2} \rceil |e] + \Pr[|\widehat{C}_y| \geq \lceil \frac{k}{2} \rceil |e]]$, which is at most $2\beta^k$, as required. □

Lemma 5. *Let $T_k := |\{e \mid |C_e| \geq k\}|$ be the number of edges e whose component C_e in the cuckoo graph contains at least k vertices. Suppose $\beta \in (0, 1)$ is the constant (depending on ϵ) from Lemma 4. For any integer k such that $n\beta^k \geq n^{0.87}$, $\Pr[T_k \geq 3n\beta^k] \leq \mathsf{negl}(\lambda)$, where $n \geq \log^8 \lambda$.*

Proof. Fix some k such that $n\beta^k \geq n^{0.87}$. Define $\mathcal{E}$ to be the event that there exists some edge e such that $|C_e| > K := \log \lambda \log \log \lambda$. Lemma 4 implies that $\Pr[\mathcal{E}] \leq \mathsf{negl}(\lambda)$.

We next define the truncated random variable $\widehat{T}_k := \{e|k \leq |C_e| \leq K\}$.

Consider the n edges in any arbitrary order, and define the filtration $\{\mathcal{F}_i : i = 0, 1, \ldots, n\}$, where $\mathcal{F}_i$ corresponds to the randomness due to the first i edges.

Next, define the Doob martingale $X_i := E[\widehat{T}_k|\mathcal{F}_i]$ for $i = 0, 1, \ldots, n]$. In order to use Azuma's inequality, we need to give a uniform upper bound on $|X_i - X_{i+1}|$. By conditioning on the randomness of the first i edges and applying a coupling argument on the $(i+2)$nd to the nth edge, it suffices to consider what happens to $\widehat{T}_k$ when a single edge changes.

It suffices to consider how $\widehat{T}_k$ can change when a single edge is removed. Observe that $\widehat{T}_k$ changes by more than 1 only when removing an edge breaks a component C into two components C_1 and C_2. The random variable can increase if originally C contains more than K vertices, but after removal at least one of C_1 and C_2 has size between k and K; hence, the increase in $\widehat{T}_k$ is at most K^2. The random variable can also decrease by more than 1 if $k \leq |C| \leq K$, but both $|C_1|$ and $|C_2|$ are less than k, in which case $\widehat{T}_k$ decreases by at most $\frac{K^2}{2}$, due to the edges in C.

Therefore, we can conclude that with probability 1, $|X_i - X_{i+1}| \leq c := 2K^2$.

Hence, By Azuma's inequality, for any $\rho > 0$,

$$\Pr[X_n \geq X_0 + \rho] \leq \exp(-\frac{2\rho^2}{nc^2}) = \exp(-\frac{\rho^2}{2nK^4}).$$

By definition of X_i and Lemma 4, $X_0 = E[\widehat{T}_k] \leq E[T_k] = \sum_e \Pr[C_e \geq k] \leq 2n\beta^k$, and $X_n = \widehat{T}_k$. Thus, we have

$$\Pr[\widehat{T}_k \geq 2n\beta^k + \rho] \leq \Pr[X_n \geq X_0 + \rho] \leq \exp(-\frac{\rho^2}{2nK^4}).$$

Choosing $\rho = n\beta^k \geq n^{0.87}$ yields $\Pr[\widehat{T}_k \geq 3n\beta^k] \leq \exp(-\log^{1.5} \lambda) \leq \mathsf{negl}(\lambda)$.

Finally, we have $\Pr[T_k \geq 3n\beta^k] \leq \Pr[T_k \geq 3n\beta^k \wedge \overline{\mathcal{E}}] + \Pr[\mathcal{E}] \leq \Pr[\widehat{T}_k \geq 3n\beta^k] + \Pr[\mathcal{E}] \leq \mathsf{negl}(\lambda)$, where we use the observation that the event $\overline{\mathcal{E}}$ implies that $T_k = \widehat{T}_k$. □

B.2 Complete Algorithm Description for Oblivious Cuckoo Hashing

In this section, we give a complete description of an oblivious Cuckoo hashing scheme — specifically, we explicitly spell out important details regarding how to prune array lengths. Although Goodrich and Mitzenmacher [21] describe their oblivious Cuckoo hashing algorithm using MapReduce as an intermediate abstraction, for concreteness and ease of understanding, we skip the intermediate abstraction and spell out the exact algorithm.

Recall that the cuckoo graph is a bipartite graph $G = (V_1 \cup V_2, E)$, where V_1 and V_2 corresponds to slots in the two hash tables, and each edge e corresponds to some key x such that $e = \{\mathsf{PRF}_1(x), \mathsf{PRF}_2(x)\}$.

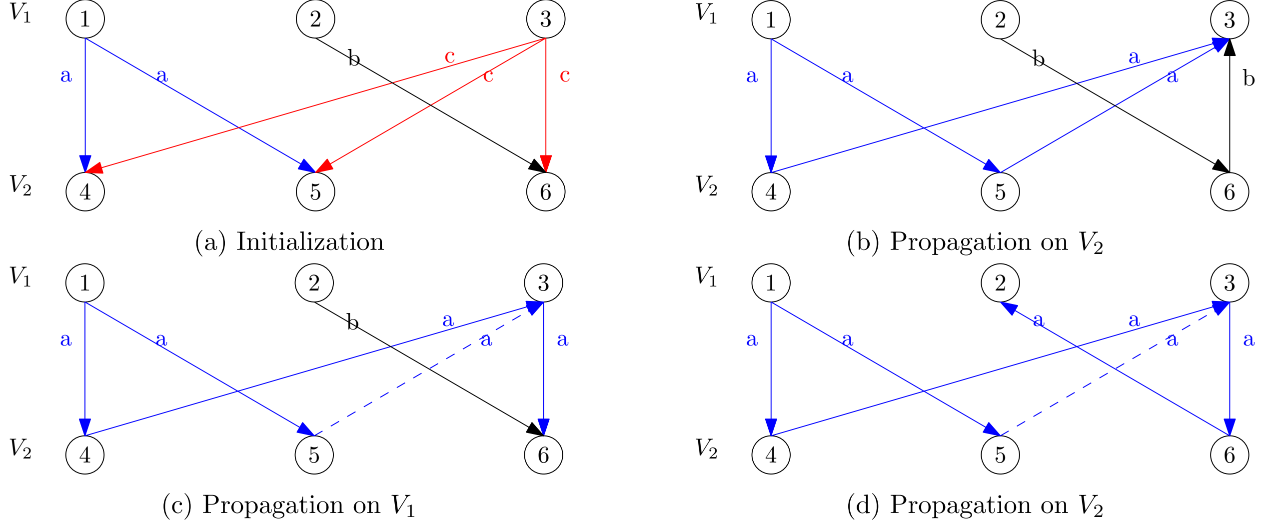
Oblivious and Parallel Breadth-First-Search (BFS). The goal of this subroutine is to build a BFS tree for each connected component in G. Specifically, the root of a component C will be the vertex in $V_1 \cap C$ with the smallest index. After BSF is performed, each edge e in the component C will be labeled as either a *tree edge* or a *cylce edge*. Moreover, every edge e will be assigned a direction away from the root, and hence the head of the edge is the vertex that is further away from the root in the BFS tree.

Algorithm $\mathsf{ObiviousBFS}(x_1, \ldots, x_n)$:
Input: a list of unique keys $x_1, \ldots, x_n$ (and implicitly the secret key of $\mathsf{PRF}_1, \mathsf{PRF}_2$).
Output: create BFS tree for each connected component.

Initialization. For each key x, we create an entry $\mathsf{ent} := (x, u_1 := \mathsf{PRF}_1(x), u_2 := \mathsf{PRF}_2(x), c, l, t)$, where the fields (c, l, t) have the following meaning. The field c indicates the root of the BFS tree that currently contains the edge of ent, and is initialized to $\mathsf{PRF}_1(x)$. The field $l \in \{1, 2\}$ indicates that the edge is directed towards V_l, and is initialized to 2. The field $t \in \{0, 1\}$ indicates whether the edge is a tree edge (1) or a cycle edge (0), and is initialized to 1.

Propagation. In one round of propagation, there are two phases. In the first phase, we extend the BFS tree from V_2, and in the second phase, we do so from V_1. The two phases are similar, and we give the description for Extend_i, where $i = 2, 1$. (Hence, we first perform Extend_2, followed by Extend_1.) For convenience, we use $i \oplus 1$ to indicate flipping between 1 and 2.

1. We perform oblivious sort on the entries according to the following fields in decreasing priority:
   (a) u_i: Edges incident on the same vertex in V_i are grouped together.
   (b) c: Edges labeled with the same root are grouped together, where a root with smaller index has higher priority.
   (c) l: Edges directing towards V_l have higher priority.
   (d) t: Tree edges (1) have higher priority.
   (e) Remaining ties are resolved consistently, for instance, using the key.
2. After sorting, the entries are grouped according to the vertex in V_i they share. By oblivious aggregation, each entry ent also knows the entry ent_0 (and its corresponding fields (c_0, l_0, t_0)) that has the highest priority in its group. In parallel over all entries, the fields (c, l, t) of an entry ent are updated (with respect to the fields (c_0, l_0, t_0)) according to the following rules:
   (a) If $\mathsf{ent} = \mathsf{ent}_0$, then the fields (c, l, t) remain unchanged.
   (b) If $c \neq c_0$, then the entry ent needs to update its root, and we perform the updates $c \leftarrow c_0, l \leftarrow i \oplus 1, t \leftarrow 1$; observe that this edge directs away from V_i and is potentially tagged as a tree edge.



(a) Initialization

(b) Propagation on V_2

(c) Propagation on V_1

(d) Propagation on V_2

Figure 1: An example of propagation on a cuckoo graph, where each entry is represented by an arrow that has fields color (a, b, c), direction ($V_1 \to V_2$ or $V_2 \to V_1$), and type (solid line for a tree edge, and dashed for a cycle edge).

(c) If $c = c_0$ and $l = i \oplus 1$, then the edge of ent directs away from V_i and its fields (c, l, t) remain unchanged.

(d) If $c = c_0$ and $l = i$, then in this case, we must have $l_0 = i$. Since the in-degree of every vertex due to tree edges is at most 1, for the entry $\mathsf{ent} \neq \mathsf{ent}_0$, either ent is already indicated as a cycle edge, or else we update its field $t \leftarrow 0$.

Oblivious pruning. Observe that if a component has k vertices, then after k rounds of propagation, the entries ent contained in that component will not be updated anymore. To reduce total work, we do not have to include these entries in future propagation steps. However, the exact size of a component will leak information, and hence, we need to perform pruning obliviously.

To this end, Lemma 5 states that after k rounds of propagation, with all but negligible probability, at most $3n\beta^k$ entries are not finished yet. Hence, we can gradually decrease the number of entries that we keep in running the first k_0 rounds of propagation, where k_0 is the largest integer such that $n\beta^k \geq n^{0.87}$.

It remains to describe how we detect whether an entry is finished. Observe that a component is finished if the c fields of all its entries stay exactly the same after one round of propagation. Hence, in each round of pruning, we perform oblivious sorting on the entries according to the field c first and resolve ties consistently (for instance according to the key). Each entry then remembers which entry follows it immediately, except for the last entry in the same component, and the first entry in the component remembers that it is the first entry in the group. During a round of pruning, an entry checks that its relevant information is not changed from the previous round and marks itself as potentially ready. Then, oblivious aggregation over all entries in the same component can decide if all the entries in that group are finished.

Observe that $k_0 = O(\log n)$. Since the entries decreases geometrically in these first k_0 rounds, the total work is $O(n \log n)$.

After the first k_0 rounds of propagation, the number of entries after pruning is at most $O(n^{0.87})$. We simply run $K = \log \lambda \log \log \lambda$ more rounds of propagation to finish all entries. This takes total

work $O(n^{0.87} \log n) \cdot K = O(n \log n)$.

Build Cuckoo Hash Table. After the BFS tree is constructed for each component, the information in the (c, l, t) fields are used to build the hash table.

Intuitively, for each component, all the entries corresponding to the tree edges, i.e., $t = 1$, will have their keys stored in table T_1 or T_2, where exactly which table is indicated by the field l. The keys for the cycle edges are stored in the stash.

However, to reduce the stash size, it is possible to store the key of one cycle edge in one of the tables as well. One arbitrary cycle edge $e = (u, v)$ (which is currently directing towards v) is picked and there is a unique path P in the BFS tree from the root r to the vertex u. For the edge e, we store its key in the slot corresponding to u. For every tree edge in the path P, we reverse the direction of the edge and store the key in the slot corresponding to the vertex closer to the root.

Observe that path P can be discovered by an oblivious procedure similar to the propagation procedure, and we omit the details.

Lemma 6. *For any integer $s \geq 1$, the size S of the stash after all items have been inserted satisfies $\Pr[S \geq s] = O(n^{-s})$. Hence, $\Pr[S \geq \log \lambda] = \mathsf{negl}(\lambda)$.*

Proof. We use Theorem 2.1 in [25], which builds cuckoo hash table in the following way: (a) inserts a key into its slot in T_1, and (b) *evicts* the other key if the slot is occupied, (c) evicts iteratively if the next slot is still occupied. In addition, if no empty slots after $A \log n$ evictions, then the key is placed in the stash S, where A is some large enough constant. Using this strategy, they showed that the probability that the stash has size at least s is at most $O(n^{-s})$.

Observe that the procedure we describe using BFS is more aggressive, because it is equivalent to searching for an empty slot with no upper bound on the number of evictions. Hence, the probability bound still holds. □

Theorem 5 (Oblivious Cuckoo Hashing)**.** *Given security parameter λ, cuckoo hashing with $n \geq \log^8 \lambda$ is a $(W_{\text{build}}, T_{\text{build}}, W_{\text{lookup}}, T_{\text{lookup}})$-parallel oblivious hashing scheme where*

$$W_{\text{build}} = O(\log n), \quad T_{\text{build}} = O(\log^2 n + \log n \log \lambda \log \log \lambda),$$
$$W_{\text{lookup}} = O(\log \lambda), \quad T_{\text{lookup}} = O(\log \log \lambda).$$

Remark: Using Cuckoo Hashing Scheme for ORAM. Observe that if we use Cuckoo Hashing directly, according to Theorem 5, each lookup takes time $\Omega(\log \lambda)$ work because of the stash. Although this seems to defeat the advantage of Cuckoo hashing, as described in [26], a shared stash (with capacity, say, at most $O(\log^{1.9} N)$) can be used to store the overflowing keys for all instances of cuckoo hashing. Therefore, not counting the work for looking up in the shared stash, each lookup in an instance of cuckoo hashing takes only $O(1)$ work. Hence, we can still use a variant of Theorem 3 to achieve an ORAM scheme with $O(\frac{\log^2 N}{\log \log N})$ runtime blowup for block sizes of $\Omega(\log N)$ bits.

Obliviousness. Our goal is to show the above construction satisfies the obliviousness defined in Section 2.3: there exists a simulator Sim such that for any *non-recurrent* query sequence $\mathbf{k} := \{k_2, \ldots, k_{\mathsf{poly}(\lambda)}\}$, it holds that $\mathsf{Expt}^{\text{ideal}, \mathcal{F}_{\text{ht}}}_{\mathcal{A}, \mathsf{Sim}}(1^\lambda)$ is computationally indistinguishable from $\mathsf{Expt}^{\text{real}, \mathsf{HT}}_{\mathcal{A}}(1^\lambda)$.

The simulator Sim can be constructed by simulating as follows: to simulate Build, just output the access pattern $\mathsf{Build}(1^\lambda, S')$, where S' is a list consisting of $|S|$ dummy entries. To simulate

Lookup on each query k_r (on the hash table), it suffices to take all k_r queries as dummy queries, and then call the real-world Lookup algorithm using dummy queries as inputs.

The remainder of the proof can be accomplished through a sequence of hybrids in almost the same manner as in Section 3.5.

C Deamortization

The high level idea of deamortizing hierarchical ORAMs is due to Ostrovsky and Shoup [34]. The original description in [34] works for the case when the size of hash tables grows by a factor of 2 between adjacent levels. We give the details here for the case when the factor μ of growth is arbitrary. For simplicity, we first describe the sequential case for ORAM.

Re-building from Smaller Levels. Recall that in Sections 4.2 and 5.2, a hash table with capacity $\mu^{\ell+1}$ is rebuilt from hash tables from levels 0 to ℓ as an atomic procedure. To carry out the amortized analysis, we break this procedure into several phases such that in each phase, the elements from μ hash tables of the same capacity (say μ^i) are combined to build a bigger hash table (say with μ^{i+1} capacity).

Properties of the Underlying Oblivious Hashing Scheme. We need the following properties of the oblivious hashing scheme. At each level $i \geq 1$, a hash table with capacity μ^i satisfies:

- It is built from at most μ smaller hash tables with capacity μ^{i-1}. The total work for building will be amortized over $(\mu - 1)\mu^{i-1}$ requests.
- It can support a sequence of non-recurrent requests with length at most $3\mu^{i+1}$.

Modified Data Structure. Denote $L = \lceil \log_\mu N \rceil$, and the levels are numbered $0, 1, \ldots, L$. In each level $0 \leq \ell < L$, we use T_ℓ to denote level ℓ, which consists a FIFO queue of at most $2\mu - 1$ hash tables, each with capacity μ^ℓ; the queue in level L contains at most 1 hash table. For $0 \leq i < 2\mu - 1$, we use T_ℓ^i to denote the i-th hash table within level ℓ, where smaller i indicates joining the queue earlier. Moreover, each level ℓ has a counter c_ℓ indicating the number of hash tables in the corresponding queue. In this deamortization, one observation is that those hash tables to be rebuilt is read-only in lookup phases, so we can copy them to another space on-the-fly. Also, delaying the output of rebuild could create more duplicated elements, but those duplication can be suppressed in the same way. Therefore, the rebuild task can be done in the background as we doubled the number of hash tables in each level.

Data Structure Invariant. When there are at least μ hash tables in the queue T_ℓ, the elements from the μ oldest hash tables will be copied to build a hash table with capacity $\mu^{\ell+1}$ in the background. Before this building is finished, these older hash tables are still accessible.

Fetch Phase. This is exactly the same as before, except that there might be more hash tables in each level.

Rebuild Phase. For each request, the algorithm performs the following. The block just requested is added to the queue T_0 as a hash table with capacity 1. Each level $\ell \in \{0, 1, 2, \ldots, L-1\}$ performs the following independently:

1. If there are at least μ hash tables in the queue T_ℓ and no rebuilding is currently under progress, then initiate rebuilding from the μ oldest hash tables in the queue.
2. If the oldest hash table has been in the queue for more than $2\mu^{\ell+1}$ requests and no rebuilding is under progress because there are less than μ hash tables in the queue T_ℓ, then initiate rebuilding a new hash table of capacity $\mu^{\ell+1}$ from the existing hash tables in the queue.
3. If the rebuilding of some hash table (of capacity $\mu^{\ell+1}$) is already initiated, then perform the work of rebuilding allocated to this request. Recall that the work for rebuilding a hash table at this level is allocated to $(\mu - 1)\mu^\ell$ requests.

4. If the rebuilding at this level (to produce a hash table of capacity $\mu^{\ell+1}$ is finished, then remove the oldest μ hash tables from the queue and pass the newly built hash table to the next level $\ell + 1$.
5. Any hash table received from level $\ell - 1$ is inserted into the queue.

The case for $\ell = L$ is special in the sense that it will start rebuilding (a hash table with capacity μ^L as long as its queue T_L contains at least 2 hash tables.

Lemma 7 (Correctness of Deamortization)**.** *The correctness follows from the two properties:*

(a) For each ℓ, in μ^ℓ consecutive requests, level ℓ receives at most one hash table (with capacity μ^ℓ).

(b) For each ℓ, any hash table stays in the queue for at most $3\mu^{\ell+1}$ requests.

Proof. We prove statement (a) by induction. The case $\ell = 0$ holds trivially. For the inductive step, it suffices to show that assuming the truth for the case ℓ, then in any $\mu^{\ell+1}$ consecutive requests, at most one hash table is sent from T_ℓ to $\mathsf{T}_{\ell+1}$.

Observe that rebuilding at level ℓ starts as soon as there are at least μ hash tables in T_ℓ. Hence, by the time a hash table of capacity $\mu^{\ell+1}$ is rebuilt and sent to level $T_{\ell+1}$, at most $\mu - 1$ hash tables of capacity μ^ℓ are in the remaining queue T_ℓ. Hence, it takes a further μ^ℓ requests for another hash table to arrive in T_ℓ; after that, it takes another $(\mu - 1)\mu^\ell$ requests to finish rebuilding another hash table of capacity $\mu^{\ell+1}$. Therefore, it takes at least $\mu^{\ell+1}$ requests to send another hash table to $T_{\ell+1}$.

Statement (b) holds because rebuilding will be initiated if any hash table stays in the queue T_ℓ for more than $2\mu^{\ell+1}$ requests, and rebuilding takes at most $(\mu - 1)\mu^\ell < \mu^{\ell+1}$ requests. □

Deamortization Analysis. Suppose the work needed to build a hash table of capacity μ^ℓ is $W(\mu^\ell)$. By our construction, this work is spread over $(\mu - 1)\mu^{\mu-1} = \Theta(\mu^\ell)$ requests. Moreover, Lemma 7 states that in μ^ℓ consecutive reqeusts, at most 1 hash table of capacity μ^ℓ will be produced. Therefore, in every request, the work used to rebuild a hash table of capacity μ^ℓ is at most $\frac{1}{\mu^\ell} W(\mu^\ell)$.