Oblivious Parallel RAM and Applications

Elette Boyle*
Technion Israel
eboyle@alum.mit.edu

Kai-Min Chung
Academica Sinica
kmchung@iis.sinica.edu.tw

Rafael Pass†
Cornell University
rafael@cs.cornell.edu

August 31, 2015

Abstract

We initiate the study of cryptography for *parallel RAM (PRAM)* programs. The PRAM model captures modern multi-core architectures and cluster computing models, where several processors execute in parallel and make accesses to shared memory, and provides the "best of both" circuit and RAM models, supporting both cheap random access and parallelism.

We propose and attain the notion of *Oblivious PRAM*. We present a compiler taking any PRAM into one whose distribution of memory accesses is statistically independent of the data (with negligible error), while only incurring a polylogarithmic slowdown (in both total and *parallel* complexity). We discuss applications of such a compiler, building upon recent advances relying on Oblivious (sequential) RAM (Goldreich Ostrovsky JACM'12). In particular, we demonstrate the construction of a *garbled PRAM* compiler based on an OPRAM compiler and secure identity-based encryption.

*The research of the first author has received funding from the European Union's Tenth Framework Programme (FP10/ 2010-2016) under grant agreement no. 259426 ERC-CaC, and ISF grant 1709/14.

†Pass is supported in part by a Google Faculty Award, Alfred P. Sloan Fellowship, Microsoft New Faculty Fellowship, NSF Award CNS-1217821, NSF CAREER Award CCF-0746990, NSF Award CCF-1214844, AFOSR YIP Award FA9550-10-1-0093, and DARPA and AFRL under contract FA8750-11-2- 0211. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

†This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467.

1 Introduction

Completeness results in cryptography provide general transformations from arbitrary functionalities described in a particular computational model, to solutions for executing the functionality securely within a desired adversarial model. Classic results, stemming from [Yao82, GMW87], modeled computation as *boolean circuits*, and showed how to emulate the circuit securely gate by gate.

As the complexity of modern computing tasks scales at tremendous rates, it has become clear that the circuit model is not appropriate: Converting "lightweight," optimized programs first into a circuit in order to obtain security is not a viable option. Large effort has recently been focused on enabling direct support of functionalities modeled as Turing machines or random-access machines (RAM) (e.g., [OS97, GKK+12, LO13a, GKP+13, GHRW14, GHL+14, GLOS15, CHJV15, BGL+15, KLW15]). This approach avoids several sources of expensive overhead in converting modern programs into circuit representations. However, it actually introduces a different dimension of inefficiency. RAM (and single-tape Turing) machines do not support *parallelism*: thus, even if an insecure program can be heavily parallelized, its secure version will be inherently *sequential*.

Modern computing architectures are better captured by the notion of a *Parallel RAM (PRAM)*. In the PRAM model of computation, several (polynomially many) CPUs are simultaneously running, accessing the same shared "external" memory. Note that PRAM CPUs can model physical processors within a single multicore system, as well as distinct computing entities within a distributed computing environment. We consider an expressive model where the number of active CPUs may vary over time (as long as the pattern of activation is fixed a priori). In this sense, PRAMs capture the "best of both" RAM and the circuit models: A RAM program handles random access but is entirely sequential, circuits handle parallelism with variable number of parallel resources (i.e., the circuit width), but not random access; variable CPU PRAMs capture both random access and variable parallel resources. We thus put forth the challenge of designing cryptographic primitives that directly support PRAM computations, while preserving computational resources (total computational complexity and parallel time) up to poly logarithmic, while using the same number of parallel processors.

Oblivious Parallel RAM (OPRAM). A core step toward this goal is to ensure that secret information is not leaked via the *memory access patterns* of the resulting program execution.

A machine is said to be *memory oblivious*, or simply *oblivious*, if the sequences of memory accesses made by the machine on two inputs with the same running time are identically (or close to identically) distributed. In the late 1970s, Pippenger and Fischer [PF79] showed that any Turing Machine Π can be compiled into an oblivious one Π' (where "memory accesses" correspond to the movement of the head on the tape) with only a logarithmic slowdown in running-time. Roughly ten years later, Goldreich and Ostrovsky [Gol87, GO96] proposed the notion of Oblivious RAM (ORAM), and showed a similar transformation result with polylogarithmic slowdown. In recent years, ORAM compilers have become a central tool in developing cryptography for RAM programs, and a great deal of research has gone toward improving both the asymptotic and concrete efficiency of ORAM compilers (e.g., [Ajt10, DMN11, GMOT11, KLO12, CP13, CLP14, GGH+13, SvDS+13, CLP14, WHC+14, RFK+14, WCS14]). However, for all such compilers, the resulting program is inherently sequential.

In this work, we propose the notion of *Oblivious Parallel RAM (OPRAM)*. We present the first OPRAM compiler, converting any PRAM into an oblivious PRAM, while only inducing a

polylogarithmic slowdown to both the total *and parallel* complexities of the program.

Theorem 1.1 (OPRAM – Informally stated)**.** *There exists an OPRAM compiler with $O(\log(m)\log^3(n))$ worst-case overhead in total and parallel computation, and $f(n)$ memory overhead for any $f \in \omega(1)$, where n is the memory size and m is an upper-bound on the number of CPUs in the PRAM.*

We emphasize that applying even the most highly optimized ORAM compiler to an m-processor PRAM program inherently inflicts $\Omega(m \log(n))$ overhead in the parallel runtime, in comparison to our $O(\log(m)\mathsf{polylog}(n))$. When restricted to single-CPU programs, our construction incurs slightly greater logarithmic overhead than the best optimized ORAM compilers (achieving $O(\log n)$ overhead for optimal block sizes); we leave as an interesting open question how to optimize parameters. (As we will elaborate on shortly, some very interesting results towards addressing this has been obtained in the follow-up work of [CLT15].)

1.1 Applications of OPRAM

ORAM lies at the base of a wide range of applications. In many cases, we can *directly* replace the underlying ORAM with an OPRAM to enable *parallelism* within the corresponding secure application. For others, simply replacing ORAM with OPRAM does not suffice; nevertheless, in this paper, we demontrate one application (garbling of PRAM programs) where they can be overcome; follow-up works show further applications (secure computation and obfuscation).

Direct Applications of OPRAM We briefly describe some direct applications of OPRAM.

Improved/Parallelized Outsourced Data. Standard ORAM has been shown to yield effective, practical solutions for securely outsourcing data storage to an untrusted server (e.g., the ObliviStore system of [SS13]). Efficient OPRAM compilers will enable these systems to support secure efficient *parallel* accesses to outsourced data. For example, OPRAM procedures securely aggregate parallel data requests and resolve conflicts client-side, minimizing expensive client-server communications (as was explored in [WST12], at a smaller scale). As network latency is a major bottleneck in ORAM implementations, such parallelization may yield significant improvements in efficiency.

Multi-Client Outsourced Data. In a similar vein, use of OPRAM further enables secure access and manipulation of outsourced shared data by multiple (mutually trusting) clients. Here, each client can simply act as an independent CPU, and will execute the OPRAM-compiled program corresponding to the parallel concatenation of their independent tasks.

Secure Multi-Processor Architecture. Much recent work has gone toward implementing secure hardware architectures by using ORAM to prevent information leakage via access patterns of the secure processor to the potentially insecure memory (e.g., the Ascend project of [FDD12]). Relying instead on OPRAM opens the door to achieving secure hardware in the multi-processor setting.

Garbled PRAM (GPRAM) Garbled circuits [Yao82] allow a user to convert a circuit C and input x into garbled versions $\tilde{C}$ and $\tilde{x}$, in such a way that $\tilde{C}$ can be evaluated on $\tilde{x}$ to reveal the output $C(x)$, but without revealing further information on C or x. Garbling schemes have found countless applications in cryptography, ranging from delegation of computation to secure multi-party protocols (see below). It was recently shown (using ORAM) how to directly garble RAM

programs [GHL+14, GLOS15], where the cost of evaluating a garbled program $\tilde{P}$ scales with its RAM (and not circuit) complexity.

In this paper, we show how to employ any OPRAM compiler to attain a *garbled PRAM (GPRAM)*, where the time to generate and evaluate the garbled PRAM program $\tilde{P}$ scales with the *parallel* time complexity of P. Our construction is based on one of the construction of [GHL+14] and extends it using some of the techniques developed for our OPRAM. Plugging in our (unconditional) OPRAM construction, we obtain:

Theorem 1.2 (Garbled PRAM – Informally stated)**.** *Assuming identity-based encryption, there exists a secure garbled PRAM scheme with total* and parallel *overhead* $\mathsf{poly}(\kappa) \cdot \mathsf{polylog}(n)$*, where* κ *is the security parameter of the IBE and* n *is the size of the garbled data.*

Secure Two-Party and Multi-Party Computation of PRAMs. Secure multi-party computation (MPC) enables mutually distrusting parties to jointly evaluate functions on their secret inputs, without revealing information on the inputs beyond the desired function output. ORAM has become a central tool in achieving efficient MPC protocols for securely evaluating RAM programs. By instead relying on OPRAM, these protocols can leverage parallelizability of the evaluated programs.

Our garbled PRAM construction mentioned above yields constant-round secure protocols where the time to execute the protocol scales with the parallel time of the program being evaluated. In a companion paper [BCP15], we further demonstrates how to use OPRAM to obtain efficient protocols for securely evaluating PRAMs in the multi-party setting; see [BCP15] for further details.

Obfuscation for PRAMs. In a follow-up work, Chung et al [CCC+15] rely on our specific OPRAM construction (and show that it satisfies an additional "puncturability" property) to achieve obfuscation for PRAMs.

1.2 Technical Overview

Begin by considering the simplest idea toward memory obliviousness: Suppose data is stored in random(-looking) shuffled order, and for each data query i, the lookup is performed to its *permuted* location, $\sigma(i)$. One can see this provides some level of hiding, but clearly does not suffice for general programs. The problem with the simple solution is in *correlated lookups* over time—as soon as item i is queried again, this collision will be directly revealed. Indeed, hiding correlated lookups while maintaining efficiency is perhaps the core challenge in building oblivious RAMs. In order to bypass this problem, ORAM compilers heavily depend on the ability of the CPU to *move data around*, and to *update its secret state* after each memory access.

However, in the parallel setting, we find ourselves back at square one. Suppose in some time step, a group of processors all wish to access data item i. Having all processors attempt to perform the lookup directly within a standard ORAM construction corresponds to running the ORAM several times *without moving data or updating state*. This immediately breaks security in all existing ORAM compiler constructions. On the other hand, we cannot afford for the CPUs to "take turns," accessing and updating the data sequentially.

In this overview, we discuss our techniques for overcoming this and further challenges. We describe our solution somewhat abstractly, building on a sequential ORAM compiler with a tree-based structure as introduced by Shi *et al.* [SCSL11]. In our formal construction and analysis, we

rely on the specific tree-based ORAM compiler of Chung and Pass [CP13] that enjoys a particularly clean description and analysis.

Tree-Based ORAM Compilers. We begin by roughly describing the structure of tree-based ORAMs, originating in the work of [SCSL11]. At a high level, data is stored in the structure of a binary tree, where each node of the tree corresponds to a fixed-size bucket that may hold a collection of data items. Each memory cell addr in the original database is associated with a random *path* (equivalently, leaf) within a binary tree, as specified by a position map $\mathsf{path}_{\mathsf{addr}} = Pos(\mathsf{addr})$.

The schemes maintain three invariants: (1) The content of memory cell addr will be found in one of the buckets *along the path* $\mathsf{path}_{\mathsf{addr}}$. (2) Given the view of the adversary (i.e., memory accesses) up to any point in time, the current mapping Pos appears uniformly random. And, (3) with overwhelming probability, no node in the binary tree will ever "overflow," in the sense that its corresponding memory bucket is instructed to store more items than its fixed capacity.

These invariants are maintained by the following general steps:

1. Lookup: To access a memory item addr, the CPU accesses all buckets down the path $\mathsf{path}_{\mathsf{addr}}$, and removes it where found.
2. Data "put-back": At the conclusion of the access, the memory item addr is assigned a *freshly random* path $Pos(\mathsf{addr}) \leftarrow \mathsf{path}'_{\mathsf{addr}}$, and is returned to the *root node* of the tree.
3. Data flush: To ensure the root (and any other bucket) does not overflow, data is "flushed" down the tree via some procedure. For example, in [SCSL11], the flush takes place by selecting and emptying two random buckets from each level into their appropriate children; in [CP13], it takes place by choosing an independent path in the tree and pushing data items down this path as far as they will go (see Figure 1 in Section 2.2).

Extending to Parallel RAMs. We must address the following problems with attempting to access a tree-based ORAM in *parallel*.

- **Parallel memory lookups:** As discussed, a core challenge is in hiding correlations in parallel CPU accesses. In tree-based ORAMs, if CPUs access different data items in a time step, they will access different paths in the tree, whereas if they attempt to simultaneously access the same data item, they will each access the same path in the tree, blatantly revealing a collision.

  To solve this problem, before each lookup we insert a *CPU-coordination* phase. We observe that in tree-based ORAM schemes, this problem only manifests when CPUs access *exactly* the same item, otherwise items are associated with independent leaf nodes, and there are no bad correlations. We thus resolve this issue by letting the CPUs check—through an oblivious aggregation operation—whether two (or more) of them wish to access the same data item; if so, a representative is selected (the CPU with the smallest id) to actually perform the memory access, and all the others merely perform "dummy" lookups. Finally, the representative CPU needs to communicate the read value back to all the other CPUs that wanted to access the same data item; this is done using an oblivious multi-cast operation.

  The challenge is in doing so without introducing too much overhead—namely, allowing only (per-CPU) memory, computation, and parallel time *polylogarithmic* in both the database size and the number of CPUs—and that itself retains memory obliviousness.
- **Parallel "put-backs":** After a memory cell is accessed, the (possibly updated) data is assigned a fresh random path and is reinserted to the tree structure. To maintain the required

invariants, the item must be inserted somewhere along its new path, *without revealing* any information about the path. In tree-based ORAMs, this is done by reinserting at the root node of the tree. However, this single node can hold only a small bounded number of elements (corresponding to the fixed bucket size), whereas the number of processors m—each with an item to reinsert—may be significantly larger.

To overcome this problem, instead of returning data items to the root, we directly insert them into level $\log m$ of the tree, while ensuring that they are placed into the correct bucket along their assigned path. Note that level $\log m$ contains m buckets, and since the m items are each assigned to random leaves, each bucket will in expectation be assigned exactly 1 item.

The challenge in this step is specifying how the m CPUs can insert elements into the tree while maintaining *memory obliviousness*. For example, if each CPU simply inserts their own item into its assigned node, we immediately leak information about its destination leaf node. To resolve this issue, we have the CPUs obliviously route items between each other, so that eventually the ith CPU holds the items to be insert to the ith node, and all CPUs finally perform either a real or a dummy write to their corresponding node.

- **Preventing overflows:** To ensure that no new overflows are introduced after inserting m items, we now flush m times instead of once, and all these m flushes are done in parallel: each CPU simply performs an independent flush. These parallel flushes may lead to conflicts in nodes accessed (e.g., each flush operation will likely access the root node). As before, we resolve this issue by having the CPUs elect some representative to perform the appropriate operations for each accessed node; note, however, that this step is required only for correctness, and not for security.

Our construction takes a modular approach. We first specify and analyze our compiler within a simplified setting, where oblivious communication between CPUs is "for free." We then show how to efficiently instantiate the required CPU communication procedures oblivious routing, oblivious aggregation, and oblivious multi-cast, and describe the final compiler making use of these procedures. In this extended abstract, we defer the first step to Appendix 3.1, and focus on the remaining steps.

1.3 Related Work

Restricted cases of parallelism in Oblivious RAM have appeared in a handful of prior works. It was observed by Williams, Sion, and Tomescu [WST12] in their PrivateFS work that existing ORAM compilers can support parallelization across data accesses up to the "size of the top level,"[1] (in particular, at most $\log n$), when coordinated through a central trusted entity. We remark that central coordination is not available in the PRAM model. Goodrich and Mitzenmacher [GM11] showed that parallel programs in MapReduce format can be made oblivious by simply replacing the "shuffle" phase (in which data items with a given key are routed to the corresponding CPU) with a fixed-topology sorting network. The goal of improving the parallel overhead of ORAM was studied by Lorch *et al.* [LPM+13], but does not support compilation of PRAMs without first sequentializing.

Follow-up work. As mentioned above, our OPRAM compiler has been used in the recent works of Boyle, Chung, and Pass [BCP15] and Chen *et al.* [CCC+15] to obtain secure multi-party computation for PRAM, and indistinguishability obfuscation for PRAM, respectively. A different follow-up

[1] E.g., for tree-based ORAMs, the size of the root bucket.

work by Nayak *et al.* [NWI+15] provides targeted optimizations and an implementation for secure computation of specific parallel tasks.

Very recently, an exciting follow-up work of Chen, Lin, and Tessaro [CLT15] builds upon our techniques to obtain two new construction: an OPRAM compiler whose overhead in expectation matches that of the best current sequential ORAM [SvDS+13]; and, a general transformation taking *any* generic ORAM compiler to an OPRAM compiler with $\log n$ overhead in expectation. Their OPRAM constructions, however, only apply to the special case of PRAM with a *fixed* number of processors being activated at every step (whereas our notion of a PRAM requires handling also a variable number of processors[2]); for the case of variable CPU PRAMs, the results of [CLT15] incurr an additional multlicative overhead of m in terms of computational complexity, and thus the bounds obtained are incomparable.

2 Preliminaries

2.1 Parallel RAM (PRAM) Programs

We consider the most general case of Concurrent Read Concurrent Write (CRCW) PRAMs. An m-processor CRCW *parallel random-access machine (PRAM)* with memory size n consists of numbered processors $CPU_1, \ldots, CPU_m$, each with local memory registers of size $\log n$, which operate synchronously in parallel and can make access to shared "external" memory of size n.

A PRAM program Π (given m, n, and some input x stored in shared memory) provides CPU-specific execution instructions, which can access the shared data via commands $\mathsf{Access}(r, v)$, where $r \in [n]$ is an index to a memory location, and v is a word (of size $\log n$) or $\perp$. Each $\mathsf{Access}(r, v)$ instruction is executed as:

1. **Read** from shared memory cell address r; denote value by v_{old}.
2. **Write** value $v \neq \perp$ to address r (if $v = \perp$, then take no action).
3. **Return** v_{old}.

In the case that two or more processors simultaneously initiate $\mathsf{Access}(r, v_i)$ with the same address r, then all requesting processors receive the previously existing memory value v_{old}, and the memory is rewritten with the value v_i corresponding to the lowest-numbered CPU i for which $v_i \neq \perp$.

We more generally support PRAM programs with a dynamic number of processors (i.e., m_i processors required for each time step i of the computation), as long as this sequence of processor numbers $m_1, m_2, \ldots$ is public information. The complexity of our OPRAM solution will scale with the number of required processors in each round, instead of the maximum number of required processors.

The *(parallel) time complexity* of a PRAM program Π is the maximum number of time steps taken by any processor to evaluate Π, where each Access execution is charged as a single step. The PRAM complexity of a function f is defined as the minimal parallel time complexity of any PRAM program which evaluates f. We remark that the PRAM complexity of any function f is bounded above by its circuit depth complexity.

Remark 2.1 (CPU-to-CPU Communication)**.** It will be sometimes convenient notationally to assume that CPUs may communicate directly amongst themselves. When the identities of sending

[2]As previously mentioned, dealing with a variable number of processors is needed to capture standard circuit models of computation, where the circuit topology may be of varying width.

and receiving CPUs is known a priori (which will always be the case in our constructions), such communication can be emulated in the standard PRAM model with constant overhead by communicating *through memory*. That is, each action "CPU1 sends message m to CPU2" is implemented in two time steps: First, CPU1 writes m into a special designated memory location addr_{CPU1}; in the following time step, CPU2 performs a read access to addr_{CPU1} to learn the value m.

2.2 Tree-Based ORAM

Concretely, our solution relies on the ORAM due to Chung and Pass [CP13], which in turn closely follows the tree-based ORAM construction of Shi *et al.* [SCSL11]. We now recall the [CP13] construction in greater detail, in order to introduce notation for the remainder of the paper.

The [CP13] construction (as with [SCSL11]) proceeds by first presenting an intermediate solution achieving obliviousness, but in which the CPU must maintain a large number of registers (specifically, providing a means for securely storing n data items requiring CPU state size $\tilde{\Theta}(n/\alpha)$, where $\alpha > 1$ is any constant). Then, this solution is recursively applied $\log_\alpha n$ times to store the resulting CPU state, until finally reaching a CPU state size $\mathsf{polylog}(n)$, while only blowing up the computational overhead by a factor $\log_\alpha n$. The overall compiler is fully specified by describing one level of this recursion.

Step 1: Basic ORAM with $O(n)$ registers. The compiler $ORAM$ on input $n \in \mathbb{N}$ and a program Π with memory size n outputs a program Π' that is identical to Π but each $\mathsf{Read}(r)$ or $\mathsf{Write}(r, val)$ is replaced by corresponding commands $\mathsf{ORead}(r)$, $\mathsf{OWrite}(r, val)$ to be specified shortly. Π' has the same registers as Π and additionally has n/α registers used to store a *position map* Pos plus a polylogarithmic number of additional *work* registers used by ORead and OWrite. In its external memory, Π' will maintain a complete binary tree Γ of depth $\ell = \log(n/\alpha)$; we index nodes in the tree by a binary string of length at most ℓ, where the root is indexed by the empty string λ, and each node indexed by γ has left and right children indexed $\gamma 0$ and $\gamma 1$, respectively. Each memory cell r will be associated with a random leaf *pos* in the tree, specified by the position map Pos; as we shall see shortly, the memory cell r will be stored at one of the nodes on the path from the root λ to the leaf *pos*. To ensure that the position map is smaller than the memory size, we assign a *block* of α consecutive memory cells to the same leaf; thus memory cell r corresponding to block $b = \lfloor r/\alpha \rfloor$ will be associated with leaf $pos = \mathsf{Pos}(b)$.

Each node in the tree is associated with a *bucket* which stores (at most) K tuples (b, pos, v), where v is the content of block b and *pos* is the leaf associated with the block b, and $K \in \omega(\log n) \cap \mathsf{polylog}(n)$ is a parameter that will determine the security of the ORAM (thus each bucket stores $K(\alpha + 2)$ words). We assume that all registers and memory cells are initialized with a special symbol $\perp$.

The following is a specification of the $\mathsf{ORead}(r)$ procedure:

Fetch: Let $b = \lfloor r/\alpha \rfloor$ be the block containing memory cell r (in the original database), and let $i = r \mod \alpha$ be r's component within the block b. We first look up the position of the block b using the position map: $pos = \mathsf{Pos}(b)$; if $\mathsf{Pos}(b) = \perp$, set $pos \leftarrow [n/\alpha]$ to be a uniformly random leaf.

Next, traverse the data tree from the root to the leaf *pos*, making exactly one read and one write operation for the memory bucket associated with each of the nodes along the path. More precisely, we read the content once, and then we either write it back (unchanged), or we

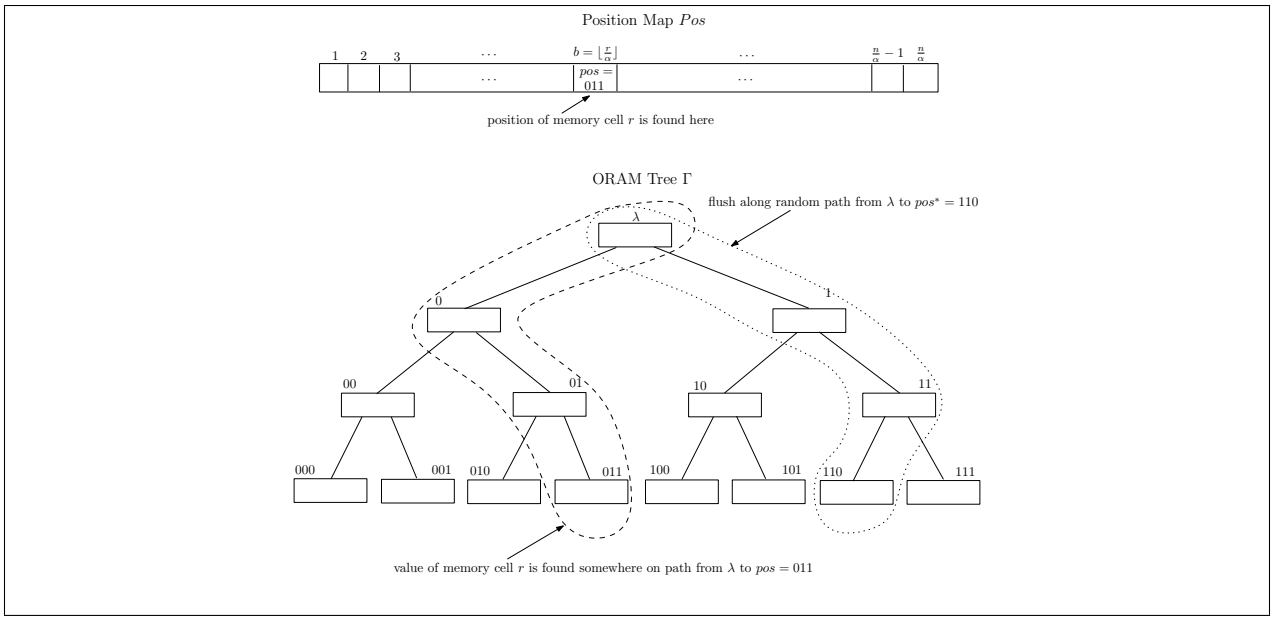


Figure 1: Illustration of the basic [CP13] ORAM construction.

simply "erase it" (writing $\perp$) so as to implement the following task: search for a tuple of the form (b, pos, v) for the desired b, pos in any of the nodes during the traversal; if such a tuple is found, remove it from its place in the tree and set v to the found value, and otherwise take $v = \perp$. Finally, return the ith component of v as the output of the $\mathsf{ORead}(r)$ operation.

Update Position Map: Pick a uniformly random leak $pos' \leftarrow [n/\alpha]$ and let $\mathsf{Pos}(b) = pos'$.

Put Back: Add the tuple (b, pos', v) to the root λ of the tree. If there is not enough space left in the bucket, abort outputting overflow.

Flush: Pick a uniformly random leaf $pos^* \leftarrow [n/\alpha]$ and traverse the tree from the roof to the leaf pos^*, making exactly one read and one write operation for every memory cell associated with the nodes along the path so as to implement the following task: "push down" each tuple (b'', pos'', v'') read in the nodes traversed so far as possible along the path to pos^* while ensuring that the tuple is still on the path to its associated leaf pos'' (that is, the tuple ends up in the node $\gamma =$ longest common prefix of pos'' and pos^*.) Note that this operation can be performed trivially as long as the CPU has sufficiently many work registers to load two whole buckets into memory; since the bucket size is polylogarithmic, this is possible. If at any point some bucket is about to overflow, abort outputting overflow.

$\mathsf{OWrite}(r, v)$ proceeds identically in the same steps as $\mathsf{ORead}(r)$, except that in the "Put Back" steps, we add the tuple (b, pos', v'), where v' is the string v but the ith component is set to v (instead of adding the tuple (b, pos', v) as in ORead). (Note that, just as ORead, OWrite also outputs the ordinal memory content of the memory cell r; this feature will be useful in the "full-fledged" construction.)

The full-fledged construction: ORAM with polylog registers. The full-fledged construction of the CP ORAM proceeds as above, except that instead of storing the position map in registers in the CPU, we now recursively store them in another ORAM (which only needs to operate on n/α

memory cells, but still using buckets that store K tuples). Recall that each invocation of ORead and OWrite requires reading one position in the position map and updating its value to a random leaf; that is, we need to perform a *single* recursive OWrite call (recall that OWrite updates the value in a memory cell, and returns the old value) to emulate the position map.

At the base of the recursion, when the position map is of constant size, we use the trivial ORAM construction which simply stores the position map in the CPU registers.

Theorem 2.2 ([CP13])**.** *The compiler ORAM described above is a secure Oblivious RAM compiler with* $\mathsf{polylog}(n)$ *worst-case computation overhead and* $\omega(\log n)$ *memory overhead, where* n *is the database memory size.*

2.3 Sorting Networks

Our protocol will employ an *n-wire sorting network*, which can be used to sort values on n wires via a fixed topology of comparisons. A sorting network consists of a sequence of *layers*, each layer in turn consisting of one or more comparator gates, which take two wires as input, and swap the values when in unsorted order. Formally, given input values $\vec{x} = (x_1, \ldots, x_n)$ (which we assume to be integers wlog), a comparator operation $\mathsf{compare}(i, j, \vec{x})$ for $i < j$ returns $\vec{x'}$ where $\vec{x} = \vec{x'}$ if $x_i \leq x_j$, and otherwise, swaps these values as $x'_i = x_j$ and $x'_j = x_i$ (whereas $x'_k = x_k$ for all $k \neq i, j$). Formally, a layer in the sorting network is a set $L = \{(i_1, j_1), \ldots, (i_k, j_k)\}$ of pairwise-disjoint pairs of distinct indices of $[n]$. A d-depth sorting network is a list $SN = (L_1, \ldots, L_d)$ of layers, with the property that for any input vector $\vec{x}$, the final output will be in sorted order $x_i \leq x_{i+1}\ \forall i < n$.

Ajtai, Komlós, and Szemerédi demonstrated a sorting network with depth logarithmic in n.

Theorem 2.3 ([AKS83])**.** *There exists an n-wire sorting network of depth* $O(\log n)$ *and size* $O(n \log n)$*.*

While the AKS sorting network is asymptotically optimal, in practical scenarios one may wish to use the simpler alternative construction due to Batcher [Bat68] which achieves significantly smaller linear constants.

3 Oblivious PRAM

The definition of an Oblivious PRAM (OPRAM) compiler mirrors that of standard ORAM, with the exception that the compiler takes as input and produces as output a *parallel* RAM program. Namely, denote the sequence of shared memory cell accesses made during an execution of a PRAM program Π on input (m, n, x) as $\tilde{\Pi}(m, n, x)$. And, denote by $\mathsf{ActivationPatterns}(\Pi, m, n., x)$ the (public) CPU activation patterns (i.e., number of active CPUs per timestep) of program Π on input (m, n, x). We present a definition of an OPRAM compiler following Chung and Pass [CP13], which in turn follows Goldreich [Gol87].

Definition 3.1 (Oblivious Parallel RAM)**.** A polynomial-time algorithm O is an *Oblivious Parallel RAM (OPRAM)* compiler with computational overhead $\mathsf{comp}(\cdot, \cdot)$ and memory overhead $\mathsf{mem}(\cdot, \cdot)$, if O given $m, n \in \mathbb{N}$ and a deterministic m-processor PRAM program Π with memory size n, outputs an m-processor program Π' with memory size $\mathsf{mem}(m, n) \cdot n$ such that for any input x, the parallel running time of $\Pi'(m, n, x)$ is bounded by $\mathsf{comp}(m, n) \cdot T$, where T is the parallel runtime of $\Pi(m, n, x)$, and there exists a negligible function μ such that the following properties hold:

- **Correctness:** For any $m, n \in \mathbb{N}$ and any string $x \in \{0,1\}^*$, with probability at least $1 - \mu(n)$, it holds that $\Pi(m, n, x) = \Pi'(m, n, x)$.
- **Obliviousness:** For any two PRAM programs Π_1, Π_2, any $m, n \in \mathbb{N}$, and any two inputs $x_1, x_2 \in \{0,1\}^*$, if $|\Pi_1(m, n, x_1)| = |\Pi_2(m, n, x_2)|$ and $\mathsf{ActivationPatterns}(\Pi_1, m, n, x_1)) = \mathsf{ActivationPatterns}(\Pi_2, m, n, x_2)$, then $\tilde{\Pi}'_1(m, n, x_1)$ is μ-close to $\tilde{\Pi}'_2(m, n, x_2)$ in statistical distance, where $\Pi'_i \leftarrow O(m, n, \Pi_i)$ for $i \in \{1, 2\}$.

We remark that not all m processors may be active in every time step of a PRAM program Π, and thus its total computation cost may be significantly less than $m \cdot T$. We wish to consider OPRAM compilers that also preserve the processor activation structure (and thus total computation complexity) of the original program up to polylogarithmic overhead. Of course, we cannot hope to do so if the processor activation patterns themselves reveal information about the secret data. We thus consider PRAMs Π whose activation schedules $(m_1, \ldots, m_T)$ are a-priori fixed and public.

Definition 3.2 (Activation-Preserving)**.** An OPRAM compiler O with computation overhead $\mathsf{comp}(\cdot, \cdot)$ is said to be *activation preserving* if given $m, n \in \mathbb{N}$ and a deterministic PRAM program Π with memory size n and fixed (public) activation schedule $(m_1, \ldots, m_T)$ for $m_i \leq m$, the program Π' output by O has activation schedule $\left((m_1)_{i=1}^t, (m_2)_{i=1}^t, \ldots, (m_T)_{i=1}^t\right)$, where $t = \mathsf{comp}(m, n)$.

It will additionally be useful in applications (e.g., our construction of garbled PRAMs in Section 4, and the MPC for PRAMs of [BCP15]) that the resulting oblivious PRAM is *collision free*.

Definition 3.3 (Collision-Free)**.** An OPRAM compiler O is said to be *collision free* if given $m, n \in \mathbb{N}$ and a deterministic PRAM program Π with memory size n, the program Π' output by O has the property that no two processors ever access the same data address in the same timestep.

We now present our main result, which we construct and prove in the following subsections.

Theorem 3.4 (Main Theorem: OPRAM)**.** *There exists an activation-preserving, collision-free OPRAM compiler with $O(\log(m)\log^3(n))$ worst-case computational overhead and $f(n)$ memory overhead, for any $f \in \omega(1)$, where n is the memory size and m is the number of CPUs.*

3.1 Rudimentary Solution: Requiring Large Bandwidth

We first provide a solution for a simplified case, where we are not concerned with minimizing communication between CPUs or the size of required CPU local memory. In such setting, communicating and aggregating information between all CPUs is "for free."

Our compiler Heavy-O, on input $m, n \in \mathbb{N}$, fixed integer constant $\alpha > 1$, and m-processor PRAM program Π with memory size n, outputs a program Π' identical to Π, but with each $\mathsf{Access}(r, v)$ operation replaced by the modified procedure Heavy-OPAccess as defined in Figure 2. (Here, "broadcast" means to send the specified message to all other processors).

Note that Heavy-OPAccess operates recursively for $t = 0, \ldots, \lceil \log_\alpha n \rceil$. This corresponds analogously to the recursion in the [SCSL11, CP13] ORAM, where in each step the size of the required "secure database memory" drops by a constant factor α. We additionally utilize a space optimization due to Gentry *et al.* [GGH+13] that applies to [CP13], where the ORAM tree used for storing data of size n' has depth $\log n'/K$ (and thus n'/K leaves instead of n'), where K is the bucket size. This enables the overall memory overhead to drop from $\omega(\log n)$ (i.e., K) to $\omega(1)$ with minimal changes to the analysis.

Heavy-OPAccess$(t, (r_i, v_i))$: The Large Bandwidth Case

To be executed by $CPU_1, \ldots, CPU_m$ w.r.t. (recursive) database size $n_t := n/(\alpha^t)$, bucket size K.

Input: Each CPU_i holds: recursion level t, instruction pair (r_i, v_i) with $r_i \in [n_t]$, global parameter α.

Each CPU_i performs the following steps, in parallel

0. Exit Case: If $t \geq \log_\alpha n$, return 0.
   This corresponds to requesting the (trivial) position map for a block within a single-leaf tree.
1. Conflict Resolution
   (a) Broadcast the instruction pair (r_i, v_i) to all CPUs.
   (b) Let $b_i = \lfloor r_i/\alpha \rfloor$. Locally aggregate incoming instructions to block b_i as $\bar{v}_i = \bar{v}_i[1] \cdots \bar{v}_i[\alpha]$, resolving write conflicts (i.e., $\forall s \in [\alpha]$, take $\bar{v}_i[s] \leftarrow v_j$ for minimal j such that $r_j = b_i\alpha + s$). Denote by $\mathsf{rep}(b_i) := \min\{j : \lfloor r_j/\alpha \rfloor = b_i\}$ the smallest index j of *any* CPU whose r_j is in this block b_i. (CPU $\mathsf{rep}(b_i)$ will actually access b_i, while others perform dummy accesses).
2. Recursive Access to Position Map (Define $L_t := 2n_t/K$, number of leaves in t'th tree).
   If $i = \mathsf{rep}(b_i)$: Sample fresh leaf id $\ell'_i \leftarrow [L_t]$. Recurse as $\ell_i \leftarrow$ Heavy-OPAccess$(t+1, (b_i, \ell'_i))$ to read the current value ℓ_i of $\mathsf{Pos}(b_i)$ and rewrite it with ℓ'_i.
   Else: Recursively initiate *dummy* access $x \leftarrow$ Heavy-OPAccess$(t+1, (1, \perp))$ at arbitrary address (say 1); ignore the read value x. Sample fresh random leaf id $\ell_i \leftarrow [L_t]$ for a dummy lookup.
3. Look Up Current Memory Values
   Read the memory contents of all buckets down the path to leaf node ℓ_i defined in the previous step, copying all buckets into local memory.
   If $i = \mathsf{rep}(b_i)$: locate and store target block triple (b_i, v_i^{old}, ℓ_i). Update $\bar{v}$ from Step 1 with existing data: $\forall s \in [\alpha]$, replace any non-written cell values $\bar{v}_i[s] = \emptyset$ with $\bar{v}_i[s] \leftarrow v_i^{old}[s]$. $\bar{v}_i$ now stores the entire data block to be rewritten for block b_i.
4. Remove Old Data from ORAM Database
   (a) If $i = \mathsf{rep}(b_i)$: Broadcast (b_i, ℓ_i) to all CPUs. Otherwise: broadcast $(\perp, \ell_i)$.
   (b) Initiate UpdateBuckets $\left(n_t, (\mathsf{remove}\text{-}b_i, \ell_i), \{(\mathsf{remove}\text{-}b_j, \ell_j)\}_{j \in [m] \setminus \{i\}}\right)$, as in Figure 3.
5. Insert New Data into Database *in Parallel*
   (a) If $i = \mathsf{rep}(b_i)$: Broadcast $(b_i, \bar{v}_i, \ell'_i)$, with updated value $\bar{v}_i$ and target leaf ℓ'_i.
   (b) Let $\mathsf{lev}^* := \lfloor \log(\min\{m, L_t\}) \rfloor$ be the ORAM tree level with number of buckets equal to number of CPUs (the level where data will be inserted). Locally aggregate all incoming instructions whose path ℓ'_j has lev^*-bit prefix i: $\mathsf{Insert}_i := \{(b_j, \bar{v}_j, \ell'_j) : (\ell'_j)^{(\mathsf{lev}^*)} = i\}$.
   (c) Access memory bucket i (at level lev^*) and rewrite contents, inserting data items Insert_i. If bucket i exceeds its capacity, abort with overflow.
6. Flush the ORAM Database
   (a) Sample a random leaf node $\ell_i^{\mathsf{flush}} \leftarrow [L_t]$ along which to flush. Broadcast ℓ_i^{flush}.
   (b) If $i \leq L_t$: Initiate UpdateBuckets $\left(n_t, (\mathsf{flush}, \ell_i^{\mathsf{flush}}), \{(\mathsf{flush}, \ell_j^{\mathsf{flush}})\}_{j \in [m] \setminus \{i\}}\right)$, in Figure 3.
   Recall that flush means to "push" each encountered triple (b, ℓ, v) down to the lowest point at which his chosen flush path and ℓ agree.
7. Update CPUs
   If $i = \mathsf{rep}(b_i)$: broadcast the *old* value v_i^{old} of block b_i to all CPUs.

Figure 2: Pseudocode for oblivious parallel data access procedure Heavy-OPAccess (where we are temporarily not concerned with per-round bandwidth/memory).

$\mathsf{UpdateBuckets}\left(n_t, (\mathsf{mycommand}, \mathsf{mypath}), \{(\mathsf{command}_j, \mathsf{path}_j)\}_{j\in[m]\setminus\{i\}}\right)$
Let $\mathsf{path}^{(0)}, \ldots, \mathsf{path}^{(\log L_t)}$ denote the bit prefixes of length 0 (i.e., $\emptyset$) to $\log(L_t)$ of path.
For each tree level $\mathsf{lev} = 0$ to $\log L_t$, each CPU i does the following at bucket $\mathsf{mypath}^{(\mathsf{lev})}$:

1. Define $\mathsf{CPUs}(\mathsf{mypath}^{(\mathsf{lev})}) := \{i\} \cup \{j : \mathsf{path}_j^{(\mathsf{lev})} = \mathsf{mypath}^{(\mathsf{lev})}\}$ to be the set of CPUs requesting changes to bucket $\mathsf{mypath}^{(\mathsf{lev})}$. Let $\mathsf{bucket\text{-}rep}(\mathsf{mypath}^{(\mathsf{lev})})$ denote the *minimal* index in the set.
2. If $i \neq \mathsf{bucket\text{-}rep}(\mathsf{mypath}^{(\mathsf{lev})})$, do nothing. Otherwise:

   Case 1: $\mathsf{mycommand} = \mathsf{remove}\text{-}b_i$.
   Interpret each $\mathsf{command}_j = \mathsf{remove}\text{-}b_j$ as a target block id b_j to be removed. Access memory bucket $\mathsf{mypath}^{(\mathsf{lev})}$ and rewrite contents, removing any block b_j for which $j \in \mathsf{CPUs}(\mathsf{mypath}^{(\mathsf{lev})})$.

   Case 2: $\mathsf{mycommand} = \mathsf{flush}$.
   Define $\mathsf{Flush} \subset \{L, R\}$ as $\{v : \exists\ \mathsf{path}_j \text{ s.t. } \mathsf{path}_j^{(\mathsf{lev}+1)} = \mathsf{mypath}^{(\mathsf{lev})}||v\}$, associating $L \equiv 0$, $R \equiv 1$. This determines whether data will be flushed left and/or right from this bucket. Access memory bucket $\mathsf{mypath}^{(\mathsf{lev})}$; denote its collection of stored data blocks b by $\mathsf{ThisBucket}$. Partition $\mathsf{ThisBucket} = \mathsf{ThisBucket\text{-}L} \cup \mathsf{ThisBucket\text{-}R}$ into those blocks whose associated leaves continue to the left or right (i.e., $\mathsf{ThisBucket\text{-}L} := \{b_j \in \mathsf{ThisBucket} : \bar{\ell}_j^{(\mathsf{lev}+1)} = \mathsf{mypath}^{(\mathsf{lev})}||0\}$, and similar for 1).
   - If $L \in \mathsf{Flush}$, then set $\mathsf{ThisBucket} \leftarrow \mathsf{ThisBucket} \setminus \mathsf{ThisBucket\text{-}L}$, access memory bucket $\mathsf{mypath}^{(\mathsf{lev})}||0$, and insert data items $\mathsf{ThisBucket\text{-}L}$ into it.
   - If $R \in \mathsf{Flush}$, then set $\mathsf{ThisBucket} \leftarrow \mathsf{ThisBucket} \setminus \mathsf{ThisBucket\text{-}R}$, access memory bucket $\mathsf{mypath}^{(\mathsf{lev})}||1$, and insert data items $\mathsf{ThisBucket\text{-}R}$ into it.

   Rewrite the contents of bucket $\mathsf{mypath}^{(\mathsf{lev})}$ with updated value of $\mathsf{ThisBucket}$. If any bucket exceeds its capacity, abort with $\mathsf{overflow}$.

Figure 3: Procedure for combining CPUs' instructions for buckets and implementing them by a single representative CPU. (Used for correctness, not security). See Figure 4 for a sample illustration.

Lemma 3.5. *For any $n, m \in \mathbb{N}$, The compiler* Heavy-O *is a secure Oblivious PRAM compiler with parallel time overhead $O(\log^3 n)$ and memory overhead $\omega(1)$, assuming each CPU has $\tilde{\Omega}(m)$ local memory.*

We will address the desired claims of correctness, security, and complexity of the Heavy-O compiler by induction on the number of levels of recursion. Namely, for $t^* \in [\log_\alpha n]$, denote by Heavy-O_{t^*} the compiler that acts on memory size $n/(\alpha^{t^*})$ by executing Heavy-O only on recursion levels $t = t^*, (t^*+1), \ldots, \lceil \log_\alpha n \rceil$. For each such t^*, we define the following property.

IH

Level-t^* Heavy OPRAM: We say that Heavy-O_{t^*} is a *valid level-t^* heavy OPRAM* if the partial-recursion compiler Heavy-O_{t^*} is a secure Oblivious PRAM compiler for memory size $n/(\alpha^{t^*})$ with parallel time overhead $O(\log^2 n \cdot \log(n/\alpha^{t^*}))$ and memory overhead $\omega(1)$, assuming each CPU has $\tilde{\Omega}(m)$ local memory.

Then Lemma 3.5 follows directly from the following two claims.

Claim 3.6. Heavy-$O_{\log_\alpha n}$ *is valid level-$(\log_\alpha n)$ heavy OPRAM.*

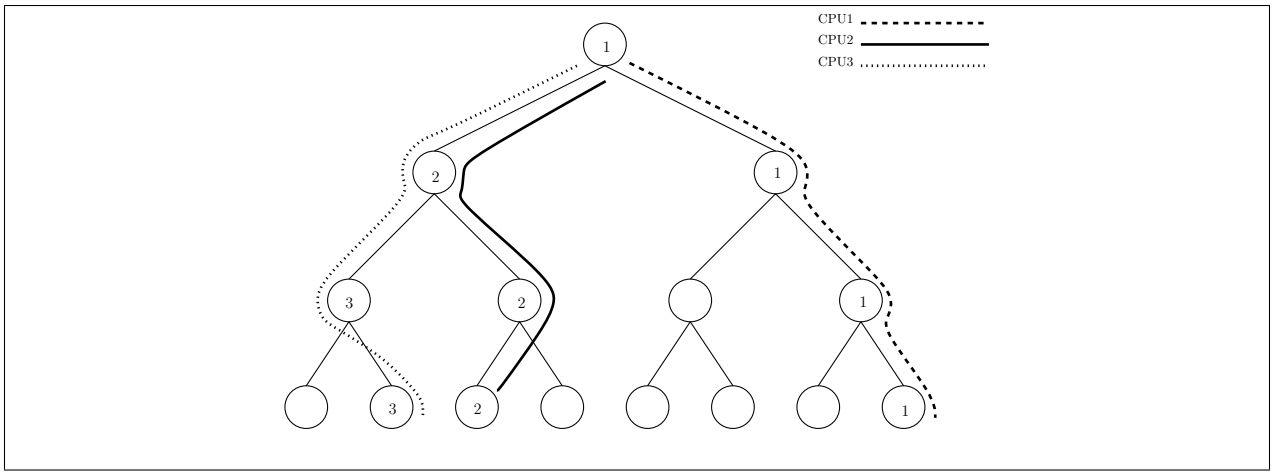


Figure 4: UpdateBuckets sample illustration. Here, CPUs 1-3 each wish to modify nodes along their paths as drawn; for each overlapping node, the CPU with lowest id receives and implements the aggregated commands for the node.

Proof. Note that Heavy-$O_{\log_\alpha n}$, acting on trivial size-1 memory, corresponds directly to the exit case (Step 0) of Heavy-OPAccess in Figure 2. Namely, correctness, security, and the required efficiency trivially hold, since there is a single data item in a fixed location to access. □

Claim 3.7. *Suppose* Heavy-O_t *is a valid level-*t *heavy OPRAM for* $t > 0$*. Then* Heavy-O_{t-1} *is a valid level-*$(t-1)$ *heavy OPRAM.*

Proof. We first analyze the correctness, security, and complexity overhead of Heavy-O_{t-1} conditioned on never reaching the event overflow (which may occur in Step 5(c), or within the call to UpdateBuckets). Then, we prove that the probability of overflow is negligible in n.

Correctness (w/o overflow). Consider the state of the memory (of the CPUs and server) in each step of Heavy-OPAccess, assuming no overflow. In Step 1, each CPU learns the instruction pairs of all other CPUs; thus all CPUs agree on single representative $\mathsf{rep}(b_i)$ for each requested block b_i, and a correct aggregation of all instructions to be performed on this block. Step 2 is a recursive execution of Heavy-OPAccess. By the inductive hypothesis, this access successfully returns the correct value ℓ_i of $\mathsf{Pos}(b_i)$ for each b_i queried, and rewrites it with the freshly sampled value ℓ_i' when specified (i.e., for each $\mathsf{rep}(b_i)$ access; the dummy accesses are read-only). We are thus guaranteed that each $\mathsf{rep}(b_i)$ will find the desired block b_i in Step 3 when accessing the memory buckets in the path down the tree to leaf ℓ_i (as we assume no overflow was encountered), and so will learn the current stored data value v_{old}.

In Step 4, each CPU learns the target block b_i and associated leaf ℓ_i of every representative CPU $\mathsf{rep}(b_i)$. By construction, each requested block b_i appears in some bucket B in the tree along his path, and there there will necessarily be some CPU assigned as $\mathsf{bucket\text{-}rep}(B)$ in UpdateBuckets, who will then successfully remove the block b_i from B. At this point, none of the requested blocks b_i appear in the tree.

In Step 5, the CPUs insert each block b_i (with updated data value v_i) into the ORAM data tree at level $\min\{\log_\alpha n/\alpha^t, \lfloor \log_2(m) \rfloor\}$ along the path to its (new) leaf ℓ_i'.

Finally, the flushing procedure in Step 6 maintains the necessary property that each block b_i appears along the path to $\mathsf{Pos}(b_i)$, and in Step 7 all CPUs learn the collection of all queried values v_{old} (in particular, including the value they initially requested).

Thus, assuming no overflow, correctness holds.

Obliviousness (w/o overflow). Consider the access patterns to server-side memory in each step of Heavy-OPAccess, assuming no overflow. Step 1 is performed locally without communication to the server. Step 2 is a recursive execution of Heavy-OPAccess, which thus yields access patterns independent of the vector of queried data locations (up to statistical distance negligible in n), by the induction hypothesis. In Step 3, each CPU accesses the buckets along a single path down the tree, where representative CPUs $\mathsf{rep}(b_i)$ access along the path given by $\mathsf{Pos}(b_i)$ (for *distinct* b_i), and non-representative CPUs each access down an independent, random path. Since the adversarial view so far has been independent of the values of $\mathsf{Pos}(b_i)$, conditioned on this view all CPU's paths are independent and random.

In Step 4, all data access patterns are publicly determinable based on the accesses in the previous step (that is, the complication in Step 4 is to ensure correctness without access collisions, but is not needed for security). In Step 5, each CPU i accesses his corresponding bucket i in the tree. In the flushing procedure of Step 6, each CPU selects an independent, random path down the tree, and the communication patterns to the server reveal no information beyond the identities of these paths. Finally, Step 7 is performed locally without communication to the server.

Thus, assuming no overflow, obliviousness holds.

Protocol Complexity (w/o overflow). First note that the server-side memory storage requirement is simply that of the [CP13] ORAM construction, together with the $\log(2n_t/K)$ tree-depth memory optimization of [GHL+14]; namely, $f(n)$ memory overhead suffices for any $f \in \omega(1)$.

Consider the local memory required per CPU. Each CPU must be able to store: $O(\log n)$-size requests from each CPU (due to the broadcasts in Steps 1(a), 4(a), 5(a), and 7); and the data contents of at most 3 memory buckets (due to the flushing procedure in UpdateBuckets). Overall, this yields a per-CPU local memory requirement of $\tilde{\Omega}(m)$ (where $\tilde{\Omega}$ notation hides $\log n$ factors).

Consider the parallel complexity of the OPRAM-compiled program $\Pi' \leftarrow \mathsf{Heavy}\text{-}O(m, n, \Pi)$. For each parallel memory access in the underlying program Π, the processors perform: Conflict resolution (1 local communication round), Read/writing the position map (which has parallel complexity $O(\log^2 n \cdot \log(n/\alpha^t))$ by the inductive hypothesis), Looking up current memory values (sequential steps = depth of level-$(t-1)$ ORAM tree $\in O(\log(n/\alpha^{t-1}))$), Removing old data from the ORAM tree (1 local communication round, plus depth of the ORAM tree $\in O(\log(n/\alpha^{t-1}))$ sequential steps), Inserting the new data in parallel (1 local communication round, plus 1 communication round to the server), Flushing the ORAM database (1 local communication round, and $2\times$ the depth of the ORAM tree rounds of communication with the server, since each bucket along a flush path is accessed once to receive new data items and once to flush its own data items down), and Updating CPUs with the read values (1 local communication round). Altogether, this yields parallel complexity overhead $O(\log^2 n \cdot \log(n/\alpha^{t-1}))$.

It remains to address the probability of encountering overflow.

Claim 3.8. *There exists a negligible function μ such that for any deterministic m-processor PRAM program Π, any database size n, and any input x, the probability that the* Heavy-O*-compiled program*

$\Pi'(m, n, x)$ *outputs* overflow *is bounded by* $\mu(n)$.

Proof. We consider separately the probability of overflow in each of the level-t recursive ORAM trees. Since there are $\lceil \log n \rceil$ of them, the claim follows by a straightforward union bound.

Taking inspiration from [CP13], we analyze the ORAM-compiled execution via an abstract dart game. The game consists of black and white darts. In each round of the game, m black darts are thrown, followed by m white darts. Each dart independently hits the bullseye with probability $p = 1/m$. The game continues until exactly K darts have hit the bullseye (recall $K \in \omega(\log n)$ is the bucket size), or after the end of the Tth round for some fixed polynomial bound $T = T(n)$, whichever comes first. The game is "won" (which will correspond to overflow in a particular bucket) if K darts hit the bullseye, and all of them are black.

Let us analyze the probability of winning in the above dart game.

Subclaim 1: With overwhelming probability in n, no more than $K/2$ darts hit the bullseye in any round. In any single round, associate with each of the $2 \cdot m$ darts thrown an indicator variable X_i for whether the dart strikes the target. The X_i are independent random variables each equal to 1 with probability $p = 1/m$. Thus, the probability that more than $K/2$ of the darts hit the target is bounded (via a Chernoff tail bound[3]) by

$$\Pr\left[\sum_{i=1}^{2m} X_i > K/2\right] \leq e^{\frac{2(K/4-1)^2}{2+(K/4-1)}} \leq e^{-\Omega(K)} \leq e^{-\omega(\log n)}.$$

Since there are at most $T = \mathsf{poly}(n)$ distinct rounds of the game, the subclaim follows by a union bound.

Subclaim 2: Conditioned on no round having more than $K/2$ bullseyes, the probability of winning the game is negligible in d. Fix an arbitrary such winning sequence s, which terminates sometime during some round r of the game. By assumption, the final partial round r contains no more than $K/2$ bullseyes. For the remaining $K/2$ bullseyes in rounds 1 through $r-1$, we are in a situation mirroring that of [CP13]: for each such winning sequence s, there exist $2^{K/2} - 1$ distinct other "losing" sequences s' that each occur with the same probability, where any non-empty subset of black darts hitting the bullseye are replaced with their corresponding white darts. Further, every two distinct winning sequences s_1, s_2 yield disjoint sets of losing sequences, and all such constructed sequences have the property that no round has more than $K/2$ bullseyes (since this number of total bullseyes per round is preserved). Thus, conditioned on having no round with more than $K/2$ bullseyes, the probability of winning the game is bounded above by $2^{-K/2} \in e^{-\omega(\log n)}$.

We now relate the dart game to the analysis of our OPRAM compiler.

We analyze the memory buckets at the nodes in the t-th recursive ORAM tree, via three subcases.

Case 1: Nodes in level $\mathsf{lev} < \log m$. Since data items are inserted to the tree in parallel directly at level $\log m$, these nodes do not receive data, and thus will not overflow.

Case 2: Consider any internal node (i.e., a node that is not a leaf) γ in the tree at level $\log m \leq \mathsf{lev} < \log(L_t)$. (Recall $L_t := 2n_t/K$ is the number of leaves in the t'th tree when applying the [GHL+14] optimization). Note that when $m > L_t$, this case is vacuous. For purposes of analysis, consider the contents of γ as split into two parts: γ_L containing the data blocks whose leaf path

[3]Explicit Chernoff bound used: for $X = X_1 + \cdots X_{2m}$ (X_i independent) and mean μ, then for any $\delta > 0$, it holds that $\Pr[X > (1+\delta)\mu] \leq e^{-\delta^2\mu/(2+\delta)}$.

continues to the left from γ (i.e., leaf $\gamma||0||\cdot$), and γ_R containing the data blocks whose leaf path continues right (i.e., $\gamma||1||\cdot$). For the bucket of node γ to overflow, there must be K tuples in it. In particular, either γ_L or γ_R must have $K/2$ tuples.

For each parallel memory access in $\Pi(m, n, x)$, in the t-th recursive ORAM tree for which $n_t \geq m/K$, (at most) m data items are inserted, and then m independent paths in the tree are flushed. By definition, an inserted data item will enter our bucket γ_L (respectively, γ_R) only if its associated leaf has the prefix $\gamma||0$ (resp., $\gamma||1$); we will assume the worst case in which *all* such data items arrive directly to the bucket. On the other hand, the bucket γ_L (resp., γ_R) will be completely emptied after any flush whose path contains this same prefix $\gamma||0$ (resp., $\gamma||1$). Since all leaves for inserted data items and data flushes are chosen randomly and independently, these events correspond directly to the black and white darts in the game above. Namely, the probability that a randomly chosen path will have the specific prefix $\gamma||0$ of length lev is $2^{-\mathsf{lev}} \leq 1/m$ (since we consider $\mathsf{lev} \geq \log m$); this corresponds to the probability of a dart hitting the bullseye. The bucket can only overflow if $K/2$ "black darts" (inserts) hit the bullseye without any "white dart" (flush) hitting the bullseye in between. By the analysis above, we proved that for any sequence of $K/2$ bullseye hits, the probability that all $K/2$ of them are black is bounded above by $2^{-K/4}$, which is negligible in n. However, since there is a fixed polynomial number $T = \mathsf{poly}(n)$ of parallel memory accesses in the execution of $\Pi(m, n, x)$ (corresponding to the number of "rounds" in the dart game), and in particular, $T(2m) \in \mathsf{poly}(n)$ total darts thrown, the probability that the sequence of bullseyes contains $K/2$ sequential blacks *anywhere* in the sequence is bounded via a direct union bound by $(T2m)2^{-K/4} \in e^{-\omega(\log n)}$, as desired.

Case 3: Consider any leaf node γ. This analysis follows the same argument as in [CP13] (with slightly tweaked parameters from the [GHL+14] tree-depth optimization). For there to be an overflow in γ at time t, there must be $K + 1$ out of n_t/α elements in the position map that map to the leaf γ. Since all positions are sampled uniformly and independently among the $L_t := 2n_t/K$ different leaves, the expected number of elements mapping to γ is $\mu = K/2\alpha$, and by a standard multiplicative Chernoff bound,[4] the probability that $K + 1$ elements are mapped to γ is upper bounded by

$$\left(\frac{e^1}{(1+1)^{(1+1)}}\right)^{\mu} \leq (2^{1/3})^{-K/2\alpha} \in 2^{-\omega(\log n)}.$$

□

Thus, the total probability of $\mathsf{overflow}$ is negligible in n, and the theorem follows.

□

3.2 Oblivious Distributed Insertion, Aggregation, and Multi-Cast

3.2.1 Oblivious Parallel Insertion (Oblivious Routing)

Recall during the memory "put-back" phase, each CPU must insert its data item into the bucket at level $\log m$ of the tree lying along a freshly sampled random path, while *hiding* the path.

We solve this problem by delivering data items to their target locations via a *fixed-topology routing network*. Namely, the m processors $CPU_1, \ldots, CPU_m$ will first write the relevant m data items msg_i (and their corresponding destination addresses addr_i) to memory in fixed order, and

[4] We use the following version of the Chernoff bound: Let $X_1, \ldots, L_n$ be independent $[0, 1]$-valued random variables. Let $X = \sum_i X_i$ and $\mu = E[X]$. For every $\delta > 0$, $\Pr[X \geq (1 + \delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$.

Parallel Insertion Routing Protocol $\mathsf{Route}(m, (\mathsf{msg}_i, \mathsf{addr}_i))$
Input: CPU_i holds: message msg_i with target destination addr_i, and global threshold K.
Output: CPU_i holds $\{\mathsf{msg}_j : \mathsf{addr}_j = i\}$.

Let $\mathsf{lev}^* = \log m$ (assumed $\in \mathbb{N}$ for simplicity). Each CPU_i performs the following.

Initialize $M_{i,0} \leftarrow \mathsf{msg}_i$. For $t = 1, \ldots, \mathsf{lev}^*$:

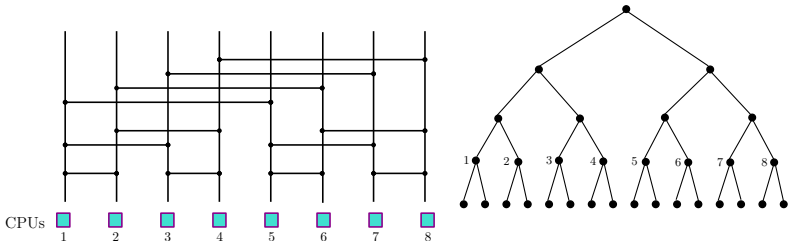
1. Perform the following symmetric message exchange with $CPU_{i \oplus 2^t}$:
$$M_{i,t+1} \leftarrow \{\mathsf{msg}_j \in M_{i,t} \cup M_{i \oplus 2^t, t} : (\mathsf{addr}_j)_t = (i)_t\}.$$
2. If $|M_{i,t+1}| > K$ (i.e., memory overflow), then CPU_i aborts.

Figure 5: Fixed-topology routing network for delivering m messages originally held by m processors to their corresponding destination addresses within $[m]$.

then rearrange them in $\log m$ sequential rounds to the proper locations via the routing network. At the conclusion of the routing procedure, each node j will hold all messages msg_i for which $\mathsf{addr}_i = j$.

For simplicity, assume $m = 2^\ell$ for some $\ell \in \mathbb{N}$. The routing network has depth ℓ; in each level $t = 1, \ldots, \ell$, each node communicates with the corresponding node whose id agrees in all bit locations except for the tth (corresponding to his tth neighbor in the $\log m$-dimensional boolean hypercube). These nodes exchange messages according to the tth bit of their destination addresses addr_i. This is formally described in Figure 5. After the tth round, each message msg_i is held by a party whose id agrees with the destination address addr_i in the first t bits. Thus, at the conclusion of ℓ rounds, all messages are properly delivered.

We demonstrate the case $m = 8 = 2^3$ below: first, CPUs exchange information along the depicted communication network in 3 sequential rounds (left); then, each CPU i inserts his resulting collection of items directly into node i of level 3 of the data tree (right).



We show that if the destination addresses addr_i are uniformly sampled, then with overwhelming probability no node will ever need to hold too many (the threshold K will be set to $\omega(\log n)$) messages at any point during the routing network execution:

Lemma 3.9 (Routing Network)**.** *If L messages begin with target destination addresses addr_i distributed independently and uniformly over $[L]$ in the L-to-L node routing network in Figure 5, then with probability bounded by $1 - (L \log L)2^{-K}$, no intermediate node will ever hold greater than K messages at any point during the course of the protocol execution.*

Proof. Consider an arbitrary node $a \in \{0,1\}^\ell$, at some level t of execution of the protocol. There are precisely 2^t possible messages m_i that could be held by node a at this step, corresponding to those originating in locations $b \in \{0,1\}^\ell$ whose final $\ell - t$ bits agree with those of a. Node a will

hold message m_b at the conclusion of round t precisely if the *first* t bits of addr_b agree with those of a. For each such message m_b, the associated destination address addr_b is a random element of $[L]$, which agrees with a on the first t bits with probability 2^t.

For each $b \in \{0,1\}^\ell$ agreeing with a on the final $\ell - t$ bits, define X_b to be the indicator variable that is equal to 1 if addr_b agrees with a on the first t bits. Then the collection of 2^t random variables $\{X_b : b_i = a_i \ \forall i = t+1, \ldots, \ell\}$ are independent, and $X = \sum X_b$ has mean $\mu = 2^t \cdot 2^{-t} = 1$. Note that X corresponds to the number of messages held by node a at level t. By a Chernoff bound,[5] it holds that

$$\Pr[X \geq K] = \Pr[X \geq (1 + (K-1))\mu] < \left(\frac{e^{K-1}}{K^K}\right) < 2^{-K}.$$

Then, taking a union bound over the total number of nodes L and levels $\ell = \log L$, we have that the probability of any node experiencing an overflow at any round is bounded by $(L \log L)2^{-K}$. □

3.2.2 Oblivious Aggregation

To perform the "CPU-coordination" phase, the CPUs efficiently identify a single representative and *aggregate* relevant CPU instructions; then, at the conclusion, the representative CPU must be able to *multi-cast* the resulting information to all relevant requesting CPUs. Most importantly, these procedures must be done *in an oblivious fashion*. In this section, we address oblivious aggregation; we treat the dual multi-cast problem in Section 3.2.3.

Formally, we want to achieve the following aggregation goal, with communication patterns independent of the inputs, using only $O(\log(m)\mathsf{polylog}(n))$ local memory and communication per CPU, in only $O(\log(m))$ sequential time steps. An illustrative example to keep in mind is where $\mathsf{key}_i = b_i$, $\mathsf{data}_i = v_i$, and Agg is the process that combines instructions to data items within the same data block, resolving conflicts as necessary.

Oblivious aggregation:

Input: Each CPU $i \in [m]$ holds $(\mathsf{key}_i, \mathsf{data}_i)$. Let $\mathsf{K} = \bigcup\{\mathsf{key}_i\}$ denote the set of distinct keys. We assume that any (subset of) data associated with the same key can be aggregated by an aggregation function Agg to a short digest of size at most $\mathsf{poly}(\ell, \log m)$, where $\ell = |\mathsf{data}_i|$.

Goal: Each CPU i outputs out_i such that the following holds.

- For every $\mathsf{key} \in \mathsf{K}$, there exists unique agent i with $\mathsf{key}_i = \mathsf{key}$ s.t. $\mathsf{out}_i = (\mathsf{rep}, \mathsf{key}, \mathsf{agg}_{\mathsf{key}})$, where $\mathsf{agg}_{\mathsf{key}} = \mathsf{Agg}(\{\mathsf{data}_j : \mathsf{key}_j = \mathsf{key}\})$.
- For every remaining agent i, $\mathsf{out}_i = (\mathsf{dummy}, \perp, \perp)$.

At a high level, we achieve this via the following steps. (1) First, the CPUs sort their data list with respect to the corresponding key values. This can be achieved via an implementation of a $\log(m)$-depth sorting network, and provides the useful guarantee that all data pertaining to the same key are necessarily held by an block of adjacent CPUs. (2) Second, we pass data among CPUs in a sequence of $\log(m)$ steps such that at the conclusion the "left-most" (i.e., lowest indexed) CPU in each key-block will learn the aggregation of *all* data pertaining to this key. Explicitly, in each step i, each CPU sends all held information to the CPU 2^i to the "left" of him, and simultaneously accepts any received information pertaining to his key. (3) Third, each CPU will learn whether he is the "left-most" representative in each key-block, by simply checking whether his left-hand

[5] Exact Chernoff bound used: $\Pr[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$ for any $\delta > 0$.

neighbor holds the same key. From here, the CPUs have succeeded in aggregating information for each key at a single representative CPU; (4) in the fourth step, they now reverse the original sorting procedure to return this aggregated information to one of the CPUs who originally requested it.

Lemma 3.10 (Space-Efficient Oblivious Aggregation)**.** *Suppose m processors initiate protocol* OblivAgg *w.r.t. aggregator* Agg*, on respective inputs $\{(\mathsf{key}_i, \mathsf{data}_i)\}_{i\in[m]}$, each of size ℓ. Then at the conclusion of execution, each processor $i \in [m]$ outputs a triple $(\mathsf{rep}'_i, \mathsf{key}'_i, \mathsf{data}'_i)$ such that the following properties hold (where asymptotics are w.r.t. m):*

1. *The protocol terminates in $O(\log m)$ rounds.*
2. *The local memory and computation required per processor is $O(\log m + \ell)$.*
3. *(Correctness). For every key $\mathsf{key} \in \bigcup\{\mathsf{key}_i\}$, there exists a unique processor i with output $\mathsf{key}'_i = \mathsf{key}$. For each such processor, it further holds that $\mathsf{key}'_i = \mathsf{key}_i$, $\mathsf{rep}'_i =$ "rep", and $\mathsf{data}'_i = \mathsf{Agg}(\{\mathsf{data}_j : \mathsf{key}_j = \mathsf{key}_i\})$. For every remaining processor, the output tuple is $(\mathsf{dummy}, \perp, \perp)$.*
4. *(Obliviousness). The inter-CPU communication patterns are independent of the inputs $(\mathsf{key}_i, \mathsf{data}_i)$.*

A full description of our Oblivious Aggregation procedure OblivAgg is given in Figure 6.

Proof of Lemma 3.10. Property (1): Steps 1 and 4 of OblivAgg each execute a sorting network, and require communication rounds equal to the depth $d \in O(\log m)$ of the sorting network implemented. Step 2 takes place in $\log m$ sequential steps. Step 3 requires a single round. And Step 5 (output) takes place locally. Thus, the combined round complexity of OblivAgg is $O(\log m)$.

Property (2): We first address the size the individual items stored, and then ensure the number of stored items is never too large.

- Keys (e.g., $\mathsf{key}_i, \mathsf{tempkey}_i$): Each key is bounded in size by the initial input size ℓ.
- Data (e.g., $\mathsf{data}_i, \mathsf{datatemp}_i, \mathsf{aggdata}_i$): Similarly, by the property of the aggregation function Agg, we are guaranteed that each data item is bounded in size by the original data size, which is in turn bounded by size ℓ.
- CPU identifiers (e.g., $\mathsf{sourceid}_i, \mathsf{idtemp}_i$): Each processor can be identified by bit string of length $\log m$.
- Representative flag (rep_i): The rep/dummy flag can be stored as a single bit.

Each processor begins with input size ℓ. In each round of executing the first sorting network (Step 1 of OblivAgg), a processor must hold *two* sets of data (sourceid, keytemp, datatemp), corresponding to at most $2(\log m + 2\ell)$ storage. Note that no more than 2 tuples are required to be held at any time within this step, as the processors exchange tuples but need not maintain both values. In each round of the Aggregation phase (Step 2), processors may need to store two pairs (keytemp, datatemp) in addition to the information held from the conclusion of the previous step (namely, a single value $\mathsf{sourceid}_i$), which totals to $\log m + 2(2\ell)$ memory. Note that by the properties of the aggregation scheme Agg, the size of the aggregated data does not grow beyond ℓ (and recall that parties do not maintain data associated with any different key). In the Representative Identification phase (Step 3), each processor receives one additional key value key_{i-1}, which requires memory $\log m$, and is then translated to a single-bit flag rep_i and then deleted. In the Reverse Sort phase (Step 4), processors within each round must again store two tuples, this time of the form (idtemp, rep, keytemp, datatemp), which corresponds to $2(\log m + 1 + \ell + \ell)$ memory. Thus, the total local memory requirement per processor is bounded by $O(\log m + \ell)$.

Oblivious Aggregation Procedure OblivAgg **(w.r.t.** Agg**)**
Input: Each CPU $i \in [m]$ holds a pair $(\mathsf{key}_i, \mathsf{data}_i)$.
Output: Each CPU $i \in [m]$ outputs a triple $(\mathsf{rep}_i, \mathsf{key}_i, \mathsf{aggdata}_i)$ corresponding to either $(\mathsf{dummy}, \perp, \perp)$ or with $\mathsf{aggdata}_i = \mathsf{Agg}(\{\mathsf{data}_j : \mathsf{key}_j = \mathsf{key}_i\})$, as further specified in Section 3.2.

1. **Sort on** key_i**.** Each CPU_i initializes a triple $(\mathsf{sourceid}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i) \leftarrow (i, \mathsf{key}_i, \mathsf{data}_i)$.
   For each layer $L_1, \ldots, L_d$ in the sorting network:
   - Let $L_\ell = ((i_1, j_1), \ldots, (i_{m/2}, j_{m/2}))$ be the comparators in the current layer ℓ.
   - In *parallel*, for each $t \in [m/2]$, the corresponding pair of CPUs (CPU_{i_t}, CPU_{j_t}) perform the following pairwise sort w.r.t. key:
     $$\text{If } \mathsf{keytemp}_{j_t} < \mathsf{keytemp}_{i_t}, \text{ then}$$
     $$\text{swap } (\mathsf{sourceid}_{i_t}, \mathsf{keytemp}_{i_t}, \mathsf{datatemp}_{i_t}) \leftrightarrow (\mathsf{sourceid}_{j_t}, \mathsf{keytemp}_{j_t}, \mathsf{datatemp}_{j_t}).$$
2. **Aggregate to left.** For $t = 0, 1, \ldots, \log m$:
   - (Pass to left). Each CPU_i for $i > 2^t$ sends his current pair $(\mathsf{keytemp}_i, \mathsf{datatemp}_i)$ to CPU_{i-2^t}.
   - (Aggregate). Each CPU_i for $i < m - 2^t$ receiving a pair $(\mathsf{keytemp}_j, \mathsf{datatemp}_j)$ will aggregate it into own pair if the keys match. That is, if $\mathsf{keytemp}_i = \mathsf{keytemp}_j$, then set $\mathsf{datatemp}_i \leftarrow \mathsf{Agg}(\mathsf{datatemp}_i, \mathsf{datatemp}_j)$. In both cases, the received pair is then erased.

   The left-most CPU_i with $\mathsf{keytemp}_i = \mathsf{key}$ now has $\mathsf{Agg}(\{\mathsf{datatemp}_j : \mathsf{keytemp}_j = \mathsf{key}\}))$.
3. **Identify representatives.** For each value key_j, the left-most CPU i currently holding $\mathsf{keytemp}_i = \mathsf{key}_j$ will identify himself as (temporary) representative.
   - Each CPU_i for $i < m$: send $\mathsf{keytemp}_i$ to right-hand neighbor, CPU_{i+1}.
   - Each CPU_i for $i > 1$: If the received value $\mathsf{keytemp}_{i-1}$ matches his own $\mathsf{keytemp}_i$, then set $\mathsf{rep}_i \leftarrow$ "dummy" and zero out $\mathsf{keytemp}_i \leftarrow \perp, \mathsf{datatemp}_i \leftarrow \perp$. Otherwise, set $\mathsf{rep}_i \leftarrow$ "rep". (CPU_1 always sets $\mathsf{rep}_1 \leftarrow$ "rep").
4. **Reverse sort (i.e., sort on** $\mathsf{sourceid}_i$**).** Return aggregated data to a requesting CPU.
   For each layer $L_1, \ldots, L_d$ in the sorting network:
   - Let $L_\ell = ((i_1, j_1), \ldots, (i_{m/2}, j_{m/2}))$ be the comparators in the current layer ℓ.
   - Each CPU_i initializes $\mathsf{idtemp} \leftarrow \mathsf{sourceid}_i$. In *parallel*, for each $t \in [m/2]$, the corresponding pair of CPUs (CPU_{i_t}, CPU_{j_t}) perform the following pairwise sort w.r.t. $\mathsf{sourceid}$:
     $$\text{If } \mathsf{idtemp}_{j_t} < \mathsf{idtemp}_{i_t}, \text{ then}$$
     $$\text{swap } (\mathsf{idtemp}_{i_t}, \mathsf{rep}_{i_t}, \mathsf{keytemp}_{i_t}, \mathsf{datatemp}_{i_t}) \leftrightarrow (\mathsf{idtemp}_{j_t}, \mathsf{rep}_{j_t}, \mathsf{keytemp}_{j_t}, \mathsf{datatemp}_{j_t}).$$

   At the conclusion, each CPU_i holds a tuple $(\mathsf{idtemp}_i, \mathsf{rep}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ with $\mathsf{idtemp}_i = i$ and $\mathsf{keytemp}_i = \mathsf{key}_i$.
5. **Output.** Each CPU_i outputs the triple $(\mathsf{rep}_i, \mathsf{key}_i, \mathsf{datatemp}_i)$.

Figure 6: Space-efficient oblivious data aggregation procedure.

Property (3): We now prove that the protocol results in the desired output. Consider the values stored by each processor at the conclusion of each phase of the protocol.

After the completion of Step 1, by the correctness of the utilized sorting network, it holds that each CPU_i holds a tuple $(\mathsf{sourceid}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ such that the list $(\mathsf{sourceid}_1, \ldots, \mathsf{sourceid}_m)$ is some permutation of $[m]$, and $\mathsf{keytemp}_i \leq \mathsf{keytemp}_j$ for every $i < j$. Note that for each i it always the case that the pair $(\mathsf{keytemp}_i, \mathsf{datatemp}_i)$ currently held by CPU_i is precisely the original input pair of CPU_j for $j = \mathsf{sourceid}_i$.

For the Aggregation phase in Step 2, we make the following claim.

Claim 3.11. *At the conclusion of Aggregate Left (Step 2), the CPU of lowest index i for which $\mathsf{keytemp}_i = \mathsf{key}$ holds $\mathsf{datatemp}_i = \mathsf{Agg}(\{\mathsf{data}_j : \mathsf{key}_j = \mathsf{key}\})$ (for each value key).*

Proof. Fix an arbitrary value key, and let $S_{\mathsf{key}} \subset [m]$ denote the subset of processors for which $\mathsf{keytemp}_i = \mathsf{key}$. From the previous sorting step, we are guaranteed that S_{key} consists of an interval of consecutive processors $i_{\mathsf{start}}, \ldots, i_{\mathsf{stop}}$. Now, consider any $j \in S_{\mathsf{key}}$ (whose data CPU i_{start} wishes to learn).

For any pair of indices $i < j \in S_{\mathsf{key}}$, denote by $t_{i,j} := \max\{t \in [\log m] : (j \oplus i_{\mathsf{start}})_t = 1\} \in \{0, 1, \ldots, \log m - 1\}$ the highest index in which the bit representations of j and i_{start} disagree. We now prove that for each such pair i, j, CPU_i will learn CPU_j's data after round $t_{i,j} \leq \log m$. The claim will follow, by applying this statement to each pair (i_{start}, j) with $j \in S_{\mathsf{key}}$.

Induct on $t_{i,j}$. Base case $t_{i,j} = 0$: follows immediately from the protocol construction; namely, in the 0-th round, each CPU j sends his data to CPU $(j-1)$, which in this case is precisely CPU i. Now, suppose the inductive hypothesis holds for all $i < j$ with $t_{i,j} = t$, and consider a pair $i < j$ with $t_{i,j} = t + 1$. In round $t + 1$ of the protocol, processor i receives from processor $(i + 2^{t+1})$ the collection of all information it has aggregated up to round t. By the definition of $t_{i,j}$, we know that $i < (i + 2^{t+1}) \leq j$, and that $t_{(i+2^{t+1}),j} \leq t$. Indeed, we know that i and j differ in bit index $(t + 1)$, and no higher; thus, $(i + 2^t)$ must agree with j in index $(t + 1)$ in addition to all higher indices. But, this means by the inductive hypothesis that CPU $(i + 2^t)$ has learned CPU j's data in a previous round. Thus, CPU i will learn CPU j's data in round $t + 1$, as desired.

□

In Step 3, each processor learns whether his left-hand neighbor holds the same temporary key as he does; that is, he learns whether or not he is the left-most CPU holding $\mathsf{tempkey}_i$ (and, in turn, holds the complete aggregation of all data relating to this key). Each processor for whom this is not the case sets his tuple to $(\mathsf{dummy}, \perp, \perp)$.

At this point in the protocol, the processors have successfully reached the state where a single self-identified representative for each queried key holds the desired data aggregation. The final step is to return these information tuples to some CPU who originally requested this key. This is achieved in the final reverse sort (Step 4). Namely, by the correctness of the implemented sorting network, at the conclusion of Step 4 each CPU_i holds a tuple $(\mathsf{idtemp}_i, \mathsf{rep}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ such that the ordered list $(\mathsf{idtemp}_1, \ldots, \mathsf{idtemp}_m)$ is precisely the ordered list $1, \ldots, m$. Since the tuples $(\mathsf{idtemp}_i, \mathsf{rep}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ are never modified (only swapped between processors), it remains to show that each non-dummy $(\mathsf{rep}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ tuple is received by an appropriate requesting CPU. But, that is precisely the information held by idtemp_i: the identity of the CPU who made the original request with respect to key $\mathsf{keytemp}_i$. Thus, the reverse sort successfully routes the aggregated tuples back to a CPU making the correct key request.

Property (4): Since we utilize a sorting network with fixed topology, and the aggregate-to-left functionality has fixed communication topology, the inter-CPU communication patterns are constant, independent of the initial CPU inputs.

□

3.2.3 Oblivious Multicasting

Our goal for Oblivious Multicasting is dual to that of the previous section: Namely, a subset of CPUs must deliver information to (unknown) collections of other CPUs who request it. This is abstractly modeled as follows, where key_i denotes which data item is requested by each CPU i.

Oblivious Multicasting:

Input: Each CPU i holds $(\mathsf{key}_i, \mathsf{data}_i)$ with the following promise. Let $\mathsf{K} = \bigcup\{\mathsf{key}_i\}$ denote the set of distinct keys. For every $\mathsf{key} \in \mathsf{K}$, there exists a unique agent i with $\mathsf{key}_i = \mathsf{key}$ such that $\mathsf{data}_i \neq \perp$; let $\mathsf{data}_{\mathsf{key}}$ denote such data_i.

Goal: Each agent i outputs $\mathsf{out}_i = (\mathsf{key}_i, \mathsf{data}_{\mathsf{key}_i})$.

Figure 7 contains the protocol OblivMCast for achieving oblivious multicasting in a space-efficient fashion. This procedure is roughly the "dual" of the OblivAgg protocol in the previous section.

Lemma 3.12 (Space-Efficient Oblivious Multicasting)**.** *Suppose m processors initiate protocol* OblivMCast *on respective inputs $\{(\mathsf{key}_i, \mathsf{data}_i)\}_{i\in[m]}$ of size ℓ that satisfies the promise specified above. Then at the conclusion of execution, each processor $i \in [m]$ outputs a pair $(\mathsf{key}'_i, \mathsf{data}'_i)$ such that the following properties hold (where asymptotics are w.r.t. m):*

1. *The protocol terminates in $O(\log m)$ rounds.*
2. *The local memory and computation required by each processor is $\tilde{O}(\log m + \ell)$.*
3. *(Correctness). For every i, $\mathsf{key}'_i = \mathsf{key}_i$, and $\mathsf{data}'_i = \mathsf{data}_{\mathsf{key}_i}$.*
4. *(Obliviousness). The inter-CPU communication patterns are independent of the inputs $(\mathsf{key}_i, \mathsf{data}_i)$.*

Proof. Identical to the proof of Oblivious Aggregation, Lemma 3.10. □

3.3 Putting Things Together

We now combine the so-called "Heavy-OPAccess" structure of our OPRAM formalized in Section 3.1 (Figure 2) within the simplified "free CPU communication" setting, together with the (oblivious) Route, OblivAgg, and OblivMCast procedures constructed in the previous subsections (specified in Figures 5,6,7). For simplicity, we describe the case in which the number of CPUs m is fixed; however, it can be modified in a straightforward fashion to the more general case (as long as the activation schedule of CPUs is a-priori fixed and public).

Recall the steps in Heavy-OPAccess where large memory/bandwidth are required.

- In Step 1, each CPU_i broadcasts (r_i, v_i) to all CPUs. Let $b_i = \lfloor r_i/\alpha \rfloor$. This is used to aggregate instructions to each b_i and determine its representative CPU $\mathsf{rep}(b_i)$.
- In Step 4, each CPU_i broadcasts (b_i, ℓ_i) or $(\perp, \ell_i)$. This is used to aggregate instructions to each buckets along path ℓ_i about which blocks b_i's to be removed.

Oblivious Multicasting Procedure OblivMCast

Input: Each CPU i holds $(\mathsf{key}_i, \mathsf{data}_i)$ with the following promise. Let $\mathsf{K} = \bigcup\{\mathsf{key}_i\}$ denote the set of distinct keys. For every $\mathsf{key} \in \mathsf{K}$, there exists a unique agent i with $\mathsf{key}_i = \mathsf{key}$ such that $\mathsf{data}_i \neq \perp$; let $\mathsf{data}_{\mathsf{key}}$ denote such data_i.

Output: Each agent i outputs $\mathsf{out}_i = (\mathsf{key}_i, \mathsf{data}_{\mathsf{key}_i})$.

1. **Sort on** $(\mathsf{key}_i, \mathsf{data}_i)$**.** Each CPU_i initializes $(\mathsf{sourceid}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i) \leftarrow (i, \mathsf{key}_i, \mathsf{data}_i)$. For each layer $L_1, \ldots, L_d$ in the sorting network:
   - Let $L_\ell = ((i_1, j_1), \ldots, (i_{m/2}, j_{m/2}))$ be the comparators in the current layer ℓ.
   - In *parallel*, for each $t \in [m/2]$, the corresponding pair of CPUs (CPU_{i_t}, CPU_{j_t}) perform the following pairwise sort w.r.t. key, additionally pushing payloads $\mathsf{data}_{\mathsf{key}}$ to the left:

     If (i) $\mathsf{keytemp}_{j_t} < \mathsf{keytemp}_{i_t}$, or (ii) $\mathsf{keytemp}_{j_t} = \mathsf{keytemp}_{i_t}$ and $\mathsf{datatemp}_{j_t} \neq \perp$, then swap $(\mathsf{sourceid}_{i_t}, \mathsf{keytemp}_{i_t}, \mathsf{datatemp}_{i_t}) \leftrightarrow (\mathsf{sourceid}_{j_t}, \mathsf{keytemp}_{j_t}, \mathsf{datatemp}_{j_t})$.
2. **Multicast to right.** For $t = 0, 1, \ldots, \log m$:
   - (Pass to right). Each CPU_i for $i \leq m - 2^t$ sends his current pair $(\mathsf{keytemp}_i, \mathsf{datatemp}_i)$ to CPU_{i+2^t}.
   - (Aggregate). Each CPU_i for $i > 2^t$ receiving a pair $(\mathsf{keytemp}_j, \mathsf{datatemp}_j)$ with $j = i - 2^t$ update its data as follows. If $\mathsf{keytemp}_i = \mathsf{keytemp}_j$ and $\mathsf{datatemp}_j \neq \perp$, then set $\mathsf{datatemp}_i \leftarrow \mathsf{datatemp}_j$.

   Every CPU i now holds $(\mathsf{keytemp}_i, \mathsf{datatemp}_i) = (\mathsf{key}, \mathsf{data}_{\mathsf{key}})$ for some $\mathsf{key} \in \mathsf{K}$.
3. **Reverse sort (i.e., sort on** $\mathsf{sourceid}_i$**).** Return received data to an original requesting CPU. For each layer $L_1, \ldots, L_d$ in the sorting network:
   - Let $L_\ell = ((i_1, j_1), \ldots, (i_{m/2}, j_{m/2}))$ be the comparators in the current layer ℓ.
   - Each CPU_i initializes $\mathsf{idtemp} \leftarrow \mathsf{sourceid}_i$. In *parallel*, for each $t \in [m/2]$, the corresponding pair of CPUs (CPU_{i_t}, CPU_{j_t}) perform the following pairwise sort w.r.t. $\mathsf{sourceid}$:

     If $\mathsf{idtemp}_{j_t} < \mathsf{idtemp}_{i_t}$, then swap $(\mathsf{idtemp}_{i_t}, \mathsf{keytemp}_{i_t}, \mathsf{datatemp}_{i_t}) \leftrightarrow (\mathsf{idtemp}_{j_t}, \mathsf{keytemp}_{j_t}, \mathsf{datatemp}_{j_t})$.

   At the conclusion, each CPU_i holds a tuple with $(\mathsf{idtemp}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ with $\mathsf{idtemp}_i = i$, $\mathsf{keytemp}_i = \mathsf{key}_i$, and $\mathsf{datatemp}_i = \mathsf{data}_{\mathsf{key}_i}$.
4. **Output.** Each CPU_i outputs $\mathsf{output}_i = (\mathsf{key}_i, \mathsf{data}_{\mathsf{key}_i})$.

Figure 7: Space-efficient oblivious data multicasting procedure.

- In Step 5, each (representative) CPU_i broadcasts $(b_i, \bar{v}_i, \ell'_i)$. This is used to aggregate blocks to be inserted to each bucket in appropriate level of the tree.
- In Step 6, each CPU_i broadcasts ℓ_i^{flush}. This is used to aggregate information about which buckets the flush operation should perform.
- In Step 7, each (representative) $CPU_{\mathsf{rep}(b)}$ broadcasts the old value v_{old} of block b to all CPUs, so that each CPU receives desired information.

We will use oblivious aggregation procedure to replace broadcasts in Step 1, 4, and 6; the parallel insertion procedure to replace broadcasts in Step 5, and finally the oblivious multicast procedure to replace broadcasts in Step 7.

Let us first consider the aggregation steps. For Step 1, to invoke the oblivious aggregation procedure, we set $\mathsf{key}_i = b_i$ and $\mathsf{data}_i = (r_i \mod \alpha, v_i)$, and define the output of $\mathsf{Agg}(\{(u_i, v_i)\})$ to be a vector $\bar{v} = \bar{v}[1] \cdots \bar{v}[\alpha]$ of read/write instructions to each memory cell in the block, where conflicts are resolved by writing the value specified by the smallest CPU: i.e., $\forall s \in [\alpha]$, take $\bar{v}[s] \leftarrow v_j$ for minimal j such that $u_j = s$ and $v_j \neq \perp$. By the functionality of $\mathsf{OblivAgg}$, at the conclusion of $\mathsf{OblivAgg}$, each block b_i is assigned to a unique representative (not necessarily the smallest CPU), who holds the aggregation of all instructions on this block.

Both Step 4 and 6 invoke $\mathsf{UpdateBuckets}$ to update buckets along m random paths. In our rudimentary solution, the paths (along with instructions) are broadcast among CPUs, and the buckets are updated level by level. At each level, each update bucket is assigned to a representative CPU with minimal index, who performs aggregated instructions to update the bucket. Here, to avoid broadcasts, we invoke the oblivious aggregation procedure per level as follows.

- In Step 4, each CPU i holds a path ℓ_i and a block b_i (or $\perp$) to be removed. Also note that the buckets along the path ℓ_i are stored locally by each CPU i, after the read operation in the previous step (Step 3). At each level $\mathsf{lev} \in [\log n]$, we invoke the oblivious aggregation procedure with $\mathsf{key}_i = \ell_i^{(\mathsf{lev})}$ (the lev-bits prefix of ℓ_i) and $\mathsf{data}_i = b_i$ if b_i is in the bucket of node $\ell_i^{(\mathsf{lev})}$, and $\mathsf{data}_i = \perp$ otherwise. We simply define $\mathsf{Agg}(\{\mathsf{data}_i\}) = \{b : \exists \mathsf{data}_i = b\}$ to be the union of blocks (to be removed from this bucket). Since $\mathsf{data}_i \neq \perp$ only when data_i is in the bucket, the output size of Agg is upper bounded by the bucket size K. By the functionality of $\mathsf{OblivAgg}$, at the conclusion of $\mathsf{OblivAgg}$, each bucket $\ell_i^{(\mathsf{lev})}$ is assigned to a unique representative (not necessarily the smallest CPU) with aggregated instruction on the bucket. Then the representative CPUs can update the corresponding buckets accordingly.
- In Step 6, each CPU i samples a path ℓ_i^{flush} to be flushed and the instructions to each bucket are simply left and right flushes. At each level $\mathsf{lev} \in [\log n]$, we invoke the oblivious aggregation procedure with $\mathsf{key}_i = \ell_i^{\mathsf{flush}(\mathsf{lev})}$ and $\mathsf{data}_i = L$ (resp., R) if the $(\mathsf{lev}+1)$-st bit of ℓ_i^{flush} is 0 (resp., 1). The aggregation function Agg is again the union function. Since there are only two possible instructions, the output has $O(1)$ length. By the functionality of $\mathsf{OblivAgg}$, at the conclusion of $\mathsf{OblivAgg}$, each bucket $\ell_i^{\mathsf{flush}(\mathsf{lev})}$ is assigned to a unique representative (not necessarily the smallest CPU) with aggregated instruction on the bucket. To update a bucket $\ell_i^{\mathsf{flush}(\mathsf{lev})}$, the representative CPU loads the bucket and its two children (if needed) into local memory from the server, performs the flush operation(s) locally, and writes the buckets back.

Note that since we update m *random* paths, we do not need to hide the access pattern, and thus the dummy CPUs do not need to perform dummy operations during $\mathsf{UpdateBuckets}$. A formal description of full-fledged $\mathsf{UpdateBuckets}$ can be found in Figure 8.

For Step 5, we rely on the parallel insertion procedure of Section 3.2.1, which routes blocks to proper destinations within the relevant level of the server-held data tree in parallel using a simple oblivious routing network. The procedure is invoked with $\mathsf{msg}_i = b_i$ and $\mathsf{addr}_i = \ell'_i$.

Finally, in Step 7, each representative CPU $\mathsf{rep}(b)$ holds information of the block b, and each dummy CPU i wants to learn the value of a block b_i. To do so, we invoke the oblivious multicast procedure with $\mathsf{key}_i = b_i$ and $\mathsf{data}_i = v_i^{old}$ for representative CPUs and $\mathsf{data}_i = \perp$ for dummy CPUs. By the functionality of $\mathsf{OblivMCast}$, at the conclusion of $\mathsf{OblivMCast}$, each CPU receives the value of the block it originally wished to learn.

The Final Compiler. For convenience, we summarize the complete protocol. Our OPRAM compiler O, on input $m, n_t \in \mathbb{N}$ and a m-processor PRAM program Π with memory size n_t (which in recursion level t will be $n_t = n/\alpha^t$), will output a program Π' that is identical to Π, but where each $\mathsf{Access}(r, v)$ operation is replaced by a sequence of operations defined by subroutine $\mathsf{OPAccess}(r, v)$, which we will construct over the following subsections. The $\mathsf{OPAccess}$ procedure begins with m CPUs, each with a requested data cell r_i (within some α-block b_i) and some action to be taken (either $\perp$ to denote read, or v_i to denote rewriting cell r_i with value v_i).

1. **Conflict Resolution:** Run $\mathsf{OblivAgg}$ on inputs $\{(b_i, v_i)\}_{i\in[m]}$ to select a unique representative $\mathsf{rep}(b_i)$ for each queried block b_i and aggregate all CPU instructions for this b_i (denoted $\bar{v}_i$).
2. **Recursive Access to Position Map:** Each representative CPU $\mathsf{rep}(b_i)$ samples a fresh random leaf id $\ell'_i \leftarrow [n_t]$ in the tree and performs a (recursive) Read/Write access command on the position map database $\ell_i \leftarrow \mathsf{OPAccess}(t+1, (b_i, \ell'_i))$ to fetch the current position map value ℓ for block b_i and rewrite it with the newly sampled value ℓ'_i. Each dummy CPU performs an arbitrary dummy access (e.g., $\mathsf{garbage} \leftarrow \mathsf{OPAccess}(t+1, (1, \emptyset))$).
3. **Look Up Current Memory Values:** Each CPU $\mathsf{rep}(b_i)$ fetches memory from the database nodes down the path to leaf ℓ_i; when b_i is found, it copies its value v_i into local memory. Each dummy CPU chooses a random path and make analogous dummy data fetches along it, ignoring all read values. (Recall that simultaneous data *reads* do not yield conflicts).
4. **Remove Old Data:** For each level in the tree,
   - Aggregate instructions across CPUs accessing the same "buckets" of memory (corresponding to nodes of the tree) on the server side. Each representative CPU $\mathsf{rep}(b)$ begins with the instruction of "remove block b if it occurs" and dummy CPUs hold the empty instruction. (Aggregation is as before, but at bucket level instead of the block level).
   - For each bucket to be modified, the CPU with the *smallest* id from those who wish to modify it executes the aggregated block-removal instructions for the bucket. Note that this aggregation step is purely for correctness and not security.
5. **Insert Updated Data into Database in *Parallel*:** Run Route on inputs $\{(m, (\mathsf{msg}_i, \mathsf{addr}_i))\}_{i\in[m]}$, where for each $\mathsf{rep}(b_i)$, $\mathsf{msg}_i = (b_i, \bar{v}_i, \ell'_i)$ (i.e., updated block data) and $\mathsf{addr}_i = [\ell'_i]_{\log m}$ (i.e., level-$\log m$-truncation of the path ℓ'_i), and for each dummy CPU, $\mathsf{msg}_i, \mathsf{addr}_i = \emptyset$.
6. **Flush the ORAM Database:** In parallel, each CPU initiates an independent flush of the ORAM tree. (Recall that this corresponds to selecting a random path down the tree, and pushing all data blocks in this path as far as they will go). To implement the simultaneous flush commands, as before, commands are aggregated across CPUs for each bucket to be modified, and the CPU with the smallest id performs the corresponding aggregated set of

$\mathsf{UpdateBuckets}\,(m, (\mathsf{command}_i, \mathsf{path}_i))$
Let $\mathsf{path}^{(1)}, \mathsf{path}^{(2)}, \ldots, \mathsf{path}^{(\log n)}$ denote the bit prefixes of length 1 to $\log n$ of path.

For each level $\mathsf{lev} = 1, \ldots, \log n$ of the tree:

1. The CPUs invoke the oblivious aggregation procedure $\mathsf{OblivAgg}$ as follows.

   Case 1: $\mathsf{command}_i = \mathsf{remove}\text{-}b_i$.
   Each CPU i sets $\mathsf{key}_i = \mathsf{path}_i^{(\mathsf{lev})}$ and $\mathsf{data}_i = b_i$ if b_i is in the bucket of node $\ell_i^{(\mathsf{lev})}$, and $\mathsf{data}_i = \perp$ otherwise. Use the union function $\mathsf{Agg}(\{\mathsf{data}_i\}) = \{b : \exists \mathsf{data}_i = b\}$ as the aggregation function.

   Case 2: $\mathsf{command}_i = \mathsf{flush}$.
   Each CPU i sets $\mathsf{key}_i = \mathsf{path}_i^{(\mathsf{lev})}$ and $\mathsf{data}_i = L$ (resp., R) if the $(\mathsf{lev}+1)$-st bit of path_i is 0 (resp., 1). Use the union function as the aggregation function.

   At the conclusion of the protocol, each bucket $\mathsf{path}_i^{(\mathsf{lev})}$ is assigned to a representative CPU $\mathsf{bucket\text{-}rep}(\mathsf{path}_i^{(\mathsf{lev})})$ with aggregated commands $\mathsf{agg\text{-}command}_i$.

2. Each representative CPU performs the updates:

   If $i \neq \mathsf{bucket\text{-}rep}(\mathsf{path}_i^{(\mathsf{lev})})$, do nothing. Otherwise:

   Case 1: $\mathsf{command}_i = \mathsf{remove}\text{-}b_i$.
   Remove all blocks $b \in \mathsf{agg\text{-}command}_i$ in the bucket $\mathsf{path}_i^{(\mathsf{lev})}$ by accessing memory bucket $\mathsf{path}_i^{(\mathsf{lev})}$ and rewriting contents.

   Case 2: $\mathsf{command}_i = \mathsf{flush}$.
   Access memory buckets $\mathsf{path}_i^{(\mathsf{lev})}, \mathsf{path}_i^{(\mathsf{lev})}||0, \mathsf{path}_i^{(\mathsf{lev})}||1$, perform flush operation locally according to $\mathsf{agg\text{-}command}_i \subset \{L, R\}$, and write the contents back.
   Specifically, denote the collection of stored data blocks b in $\mathsf{path}_i^{(\mathsf{lev})}$ by $\mathsf{ThisBucket}$. Partition $\mathsf{ThisBucket} = \mathsf{ThisBucket\text{-}L} \cup \mathsf{ThisBucket\text{-}R}$ into those blocks whose associated leaves continue to the left or right (i.e., $\{b_j \in \mathsf{ThisBucket} : \bar{\ell}_j^{(\mathsf{lev}+1)} = \mathsf{mypath}^{(\mathsf{lev})}||0\}$, and similar for 1).
   - If $L \in \mathsf{agg\text{-}command}_i$, then set $\mathsf{ThisBucket} \leftarrow \mathsf{ThisBucket} \setminus \mathsf{ThisBucket\text{-}L}$, and insert data items $\mathsf{ThisBucket\text{-}L}$ into bucket $\mathsf{path}_i^{(\mathsf{lev})}||0$.
   - If $R \in \mathsf{agg\text{-}command}_i$, then set $\mathsf{ThisBucket} \leftarrow \mathsf{ThisBucket} \setminus \mathsf{ThisBucket\text{-}R}$, and insert data items $\mathsf{ThisBucket\text{-}L}$ into bucket $\mathsf{path}_i^{(\mathsf{lev})}||0$.

Figure 8: A space-efficient implementation of the $\mathsf{UpdateBuckets}$ procedure.

commands. (For example, all CPUs will wish to access the root node in their flush; the aggregation of all corresponding commands to the root node data will be executed by the lowest-numbered CPU who wishes to access this bucket, in this case CPU 1).

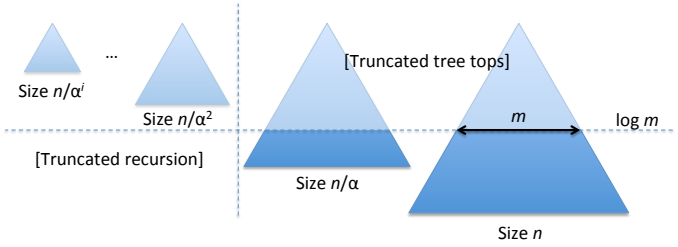
7. **Return Output:** Run $\mathsf{OblivMCast}$ on inputs $\{(b_i, v_i)\}_{i\in[m]}$ (where for dummy CPUs, $b_i, \bar{v}_i := \emptyset$) to communicate the *original* (pre-updated) value of each data block b_i to the subset of CPUs that originally requested it.

A few remarks regarding our construction.

Remark 3.13 (Truncating OPRAM for Fixed m)**.** In the case that the number of CPUs m is fixed and known a priori, the OPRAM construction can be directly trimmed in two places.

Trimming tops of recursive data trees: Note that data items are always inserted into the OPRAM trees at level $\log m$, and flushed down from this level. Thus, the top levels in the ORAM tree are *never utilized*. In such case, the data buckets in the corresponding tops of the trees, from the root node to level $\log m$ for this bound, can simply be removed without affecting the OPRAM.

Truncating recursion: In the t-th level of recursion, the corresponding database size shrinks to $n_t = n/\alpha^t$. In recursion level $\log_\alpha n/m$ (i.e., where $n_t = m$), we can then achieve oblivious data accesses via local CPU communication (storing each block $i \in [n_t] = [m]$ locally at CPU i, and running $\mathsf{OblivAgg}, \mathsf{OblivMCast}$ directly) without needing any tree lookups or further recursion.



Remark 3.14 (Collision-Freeness)**.** In the compiler above, CPUs only access the same memory address simultaneously in the (read-only) memory lookup in Step 3. However, a simple tweak to the protocol, replacing the direct memory lookups with an appropriate aggregation and multicast step (formally, the procedure $\mathsf{UpdateBuckets}$ as described in the appendix), yields collision freeness.

4 Garbled PRAM

As an application of OPRAM, we demonstrate a construction of *garbled parallel RAMs*. Specifically, we show that the IBE-based garbled RAM of Gentry *et al.* [GHL+14] (which in turn builds upon [LO13b]) can be directly generalized to garble PRAMs in a simple and modular way, given an OPRAM compiler with certain properties, and instead relying on 2-level hierarchical IBE. We then show (in the appendix) how to obtain these required properties generically from any OPRAM, and how to reduce the assumption from 2-HIBE back to IBE with a further modification of the scheme. We remark that constructions of garbled RAM can be obtained directly from one-way functions [GHL+14, GLOS15, GLO15], and leave as an interesting open problem how to extend these techniques to the PRAM setting.

We start by generalizing the notion of garbled RAM [LO13b, GHL+14] to garbled PRAM, where the main difference is that a PRAM program Π consists of m CPUs. We allow each CPU ℓ to take a short input x_ℓ, which can be thought of as the initial CPU state. We model the garbling algorithm and garbled program evaluator also as PRAMs, and aim to preserve the parallel runtime of Π.

4.1 Definition of Garbled PRAM

Following [GHL+14], we consider a scenario where an initial memory data mem_{init} is garbled once, and then multiple garbled PRAM programs can be executed in a fixed order with the memory changes persisting throughout executions. Our presentation here follows closely to [GHL+14].

Definition 4.1 (Garbled PRAM)**.** A garbled PRAM scheme consists of a tuple of procedures $(\mathsf{GData}, \mathsf{GProg}, \mathsf{GInput}, \mathsf{GEval})$ with the following syntax:

- $\tilde{\mathsf{mem}} \leftarrow \mathsf{GData}(\mathsf{mem}, k)$: Takes initial memory data $\mathsf{mem} \in \{0,1\}^n$ and a key k. Outputs the garbled data $\tilde{\mathsf{mem}}$.
- $(\tilde{\Pi}, k^{in}) \leftarrow \mathsf{GProg}(\Pi, k, n, t^{init}, t^{cur})$: Takes a key k and a description of a RAM program P with memory-size n and run-time consisting of t^{cur} parallel CPU timesteps. In the case of garbling multiple programs, we also provide t^{init} indicating the cumulative timesteps executed by all previous programs. Outputs a garbled program $\tilde{\Pi}$ and an input-garbling-key k^{in}.
- $\tilde{x} \leftarrow \mathsf{GInput}(x, k^{in})$: Takes an input vector $x = (x_1, \ldots, x_m)$ and input-garbling-key k^{in} and outputs a garbled input vector $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_m)$.
- $y = \mathsf{GEval}^{\tilde{\mathsf{mem}}}(\tilde{\Pi}, \tilde{x})$: Takes a garbled program $\tilde{\mathsf{mem}}$, garbled input vector $\tilde{x}$ and garbled memory data $\tilde{\mathsf{mem}}$ and computes the output $y = \Pi^D(x)$.

We model these procedures as PRAM programs, and require both GProg and GEval to preserve the *parallel runtime* of Π. We additionally require GData to be parallelizable across CPUs. On the other hand, the size of the garbled program may be proportional to the total time complexity of Π.

- (Parallel) Efficiency: The parallel runtime of GProg and GEval must be $|C^{\Pi}_{CPU}| \cdot t^{cur} \cdot \mathsf{polylog}(n) \cdot \mathsf{poly}(\kappa)$, where κ is the security parameter. The parallel time of GData must be $n/m \cdot \mathsf{polylog}(\kappa)$, where m is the number of processors of the PRAM program to evaluate GData.

The correctness and security requirements of garbled PRAM are identical to that of garbled RAM. Consider the following experiment: choose a key $k \leftarrow \{0,1\}^\kappa$, $\tilde{\mathsf{mem}} \leftarrow \mathsf{GData}(\mathsf{mem}, k)$ and for $i = 1, \ldots, s$: $(\tilde{\Pi}_i, k^{\in}_i) \leftarrow \mathsf{GProg}(P_i, n, t^{init}_i, t_i, k)$, and $\tilde{x}_i \leftarrow \mathsf{GInput}(x_i, k^{\in}_i)$, where $t^{init}_i = \sum_{j \in [i-1]} t_i$ denotes the parallel runtime of all programs prior to Π_i. Let

$$(y'_1, \ldots, y'_s) = (\mathsf{GEval}(\tilde{\Pi}_1, \tilde{x}_1), \ldots, \mathsf{GEval}(\tilde{\Pi}_s, \tilde{x}_s))^{\tilde{\mathsf{mem}}}$$

denote the output of evaluating the garbled programs sequentially over the garbled memory. We require

- Correctness: $\Pr[y'_i = y_i \forall i \in [s]] \geq 1 - \mu(\kappa)$, where $\mu(\cdot)$ is a negligible function.
- Security: There exists a universal simulator Sim such that

$$(\tilde{\mathsf{mem}}, \tilde{\Pi}_1, \ldots, \tilde{\Pi}_s, \tilde{x}_1, \ldots, \tilde{x}_s) \approx \mathsf{Sim}(1^\kappa, \{P_i, t_i, y_i\}_{i=1}^s, n).$$

We also define a weaker security notion of security with unprotected memory access (UMA) for garbled PRAM in an analogous way to [GHL+14], where the simulator is additionally given the initial memory data mem, as well as the history of memory access throughout the computation, including both access pattern and memory update content.

Definition 4.2 (UMA security)**.** Let $\mathsf{MemAccess} = \{(\mathsf{addr}^{\mathsf{Read}}_{j,\ell}, \mathsf{addr}^{\mathsf{Write}}_{j,\ell}, b^{\mathsf{Write}}_{j,\ell}) : j = 1, \ldots, t, \ell = 1, \ldots, m\}$ denote the output memory instruction of the CPU-step for each CPU and each timestep of the execution. A garbled PRAM has (weaker) security with *Unprotected Memory Accesses (UMA)* if in the place of the Security requirement in Definition 4.1, we require only existence of a universal simulator Sim such that

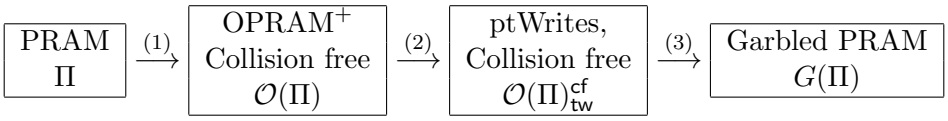
$$(\tilde{\mathsf{mem}}, \tilde{\Pi}_1, \ldots, \tilde{\Pi}_s, \tilde{x}_1, \ldots, \tilde{x}_s) \approx \mathsf{Sim}(1^\kappa, \{P_i, t_i, y_i\}_{i=1}^s, \mathsf{mem}, \mathsf{MemAccess}, n).$$

Finally, we also define a predictably timed writes (ptWrites) property for a PRAM program in an analogous way, which is used in intermediate steps of our garbled PRAM construction.

Definition 4.3 (ptWrites)**.** A collision-free PRAM program Π has *predictably timed writes (ptWrites)* if there exists a poly-size circuit $\mathsf{WriteTime}$ such that the following holds for any execution of Π, each timestep j, and each CPU ℓ. Let the input/outputs of the timestep of the CPU be $C^{\Pi}_{CPU}(\mathsf{state}_{j,\ell}, b^{\mathsf{Read}}_{j,\ell}) = (\mathsf{state}_{j+1,\ell}, \mathsf{addr}^{\mathsf{Read}}_{j,\ell}, \mathsf{addr}^{\mathsf{Write}}_{j,\ell}, b^{\mathsf{Write}}_{j,\ell})$. Then, $u = \mathsf{WriteTime}(j, \ell, \mathsf{state}_{j,\ell}, \mathsf{addr}^{\mathsf{Read}}_{j,\ell})$ is the largest value of $u < j$ such that some CPU ℓ wrote to location $\mathsf{addr}^{\mathsf{Read}}_{j,\ell}$ at timestep u.

4.2 Overview of GPRAM Construction

At a high level, our construction takes the following form, analogous to the steps in [GHL+14]:



Step (1) is a collision-free "OPRAM$^+$" compiler, consisting of standard collision-free OPRAM (as defined and constructed in the previous sections), together with a simple layer of symmetric-key encryption (SKE). Namely, data is encrypted using the SKE under a random key k that is hardcoded into each CPU. Within each CPU step, read values are decrypted before computation, and values to write are first encrypted (both using k). Thus, OPRAM$^+$ hides both the memory content and memory access patterns of the original program (i.e., these values can be simulated).

Step (2) converts a collision-free OPRAM$^+$ compiled program to one with predictably timed writes (ptWrites), while preserving the above simulation property and collision freeness. This can be achieved by combining the generic transformation of [GHL+14] to obtain the ptWrites property, together with techniques developed in our OPRAM construction in order to regain collision-freeness. We elaborate on this transformation in Section 4.3.

Step (3) converts any collision-free, ptWrites PRAM into one with security with unprotected memory access (UMA), as defined by [GHL+14] (i.e., that it leaks only memory access patterns and database memory contents). When composed with the initial OPRAM$^+$ compiler, this yields *full* GPRAM security. We elaborate on this transformation in Section 4.4

4.3 Obtaining ptWrites *and* Collision-Freeness

As part of their garbled RAM construction, Gentry *et al.* [GHL+14] provide a general transformation taking any (sequential) RAM program generically to an equivalent one with the predictably timed writes (ptWrites) property. The same approach may be taken in the PRAM setting, but direct application of the [GHL+14] transformation introduces access collisions across CPUs, which

will be a problem when attempting to garble the program (see Section 4.4). To avoid this, we will apply a second transformation on top that removes these collisions. We make use of the specific structure of the [GHL+14] transformation (and assume that we begin with an original PRAM that is collision-free).

The [GHL+14] transformation applies a binary tree on top of the original data, where the leaves of the tree correspond to the bits of the original data, and each internal node of the tree contains the last write time of each of its two children (initially, these are all set to 0). For each memory access request to location addr in the original data, the transformed program will now access the path of nodes in the tree down the path to addr. The last-write time of the root corresponds to the total number of write operations performed so far, which can be stored within the state of the computation. Whenever a node is accessed, we temporarily remember the last-write-time of its children (by keeping this info in the state). This ensures that before the contents of any node in the tree are read (including the actual data at the leaves), its last-write time is known. To write to some location in the original data, the same procedure is followed as in the case of a read, but after the values in each node are read, we also increment the last-write-time for the corresponding child on the path to the leaf. The resulting overhead is $\log n$ (for data size n), since each memory access is now performed by accessing a $\log n$-depth path in the tree.

Now, consider imposing the same tree structure in the PRAM setting. If the original PRAM was collision free, then no two CPUs will wish to access the same leaf node addr in any step. However, they will necessarily have collisions in nodes higher in the tree.

We observe that this is precisely the same scenario as faced in our OPRAM construction, when CPUs wished to simultaneously update distinct paths within the ORAM data tree. To address this, we provided the $\mathsf{UpdateBuckets}$ procedure (Figure 8), where for each level of the tree, the CPUs perform a "coordination stage" (communicating through coordinated reads/writes in the memory) in which for every collision the CPU with the smallest id receives and implements the aggregated collection of instructions to take place at the target memory address. This same procedure will also successfully remove collisions in the ptWrites-compiled PRAM program. The resulting overhead is a factor of $\log m$, since in each level of the tree (i.e., each memory access in the ptWrites-compiled program), the m CPUs must execute the $O(\log m)$-cost $\mathsf{UpdateBuckets}$ procedure.

Theorem 4.4 (ptWrites and Collision Freeness)**.** *There exists an efficient compiler taking any collision-free PRAM* Π *to a functionally equivalent collision-free PRAM* Π' *with ptWrites, with both total and parallel overhead* $\mathsf{polylog}(n)$*. If the original program has simulatable data values and access patterns, then this property is preserved.*

4.4 Obtaining UMA-Secure GPRAM

We first describe the construction of [GHL+14] for obtaining UMA-secure garbled RAM, and then present our analogous construction for garbled PRAM.

Construction of [GHL+14] and its UMA Security. Let Π be a RAM program with ptWrites property. Π is represented as a small CPU-step circuit C^{Π}_{CPU} which executes each single CPU step. Namely, C^{Π}_{CPU} on input a CPU state state and a read-bit b^{Read} read from memory, outputs an updated CPU state state', a next read location $\mathsf{addr}^{\mathsf{Read}}$, write location $\mathsf{addr}^{\mathsf{Write}}$, and a bit value b^{Write} to write to $\mathsf{addr}^{\mathsf{Write}}$. The RAM computation is done by iteratively applying C^{Π}_{CPU} for t

steps starting with some initial state state_{init} and memory mem_{init}, where t is the runtime of the computation.

At a high level, Π is garbled by garbling t copies of some "augmented" CPU-step circuit $C^{\Pi}_{CPU^+}$. The garbled CPU-step circuits can pass the CPU state securely from one to the next by identifying the state output wire labels of j-th garbled circuit $\tilde{C}^{\Pi}_{CPU^+}(j)$ to the state input wire labels of $j+1$-st garbled circuit $\tilde{C}^{\Pi}_{CPU^+}(j+1)$. The main issue is to handle memory accesses, which deal with runtime information that cannot be determined in the compile time (i.e., the time to generate garbled program). To handle a memory read, $\tilde{C}^{\Pi}_{CPU^+}(j)$ outputs some translation information $\mathsf{translate}$, which together with the garbled memory bit $\tilde{\mathsf{mem}}[\mathsf{addr}^{\mathsf{Read}}]$ allows the evaluator to compute the input label of the read-bit b^{Read} of the next garbled circuit $\tilde{C}^{\Pi}_{CPU^+}(j+1)$. To do so, $\tilde{C}^{\Pi}_{CPU^+}(j)$ has both labels label_0 and label_1 of $\tilde{C}^{\Pi}_{CPU^+}(j+1)$'s b^{Read} wire hard-wired in, the translation information $\mathsf{translate}$ consists of encryptions $(\mathsf{ct}_0, \mathsf{ct}_1)$ of the labels $(\mathsf{label}_0, \mathsf{label}_1)$ under distinct keys, and the garbled memory bit $\tilde{\mathsf{mem}}[\mathsf{addr}^{\mathsf{Read}}]$ stores exactly one of the secret key sk_b that allows the evaluator to decrypt label_b of the bit value of $\mathsf{addr}^{\mathsf{Read}}$ while hides the other $\mathsf{label}_{\bar{b}}$. In this way, writing to memory corresponds to outputting an appropriate version of a secret key.

A subtle circularity issue may arise in the above strategy. Roughly speaking, the security of garbled circuits relies on one label for each input wire being hidden, which in ture relies on the semantic security of the encryption scheme. However, to update memory, $\tilde{C}^{\Pi}_{CPU^+}(j)$ needs to have the ability to produce secret keys (say, with some master secret key hard-wired in). Thus, semantic security in turn relies on the security of the garbled circuits. To cope with circularity, [GHL+14] relies on a weaker "bounded" variant of 2-level hierachical IBE (HIBE) called timed IBE (TIBE), which can be constructed based on any regular IBE. Here, we present their construction based on 2-level HIBE, which enables a clean generalization to the parallel setting. We discuss how to modify the construction to reduce the assumption back down to IBE in the end of this section.

Specifically, in the [GHL+14] construction, a garbled memory bit $\tilde{\mathsf{mem}}[\mathsf{addr}]$ is a secret key $\mathsf{sk}_{\mathsf{id}}$ of a 2-level HIBE scheme with identity $\mathsf{id} = (\mathsf{lwtime}, (\mathsf{addr}, b))$ indexed by the timestep lwtime that the memory is updated, the address addr of the memory, and the bit value b stored in the memory. Here, the first-level id is the timestep lwtime, and the second-level id is (addr, b). To compute the translation information $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$ for read location $\mathsf{addr}^{\mathsf{Read}}$, $\tilde{C}^{\Pi}_{CPU^+}(j)$ relies on the ptWrites property to compute the last access time lwtime of $\mathsf{addr}^{\mathsf{Read}}$, and encrypts each label_b (which are both hardwired) with identity $(\mathsf{lwtime}, \mathsf{addr}^{\mathsf{Read}}, b)$ using the hardwired master public key mpk. Namely, $\mathsf{ct}_b = \mathrm{Enc}_{\mathsf{mpk}}(\mathsf{label}_b, (j, (\mathsf{addr}^{\mathsf{Read}}, b)))$ for each $b \in \{0, 1\}$. To write b^{Write} to memory location $\mathsf{addr}^{\mathsf{Write}}$, a delegation key dsk_j is hard-wired in $\tilde{C}^{\Pi}_{CPU^+}(j)$ to generate $\mathsf{sk}_{\mathsf{id}}$ with identity $\mathsf{id} = (j, (\mathsf{addr}^{\mathsf{Write}}, b^{\mathsf{Write}}))$. The initial memory content is garbled using timestep $\mathsf{lwtime} = 0$, namely, $\tilde{\mathsf{mem}}_{init}[\mathsf{addr}] = \mathsf{sk}_{(0,(\mathsf{addr},\mathsf{mem}_{init}[\mathsf{addr}]))}$.

At a high level, the UMA security is proved by a sequence of hybirds that "erase" the computation by "erasing" the garbled circuits and "unused" input labels step by step "*forward in time*." Namely, starting from the real garbled program, the first garbled CPU-step circuit $\tilde{C}^{\Pi}_{CPU^+}(j=1)$ is "erased" by replacing it with a simulated one, then the unused input lables of the second garbled CPU-step circuit $\tilde{C}^{\Pi}_{CPU^+}(2)$ are "erased" by replacing ciphertexts in $\mathsf{translate}$ with encryption of dummy, then $\tilde{C}^{\Pi}_{CPU^+}(2)$ is erased, and so on. This can be done since each time when we want to erase a garbled circuit $\tilde{C}^{\Pi}_{CPU^+}(j)$, we are in a hybrid where one of each input wire labels is information theoretically erased, and when we want to erase the "unused" input label for $\tilde{C}^{\Pi}_{CPU^+}(j+1)$, the ciphertexts are encrypted with timestep $\leq j$ and all "future" garbled circuits only contain

delegation keys with timestep $\geq j+1$. At the end, this leads to a hybrid that can be simulated by a UMA simulator who is given the inital memory content as well as complete history of memory access.

Our Construction of UMA-Secure Garbled PRAM. Let Π be a PRAM program with ptWrites and collision free property with CPU-step circuit C^{Π}_{CPU} (recall that all CPUs have the same CPU program with the same input/output interface as the RAM program). We show that the above construction of [GHL+14] can be directly generalized to garble Π with UMA security.

For each timestep j, instead of one garbled CPU-step circuit $\tilde{C}^{\Pi}_{CPU^+}(j)$, we generate m garbled CPU-step circuits $\tilde{C}^{\Pi}_{CPU^+}(j,\ell)$, one for each CPU $\ell \in [m]$. As before, each garbled CPU-step circuit $\tilde{C}^{\Pi}_{CPU^+}(j,\ell)$ can pass its CPU state securely to the next timestep $\tilde{C}^{\Pi}_{CPU^+}(j+1,\ell)$ by identifying wire labels. Also, thanks to the collision free and ptWrites property, memory access can be handled in the same way. Namely, each garbled memory bit $\tilde{\mathsf{mem}}[\mathsf{addr}]$ is a secret key $\mathsf{sk}_{\mathsf{id}}$ with identity $\mathsf{id} = (\mathsf{lwtime}, (\mathsf{addr}, b))$. To handle memory read, $\tilde{C}^{\Pi}_{CPU^+}(j,\ell)$ computes $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$ with $\mathsf{ct}_b = \mathrm{Enc}_{\mathsf{mpk}}(\mathsf{label}_b, (j, (\mathsf{addr}^{\mathsf{Read}}, b)))$ using hard-wirded master public key mpk. To handle memory write, delegation key dsk_j is hard-wired in $\tilde{C}^{\Pi}_{CPU^+}(j,\ell)$ to generate $\mathsf{sk}_{\mathsf{id}}$ with identity $\mathsf{id} = (j, (\mathsf{addr}^{\mathsf{Write}}, b^{\mathsf{Write}}))$. Note that all CPUs handle memory access in the same way, independent of their CPU id ℓ.

Note that ptWrites of the original program Π is required for correctness of the scheme (as in [GHL+14]), and that *collision-freeness* is crucial for *security*. Indeed, if there is a write collision of two CPUs in the same timestep with different write values, this would release both 0 and 1 HIBE secret keys for the corresponding memory address in the GPRAM evaluation, which will decrypt *both the 0 and 1* input labels for the next garbled circuit who reads in this value.

- GData: The initial memory content is garbled using timestep $\mathsf{lwtime} = 0$ as before. Namely, for each location addr, the corresponding garbled data bit will be a 2-HIBE secret key corresponding to identity $(0, (\mathsf{addr}, \mathsf{mem}_{init}[\mathsf{addr}]))$: i.e., $\tilde{\mathsf{mem}}_{init}[\mathsf{addr}] = \mathsf{sk}_{(0,(\mathsf{addr},\mathsf{mem}_{init}[\mathsf{addr}]))}$.
- GProg: Generate m garbled CPU-step circuits $\tilde{C}^{\Pi}_{CPU^+}(j,\ell)$, one for each CPU $\ell \in [m]$. This is done in reverse chronological order: for each timestep j, the resulting garbled input wire labels are hardcoded into the circuit to be garbled for timestep $j-1$. In addition, within each circuit for timestep j is hardcoded the 2-HIBE public key pk and a delegated secret key sk_j that can generate 2-HIBE secret keys for ids $(j,\cdot)$.
- GInput: Initial CPU state inputs are garbled simply as the corresponding input labels to the m garbled circuits at initial timestep 1.
- GEval: Evaluate the m garbled circuits *in parallel* for each timestep, mimicking [GHL+14].

Theorem 4.5. *Suppose Π is an m-processor PRAM with collision-free accesses and ptWrites. Then, assuming the existence of 2-HIBE, the procedures $(\mathsf{GData}, \mathsf{GProg}, \mathsf{GInput}, \mathsf{GEval})$ described above yield a UMA-secure garbled PRAM of size $O(\mathsf{poly}(\kappa) \cdot m \cdot \mathsf{time}(\Pi))$ and (parallel) evaluation time $O(\mathsf{poly}(\kappa) \cdot \mathsf{time}(\Pi))$.*

Sketch of proof. The size of the garbled PRAM corresponds to $m \cdot \mathsf{time}(\Pi)$ garbled CPU-step circuits, which may be garbled in parallel time comparable to $\mathsf{time}(\Pi)$ (since CPU-step circuits for the m CPUs may be garbled in parallel for each timestep).

The proof of security mirrors that of [GHL+14], except that we now "erase" garbled circuits and unused input labels along *two dimensions*: first removing each garbled CPU circuit one by one

for a given time step j, and then progressing "forward in time" to the next time $(j+1)$. This can be done since each time when we want to erase a garbled circuit $\tilde{C}^{\Pi}_{CPU}(j,\ell)$, we are in a hybrid where one of each input wire labels is information theoretically erased, and when we want to erase the "unused" input label for $\tilde{C}^{\Pi}_{CPU}(j+1,\ell)$, the ciphertexts are encrypted with timestep $\leq j$ and all "future" garbled circuits only contains delegation keys with timestep $\geq j+1$.

□

4.5 GPRAM: Combining the Steps

Combining Theorems 4.4 and 4.5, we obtain UMA-secure GPRAM for any PRAM, assuming 2-HIBE. It remains to combine this with the original collision-free OPRAM$^+$step to obtain full security, and to reduce the 2-HIBE assumption back down to IBE.

Full Security. Recall that UMA security leaks the patterns of memory accesses, and the values held in memory during the GPRAM evaluation. Following the generic transformation of [GHL$^+$14] (which generalized to the parallel setting directly), when the PRAM program Π is first compiled by OPRAM$^+$compiler (as discussed in Section 4.2), then the final garbled PRAM program $G(\Pi)$ is *fully* secure. Indeed, the OPRAM$^+$compiler precisely guarantees that these values can be simulated without any secret information: OPRAM for the memory access patterns, and the extra layer of SKE for the values stored in memory.

Reducing the Assumption to IBE. Note that the above construction of garbled PRAM relies on 2-level HIBE, instead of the timed IBE (TIBE) used in [GHL$^+$14]. The reason is that in TIBE, each delegation key dsk_j can only be used to generate a single secret key $\mathsf{sk}_{\mathsf{id}}$ with $\mathsf{id} = (j,\cdot)$ (otherwise, the security breaks down). Namely, TIBE can be viewed as 2-level HIBE with "1-bounded key delegation security." In the PRAM setting, the m CPUs may perform parallel writes (to distinct locations) in a timestep j, which create m secret keys with the same j. To overcome this problem, simply "generalize" a timestep to consist of pair (t,i) corresponding to an actual timestep t and a processor i (and record these generalized timesteps in the ptWritestree); this means that generalize timesteps are only used once and TIBE suffices.

References

[Ajt10] Miklós Ajtai, *Oblivious rams without cryptogarphic assumptions*, STOC, 2010, pp. 181–190.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi, *An 0(n log n) sorting network*, Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '83, ACM, 1983, pp. 1–9.

[Bat68] K. E. Batcher, *Sorting networks and their applications*, Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference (New York, NY, USA), AFIPS '68 (Spring), ACM, 1968, pp. 307–314.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass, *Large-scale secure computation: Multi-party computation for (parallel) RAM programs*, CRYPTO, 2015.

[BGL+15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang, *Succinct randomized encodings and their applications*, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, 2015, pp. 439–448.

[CCC+15] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou, *Computation-trace indistinguishability obfuscation and its applications*, Cryptology ePrint Archive, Report 2015/406, 2015.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan, *Succinct garbling and indistinguishability obfuscation for RAM programs*, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, 2015, pp. 429–437.

[CLP14] Kai-Min Chung, Zhenming Liu, and Rafael Pass, *Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead*, Advances in Cryptology - ASIACRYPT 2014, 2014, pp. 62–81.

[CLT15] Binyi Chen, Huijia Lin, and Stefano Tessaro, *Oblivious parallel ram: Improved efficiency and generic constructions*, Cryptology ePrint Archive, 2015.

[CP13] Kai-Min Chung and Rafael Pass, *A simple ORAM*, Cryptology ePrint Archive, Report 2013/243, 2013.

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

[FDD12] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas, *A secure processor architecture for encrypted computation on untrusted programs*, Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing (New York, NY, USA), STC '12, ACM, 2012, pp. 3–8.

[GGH+13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs, *Optimizing oram and using it efficiently for secure computation*, Privacy Enhancing Technologies, 2013, pp. 1–18.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs, *Garbled RAM revisited*, Advances in Cryptology - EUROCRYPT 2014, 2014, pp. 405–422.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs, *Outsourcing private RAM computation*, Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014, 2014, pp. 404–413.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis, *Secure two-party computation in sublinear (amortized) time*, the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012, 2012, pp. 513–524.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich, *How to run turing machines on encrypted data*, CRYPTO (2), 2013, pp. 536–553.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky, *Black-box garbled RAM*, Electronic Colloquium on Computational Complexity (ECCC) **22** (2015), 56.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro, *Garbled RAM from one-way functions*, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, 2015, pp. 449–458.

[GM11] Michael T. Goodrich and Michael Mitzenmacher, *Privacy-preserving access of outsourced data via oblivious RAM simulation*, Automata, Languages and Programming - 38th International Colloquium, ICALP, 2011, pp. 576–587.

[GMOT11] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia, *Oblivious ram simulation with efficient worst-case access overhead*, CCSW, 2011, pp. 95–100.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson, *How to play any mental game or a completeness theorem for protocols with honest majority*, STOC, 1987, pp. 218–229.

[GO96] Oded Goldreich and Rafail Ostrovsky, *Software protection and simulation on oblivious RAMs*, J. ACM **43** (1996), no. 3, 431–473.

[Gol87] Oded Goldreich, *Towards a theory of software protection and simulation by oblivious RAMs*, STOC, 1987, pp. 182–194.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky, *On the (in)security of hash-based oblivious ram and a new balancing scheme*, SODA, 2012, pp. 143–156.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters, *Indistinguishability obfuscation for turing machines with unbounded memory*, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015, 2015, pp. 419–428.

[LO13a] Steve Lu and Rafail Ostrovsky, *Distributed oblivious ram for secure two-party computation*, TCC, 2013, pp. 377–396.

[LO13b] ———, *How to garble RAM programs*, Advances in Cryptology - EUROCRYPT 2013, 2013, pp. 719–734.

[LPM+13] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman, *Shroud: ensuring private access to large-scale data in the data center.*, FAST, 2013, pp. 199–214.

[NWI+15] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi, *GraphSC: Parallel secure computation made easy*, IEEE Symposium on Security and Privacy (S&P), 2015.

[OS97] Rafail Ostrovsky and Victor Shoup, *Private information storage (extended abstract)*, STOC, 1997, pp. 294–303.

[PF79] Nicholas Pippenger and Michael J. Fischer, *Relations among complexity measures*, J. ACM **26** (1979), no. 2, 361–381.

[RFK+14] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas, *Ring ORAM: closing the gap between small and large client storage oblivious RAM*, IACR Cryptology ePrint Archive **2014** (2014), 997.

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

[SS13] Emil Stefanov and Elaine Shi, *ObliviStore: High performance oblivious cloud storage*, IEEE Symposium on Security and Privacy, 2013, pp. 253–267.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas, *Path ORAM: an extremely simple oblivious RAM*

protocol, ACM Conference on Computer and Communications Security, 2013, pp. 299–310.

[WCS14] Xiao Shaun Wang, T.-H. Hubert Chan, and Elaine Shi, *Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound*, IACR Cryptology ePrint Archive **2014** (2014), 672.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi, *SCORAM: oblivious RAM for secure computation*, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014, pp. 191–202.

[WST12] Peter Williams, Radu Sion, and Alin Tomescu, *PrivateFS: A parallel oblivious file system*, Proceedings of the 2012 ACM Conference on Computer and Communications Security (New York, NY, USA), CCS '12, ACM, 2012, pp. 977–988.

[Yao82] Andrew Chi-Chih Yao, *Protocols for secure computations (extended abstract)*, 23rd Annual Symposium on Foundations of Computer Science (FOCS), 1982, pp. 160–164.