Is There an Oblivious RAM Lower Bound?

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

Moni Naor†
Weizmann Institute of Science
moni.naor@weizmann.ac.il

September 7, 2015

Abstract

An Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky (JACM 1996), is a (probabilistic) RAM that hides its access pattern, i.e. for every input the observed locations accessed are similarly distributed. Great progress has been made in recent years in minimizing the overhead of ORAM constructions, with the goal of obtaining *the smallest* overhead possible.

We revisit the lower bound on the overhead required to obliviously simulate programs, due to Goldreich and Ostrovsky. While the lower bound is fairly general, including the offline case, when the simulator is given the reads and writes ahead of time, it does assume that the simulator behaves in a "balls and bins" fashion. That is, the simulator must act by shuffling data items around, and is not allowed to have sophisticated encoding of the data.

We prove that for the *offline* case, showing a lower bound without the above restriction is related to the size of the circuits for sorting. Our proof is constructive, and uses a bit-slicing approach which manipulates the bit representations of data in the simulation. This implies that without obtaining yet unknown superlinear lower bounds on the size of such circuits, we cannot hope to get lower bounds on offline (unrestricted) ORAMs.

*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.

†Weizmann Institute of Science. Incumbent of the Judith Kleeman Professorial Chair. Research supported in part by grants from the Israel Science Foundation, BSF and Israeli Ministry of Science and Technology and from the I-CORE Program of the Planning and Budgeting Committee and the Israel Science Foundation (grant No. 4/11).

‡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

An *Oblivious RAM* (ORAM), introduced by Goldreich and Ostrovsky [Gol87, GO96] is a (probabilistic) RAM machine whose memory accesses do not reveal anything about the input—including both program and data—on which it is executed. More specifically, for any two inputs (Π_1, x_1) and (Π_2, x_2) with an equal number of memory accesses, the resulting distributions of accessed memory locations is the same, or similar.

Since their inception, oblivious RAM machines (more specifically, simulations of RAM on oblivious RAMs) have become an invaluable tool in designing cryptographic systems, where observable memory access patterns crucially must not leak sensitive information. This arises in the context of software protection (already in the original work by [GO96]), secure computation protocols utilizing the random access nature of computation [OS97, DMN11, GKK$^+$12, LO13, GGH$^+$13, WHC$^+$14, BCP15], building secure hardware with untrusted memory [FDD12], outsourcing data [SS13], protection against cache attacks [Nao09], further server-delegation scenarios (e.g., [CKW13, GHRW14]), and much more.

One can trivially simulate a RAM program by an oblivious one by simply replacing each data access with a scan of the entire memory. To be useful, an ORAM simulation should only introduce a small overhead. The primary metric analyzed is the overhead in *bandwidth*: that is, how many data items must be accessed in the oblivious simulation as compared to the original. A great deal of research has gone toward simplifying and optimizing the efficiency of ORAM constructions (e.g., [GO96, Ajt10, DMN11, GM11, GMOT11, KLO12, CP13, CLP14, GGH$^+$13, SvDS$^+$13, CLP14, WHC$^+$14, RFK$^+$14, WCS14]), with a clear goal of obtaining *the smallest* overhead possible.

This spurs the immediate question: *what is the best ORAM overhead possible?* The original Goldriech-Ostrovsky constructions [GO96] incurred multiplicative overhead $\mathsf{polylog}(n)$ for data size n. After years of progress, the most asymptotically optimized constructions to date achieve overhead $\Omega(\log n)$ for particular choices of block sizes [WCS14]. How much further, if at all, can this be pushed?

The lower bound of [GO96]. Presumably, the answer to this question is widely known. In the original work introducing ORAMs, Goldreich and Ostrovsky [GO96] also showed a $\Omega(\log n)$ lower bound for ORAM overhead, for data size n. This has been described as:

- *"In their seminal work [GO96], Goldreich and Ostrovsky showed that an ORAM of n blocks must incur a $O(\log n)$ lower bound in bandwidth blowup, under $O(1)$ blocks of client storage."* [DvDF$^+$15]
- *"[M]emory bandwidth is always a bottleneck of ORAM. All ORAMs are subject to a $O(\log n)$ lower bound on memory bandwidth overhead [Gol87, GO96], where n is the number of blocks in ORAM."* [Ren14]
- *"Even if new methods for oblivious RAM are discovered, there is an* inherent limit *to how much these schemes can be improved. It was shown in the original work of Goldreich and Ostrovsky [GO96] that there is a lower bound for oblivious RAM in this model.*
  Theorem ([GO96], Theorem 6): To obliviously perform n queries using only $O(1)$ client memory, there is a lower bound of $O(\log n)$ amortized overhead per access." [LO13]
- *"...due to the lower bound by Goldreich and Ostrovsky, any ORAM scheme must have bandwidth cost $\Omega(\beta \log n)$ to read or write a block of β bits.* [AKST14]

As noted in [GO96, WHC+14], the [GO96] lower bound is very powerful, applying also for the offline case (where all the accesses are given in advance), for arbitrary block sizes, for several relevant overhead metrics, and even when tolerating up to $O(1)$ statistical failure probability. E.g.,

- *"This is almost optimal since the well-known logarithmic ORAM lower bound [GO96] is immediately applicable to the circuit size metric as well."* [WCS14].

Altogether, the solidity of the $\Omega(\log n)$ barrier would seem to be inescapable. Or is it?

Reexamining the [GO96] bound. As is well recognized, the Goldreich-Ostrovsky work [GO96] provided a seminal foundation for understanding ORAM and its restrictions. Upon closer observation, however, one begins to see that the lower bound of [GO96] is not the end of the story. Despite being broadly interpreted as a hard lower bound, applying to all scenarios, the [GO96] bound actually bears significant limitations.

"Balls and bins" storage. Perhaps most surprising, the [GO96] lower bound is within a very restricted model of *"balls and bins"* data manipulation. Namely, the n data items are modeled as "balls," CPU registers and server-side data storage locations are modeled as "bins," and the set of allowed data operations consists *only* of moving balls between bins.

This immediately precludes any ORAM construction approach making use of data encoding, leveraging alternative representations of information, or any other form of non-black-box data manipulation. Such techniques have been shown to surpass performance of analogous "black-box" approaches in several related tasks within computer science, such as improving overhead in distributed file sharing, and optimizing network throughput via network coding (e.g., [NR95, MS11]). It is not clear whether the $\Omega(\log n)$ bound extends at all once these strong restrictions are lifted, and in light of our work this is not going to be simple to show.

Statistical security. The bound applies to ORAMs with *statistical* security: i.e., where the distribution of access patterns for two different inputs are statistically close. This statistical relation is crucial for the proof approach to proceed.

However, in many cases statistical guarantees may be stronger than necessary. Interestingly enough, the constructions presented within the same original ORAM paper [GO96]—and in fact, all ORAM constructions for the following 15 years, until the works of Ajtai [Ajt10] and Damgard *et al.* [DMN11] in 2010—were *not* statistically secure. Rather, due to use of pseudorandom functions and related tools, they guaranteed only that the distributions of memory accesses were *computationally* indistinguishable. Whether such constructions could bypass the $\Omega(\log n)$ bound is unknown.

1.1 Our results

In this work, we further explore the [GO96] lower bound, its extensions, and its limitations. As our main technical contribution, we provide evidence that the [GO96] lower bound does *not* extend directly beyond the "balls and bins" storage model (in the offline case)[1], or at least that such an assertion will require developing dramatically new techniques.

Think of a RAM machine that has an external memory of size n words, each of length w bits (where $\log n \leq w \ll n$). Loosely speaking, an *offline* oblivious simulation RAM' of a RAM machine guarantees obliviousness of memory accesses only for inputs $y_i = (\Pi_i, x_i)$ (consisting of

[1] Recall that offline ORAM corresponds to answering a sequence of requests all specified at once.

program and data) for which the program Π_i specifies its desired memory access instructions up front, within its description. We demonstrate that general logarithmic lower bounds $\Omega(\log n)$ on the overhead of offline ORAM compilers—as is implied by the [GO96] lower bound within the "balls and bins" setting—would directly imply *new circuit lower bounds*. Our proof is constructive: We show that the existence of n-word sorting circuits of size $o(\log n)$ times linear (i.e., $o(nw \log n)$ gates, for word size w) yields secure offline ORAM with sub-logarithmic overhead. While simple $\Omega(n \log n)$ lower bounds are known on the complexity of *comparator-gate* sorting circuits (sorting networks), in which data items can only be swapped as whole entities, no such lower bounds exist for the case of Boolean circuits which may further utilize the bit representation of the data being sorted.[2] In fact, in the RAM model one can obtain *near-linear* $O(n \log \log n)$ complexity sorting algorithms [AHNR95, Han04]; it is not known whether these algorithms can lead to small sorting circuits.

Theorem 1.1 (Informal)**.** *Suppose there exists a Boolean circuit family for sorting n words of size w-bits with size $o(nw \log n)$. Then there exists an offline ORAM compiler for $O(1)$ CPU registers, with bandwidth overhead $o(\log n)$. The oblivious simulation uses only* public *randomness to hide its access patterns.*

We remark that sorting networks appear frequently as tools within existing ORAM constructions, dating back to the original works [Gol87, GO96]. Our result can be interpreted as observing that, for the offline case, sorting is essentially *all* you need; and, further, that one need not restrict themselves to circuits with this special comparator-gate structure.

Our offline ORAM construction makes heavy "non-black-box" use of data storage and manipulation, violating the balls and bins restriction of the [GO96] bound. For example, our ORAM-compiled CPU will "re-pack" words to be stored on the server side such that a single word will contain bits of information from several data items. We do not assume much on the computational power of the (compiled) CPU, except that it be able to perform bitwise logical operations and be able to extract parts of a word.

Aside from offline ORAM, our construction also obliviously simulates a different restricted class of RAM programs: those that do not necessarily contain their access instructions in explicit form (in contrast to the offline setting), but which can be *heavily parallelized*. Namely, we can provide oblivious simulation of *Parallel* RAM (PRAM) of the CRCW-Priority variety (see [KR88, Vis15] for a survey) that has n processors and memory of size n. The overhead in simulating such PRAM programs is as above, yielding an improved complexity for this special highly-parallel case.[3] Intuitively, the offline and the PRAM cases fall within the same general framework with respect to our techniques, where in the offline case the explicit specification of access instructions allows us to parallelize the oblivious simulation over time. We elaborate on this topic in Section 3.3.

In Appendix A, we include a complete restatement and proof of the Goldreich-Ostrovksy [GO96] lower bound, together with a collection of specific extensions. For example, the bound's strict balls and bins data storage model can be mildly relaxed, allowing the CPU to copy and delete balls, and to also write non-data information in "bins" (but cannot output such information). Note that several recent positive results in ORAM make use of the latter technique, storing "helper" information in memory unrelated to the data values themselves, used to locate where within memory the data is stored.

[2] An $O(n \log n)$ sorting network (such as the famed AKS [AKS83] or the most recent work of Goodrich [Goo14]), translates to a Boolean circuit of size $O(nw \log n)$ gates.

[3] Note, however, the resulting system is still a sequential RAM, and not an Oblivious Parallel RAM as in [BCP14].

Additionally, in Section 1.2, we highlight several key research questions that remain open. In this context, we propose additional notions of ORAM security (both stronger and weaker) within which we can hope to either prove full lower bounds, or to extend our circuit-based upper bounds.

Technical Overview. Our offline ORAM compiler construction proceeds in two primary steps.

Step 1: Sorting Circuit → Oblivious-Access Sort. We first demonstrate how to use the structure of the given sorting circuit to obtain an efficient *Oblivious-Access Sorting* algorithm: that is, a (randomized) RAM program for sorting the values within an n-word database, where the distribution of access patterns is statistically close over any two choices of the input database (analogous to the definition of obliviousness in ORAM).

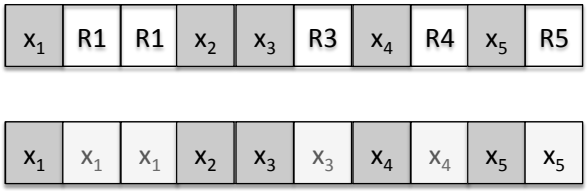
The challenge is in making this transformation tight. We can of course immediately obtain an oblivious algorithm, by directly emulating the circuit structure with the RAM (i.e., for each Boolean gate, read into memory the 2 words in which the desired bits reside, evaluate the gate on the bits, and write the result back to memory). However, unless the Boolean circuit has a very specific "word-preserving" structure, for which all bits within a single word are operated on at the same time, this approach will generically incur overhead equal to the word size w—since we are stuck reading an entire word just to operate on a single bit. In our solution, we show how to avoid this w multiplicative overhead, suffering only an additive term comparable to $\log w$.

Two important ideas we employ in our solution are: (i) a bit-slicing/SIMD approach where we utilize the inherent w parallelism of a CPU with words of size w in order to simulate w circuits in parallel. This approach was used by Biham in 1997 [Bih97] in order to speed up software implementations of DES. To do so, we make use of efficient (recursive) algorithms for *transposing* data.[4] The main issue with using this idea is to get w independent problems. (ii) When we randomly split the data into w parts, and sort each one separately, then for each element we have a pretty good idea where its location in the full sorted list should be, up to $\sqrt{n}$ accuracy. We can then refine this almost-sorted via a new set of w parallel sorts, this time independently within local regions.

Step 2: Oblivious-Access Sort → Offline ORAM. Next, we use this oblivious-access sort algorithm as a black box in order to construct the final offline ORAM compiler. The main idea within this step is to treat the program Π and the data x *together*, and to make use of the Oblivious-Access Sort procedure to enable routing of values to particular desired sub-orderings.

Consider, for example, a slightly simplified case where the programs simply indicate a length-n sequence of Read operations at fixed addresses in $[n]$. This can be obliviously simulated via the following sub-steps. First, the entire size-$2n$ memory contents—including both the query sequence and the data—are labeled and sorted so as to move memory into blocks each associated with a single index of data: starting with the data value itself x_i, and followed by the chronological sequence of Read requests to this address i. This is depicted in the first figure below (where "Ri" denotes Read request at address i). Once we have this structure, each Read request can be satisfied by a single pass through the database, "filling in" the correct value by (always) looking at the preceding word. This is depicted in the second figure below.

[4] This transpose step is the source of the $\log w$ additive complexity overhead term.



Then, the memory contents are returned to their original ordering by a reverse label and sort, yielding the n data items followed by the n requested items in desired order.

We remark that in this work we consider a somewhat restricted offline setting, where access Read/Write instructions are pre-specified, as opposed to simply access addresses. However, this case is already strong enough to be covered by the [GO96] lower bound, assuming balls and bins.

1.2 Open Questions

The observations and results above draw forth several interesting open questions in oblivious RAM research. We view these research directions as a further contribution of the present work.

Online lower bound? A good starting point toward proving general ORAM lower bounds (without balls and bins restrictions) is within an even stronger *online* model, where the simulation must successfully answer each data access request before learning the next. This more stringent variant is, in fact, the notion satisfied by essentially all known positive results in ORAM. We propose a definition of online ORAM in Section 2.2 (Definition 2.10).

Relaxing Balls and Bins? One can orthogonally relax the balls and bins model restriction. For example, does the [GO96] lower bound extend to a setting of *Balls and Bins with Linear Encoding*, where the CPU may generate balls of a different "color" (say, black) as linear combinations of original white balls (treated as formal monomials), may only output white balls at the conclusion of simulation, and can convert a black ball back to white only if its formal polynomial reduces to a single monomial. Note that this model captures approaches in the style of network coding, but does not allow for the type of bit-slicing manipulation we employ in our offline ORAM solution.

Strong offline ORAM from circuits? Is it possible to extend our circuit-based ORAM construction to a more expressive class of programs? For example, suppose the programs explicitly specify the addresses of memory accesses, but only know *at runtime* (e.g., as a result of partial computation) what actual read/write instruction will be performed at this location. We refer to this as a "strong offline" case.

Converse Relation? We show that small sorting circuits imply efficient offline ORAMs; does the converse also hold? Namely, given an offline (or even online) n-word ORAM with $O(1)$ registers and bandwidth and/or computation overhead $o(\log n)$, does this imply the existence of sorting circuits of size $o(nw \log n)$? The challenge here is that, although the access patterns of the ORAM are (essentially) input-independent, they may rely on randomness generated at runtime in order to attain low overhead.

Computationally Secure ORAM? As mentioned, the [GO96] lower bound crucially relies on the *statistical* closeness of access pattern distributions for any two inputs. As soon as this is relaxed to *computational* indistinguishability, where distributions of access patterns generated

from two inputs cannot be distinguished by efficient algorithms (but may even be disjoint), we could even hope to attain constant overhead in bandwidth and computation.[5]

1.3 Related Work

Pippenger and Fischer [PF79] were the first to consider the issue of oblivious access patterns, in their case, for Turing Machines (TM). Our work can be seen as a converse of the Pippenger-Fischer approach who showed how to efficiently translate a TM into a circuit via *Oblivious* TMs, whereas we obtain Oblivious RAM via circuits. Note that they obtain an $\Omega(\log n)$ lower bound for online oblivious simulation of Turing Machines, but it is not clear that their bound is relevant to the question of online oblivious simulation of RAM.

Beame and Machmouchi [BM11] showed a super-logarithmic lower bound for *oblivious branching programs*. However, as they noted, this bound is not applicable to the standard model of Oblivious RAM: The standard ORAM model requires the probability distribution of the observed access patterns to be statistically close regardless of the input; in contrast, Beame's model requires that this holds *for every random string* chosen. Hence, to date, Goldreich and Ostrovsky's original lower bound is the only applicable bound known.

Damgård, Medlegard, and Nielsen [DMN11] proved a lower bound on the amount of secret randomness required for a probabilistic RAM to obliviously simulate an arbitrary RAM in an online setting. Namely, if fewer than $n/8$ items are accessed per original access, then the simulation must use at least $\theta(1)$ *secret* random bits on average per simulated read operation. Their lower bound applies to *online* ORAMs. Our work (on *offline* ORAMs) demonstrates that this is essential, since instantiating our construction with known sorting circuit constructions yields an unconditional offline ORAM that does not make use of any secret randomness.

It was observed by Apon *et al.* [AKST14] that the Goldriech-Ostrovsky [GO96] bandwidth lower bound does not hold if one allows *server-side computation* on data before sending. Recent constructions (e.g., [AKST14, DvDF$^+$15]) provide ORAM-like access pattern security with *constant* bandwidth per data query, by leveraging polylogarithmic server computation (and computational assumptions). In fact, we argue that the reason these works overcome the lower bound is that they 'violated' the balls and bins restriction (and provide computational security).

An alternative weaker "Oblivious *Network* RAM" model was proposed by Dachman-Soled *et al.* [DSLP$^+$15] which does not fall within the [GO96] lower bound, where the adversary sees only the most significant bits of accessed addresses (and the lower-order bits are hidden from adversarial view).

We emphasize that our offline ORAM construction is in the oblivious RAM computation model of [GO96], where server-side computation is not allowed, and the adversary sees the full addresses of accessed memory.

2 RAM and Oblivious RAM

We follow the terminology in [GO96].

[5] Constructions attaining $O(1)$ bandwidth overhead in the computational setting have recently been demonstrated, however fall outside of the standard ORAM model since they assume the RAM memory can perform local computations on data before sending to the CPU: See Related Work.

2.1 Random Access Machines

A Random Access Machine (RAM) is modeled as a pair of interactive Turing machines (ITM), corresponding to CPU and Memory, which "interact" with each other via a set of specified actions. In what follows, n, w, and $r \in \mathbb{N}$ will be used to denote the memory size, word size, and number of registers (i.e., CPU memory size) of the system.

Definition 2.1 (Memory)**.** For every $n, w, r \in \mathbb{N}$, let $\mathsf{MEM}_{n,w,r}$ be an ITM with communication tape and input/output/work tapes of size $O(rw)$ and $O(nw)$, respectively. It partitions its work tape into n words, each of size $O(w)$. After copying its input to its work tape, machine $\mathsf{MEM}_{n,w,r}$ is message-driven. Upon receiving a message $(\mathsf{inst}, \mathsf{addr}, \mathsf{val})$, where $\mathsf{inst} \in \{\mathsf{store}, \mathsf{fetch}, \mathsf{halt}\}$ (an *instruction*), $\mathsf{addr} \in [n]$ (an *address*), and $\mathsf{val} \in \{0,1\}^{O(w)}$ (a *value*), machine $\mathsf{MEM}_{n,w,r}$ acts as follows:

- If $\mathsf{inst} = \mathsf{store}$, then machine $\mathsf{MEM}_{n,w,r}$ copies the value val from the current message into word number addr of its work tape.
- If $\mathsf{inst} = \mathsf{fetch}$, then machine $\mathsf{MEM}_{n,w,r}$ sends a message consisting of the current contents of word number addr (of its work tape).
- If $\mathsf{inst} = \mathsf{halt}$, then machine $\mathsf{MEM}_{n,w,r}$ copies a prefix of its work tape (until a special symbol) to its output tape, and halts.

Definition 2.2 (CPU)**.** For every $n, w, r \in \mathbb{N}$, let $\mathsf{CPU}_{n,w,r}$ be an ITM with communication and work tapes of size $O(rw)$, operating as hereby specified. After copying its input to its work tape, machine $\mathsf{CPU}_{n,w,r}$ conducts a computation on its work tape, and sends a message determined by this computation. In subsequent rounds, $\mathsf{CPU}_{n,w,r}$ is message driven. Upon receiving a new message, machine $\mathsf{CPU}_{n,w,r}$ copies the message to its work tape, and based on its computation on the work tape, sends a message. In case the $\mathsf{CPU}_{n,w,r}$ sends a halt message, the $\mathsf{CPU}_{n,w,r}$ halts immediately (with no output).

Definition 2.3 (RAM)**.** For every $n, w, r \in \mathbb{N}$, let $\mathsf{RAM}_{n,w,r}$ be a pair of $(\mathsf{CPU}_{n,w,r}, \mathsf{MEM}_{n,w,r})$, where $\mathsf{CPU}_{n,w,r}$'s read-only message tape coincides with $\mathsf{MEM}_{n,w,r}$'s write-only message tape, and vice versa. The *input* to $\mathsf{RAM}_{n,w,r}$ is a pair (s, y), where s is an initialization input for $\mathsf{CPU}_{n,w,r}$,[6] and y is the input to $\mathsf{MEM}_{n,w,r}$. The *output* of $\mathsf{RAM}_{n,w,r}$ on input (s, y), denoted $\mathsf{RAM}_{n,w,r}(s, y)$, is defined as the output of $\mathsf{MEM}_{n,w,r}(y)$ when interacting with $\mathsf{CPU}_{n,w,r}(s)$.

A *probabilistic*-$\mathsf{RAM}_{n,w,r}$ is a $\mathsf{RAM}_{n,w,r}$ in which $\mathsf{CPU}_{n,w,r}$ additionally has the ability to generate randomness on the fly as part of its local computation.

To view $\mathsf{RAM}_{n,w,r}$ as a universal machine, we separate the input y to $\mathsf{MEM}_{n,w,r}$ as $y = (\Pi, x)$ containing both the "program" and "data."

Remark 2.4. For purposes of lower bounds, it is generally considered that the running time of $\mathsf{RAM}_{n,w,r}$ is always greater than the length of the input (i.e., $|y|$). Under this assumption, we may ignore the "loading time" and count only the number of machine cycles in the execution of Π on x (ie., the number of rounds of message exchange between $\mathsf{CPU}_{n,w,r}$ and $\mathsf{MEM}_{n,w,r}$).

[6] Without loss of generality, s may be a fixed "start symbol."

2.2 Oblivious RAM

To define oblivious simulation of RAMs, we first define oblivious RAM machines. Loosely speaking, the "memory access pattern" in an oblivious RAM, on each input, depends only on its running time (on this input). Note that we do not regard hiding the *contents* of memory, which can be achieved independently (e.g., using encryption).

Definition 2.5 (Access Pattern)**.** The *access pattern* of a (probabilistic-)$\mathsf{RAM}_{n,w,r}$ on input y, denoted by $\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, y)$, is a distribution (over the random coins of $\mathsf{CPU}_{n,w,r}$) of sequences $(\mathsf{addr}_1, \ldots, \mathsf{addr}_i, \ldots)$, such that for every i, the ith message sent by $\mathsf{CPU}_{n,w,r}$, when interacting with $\mathsf{MEM}_{n,w,r}(y)$ with corresponding random coins, is of the form $(\cdot, \mathsf{addr}_i, \cdot)$.

We define an oblivious RAM to be a probabilistic RAM for which the probability distribution of memory access patterns during an execution depends only on the running time (i.e., is independent of the particular input $y = (\Pi, x)$).

Definition 2.6 (Oblivious RAM)**.** For every $n, w, r \in \mathbb{N}$, we define an *oblivious* $\mathsf{RAM}_{n,w,r}$ as a probabilistic $\mathsf{RAM}_{n,w,r}$ satisfying the following condition. For every two strings y_1, y_2, if $|\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, y_1)|$ and $|\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, y_2)|$ are identically distributed, then $\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, y_1)$ and $\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, y_2)$ are identically distributed.

We consider three primary notions (of varying strength) for oblivious simulation of an arbitrary RAM program on an oblivious RAM. The first, *"standard"* notion (as in [GO96]) holds for all RAM programs, and requires only that both machines compute the same function. In contrast, the [GO96] lower bound holds also within a weaker *"offline"* setting—considering only those programs whose memory access addresses are specified explicitly within their descriptions. We additionally propose a definition of a stronger *"online"* notion of oblivious simulation (that is obtained by most ORAM constructions)—where the compiler must satisfy each access query on the fly.

Definition 2.7 ((Standard) Oblivious RAM Simulation)**.** We say that a probabilistic-$\mathsf{RAM}'_{n',w',r'}$ *obliviously simulates* $\mathsf{RAM}_{n,w,r}$ if the following conditions hold:

- **Correctness.** There exists a negligible function ν for which the probabilistic-$\mathsf{RAM}'_{n',w',r'}$ simulates $\mathsf{RAM}_{n,w,r}$ with probability $1 - \nu(n)$. That is, for every input y, with probability $1 - \nu(n)$ over the choice of random coins of $\mathsf{CPU}'_{n',w',r'}$, the output of $\mathsf{RAM}'_{n',w',r'}$ on input y equals the output $\mathsf{RAM}_{n,w,r}(y)$ of $\mathsf{RAM}_{n,w,r}$ on the input y.
- **Obliviousness.** The probabilistic-$\mathsf{RAM}'_{n',w',r'}$ is oblivious (as per Definition 2.6).
- **Non-triviality.** The random variable representing the running-time of probabilistic-$\mathsf{RAM}'_{n',w',r'}$ (on input y) is fully specified by the running-time of $\mathsf{RAM}_{n,w,r}$ (on input y).

Definition 2.8 (*Offline* Oblivious RAM Simulation)**.** We say that a probabilistic-$\mathsf{RAM}'_{n',w',r'}$ is an *offline oblivious simulation of* $\mathsf{RAM}_{n,w,r}$ if the Correctness, Obliviousness, and Non-triviality properties of Definition 2.7 hold for the restricted class of $y = (\Pi, x)$ corresponding to *Fixed-Access programs*: A Fixed-Access program Π contains within its description the explicit sequence of communication triples of the form $(\mathsf{fetch}, \mathsf{addr}, \perp)$ or $(\mathsf{store}, \mathsf{addr}, \mathsf{val})$ for pre-specified val.

Note that, for the above definition to be non-trivial, each fetch operation will implicitly be followed by an output of the fetched value.

Intuitively, an *online* ORAM simulation requires that each access instruction (and output) is successfully completed before learning the next. This prevents the simulation from "pre-processing"

any of its access patterns in order to aid future lookups. We formalize this by splitting the program into sequential sub-programs $\Pi = \Pi_1, \Pi_2, \ldots, \Pi_t$ (determined at runtime based on execution), each with a single memory access (and, we will assume, a single output), and introducing an oracle $\mathcal{O}^{\mathsf{NextStep}}$. Instead of the entire input program Π being loaded into memory at initialization, Π is instead given only oracle access to $\mathcal{O}^{\mathsf{NextStep}}$. At any time during execution, the CPU may send a message next to $\mathcal{O}^{\mathsf{NextStep}}$, who responds by loading the next piece Π_{i+1} of the program into memory (i.e., the work tape of $\mathsf{MEM}_{n,w,r}$). However, the CPU must specify its output to the previous Π_i before requesting Π_{i+1}: namely, the ith output cannot be modified after the ith next request is made.

Definition 2.9 (Online RAM Model)**.** For every $n, w, r \in \mathbb{N}$, let a (probabilistic) *online RAM* $\mathsf{onlineRAM}_{n,w,r}$ be a triple of ITMs $(\mathsf{CPU}_{n,w,r}, \mathsf{MEM}_{n,w,r}, \mathcal{O}^{\mathsf{NextStep}})$. The input to $\mathsf{onlineRAM}_{n,w,r}$ is a triple (s, x, Π), where s is an initialization start input for $\mathsf{CPU}_{n,w,r}$, x is the (data) input to $\mathsf{MEM}_{n,w,r}$, and Π is the (program) input to $\mathcal{O}^{\mathsf{NextStep}}$. $\mathsf{CPU}_{n,w,r}$ begins by performing a computation on its work tape, and then may send a message to either $\mathsf{MEM}_{n,w,r}$ or $\mathcal{O}^{\mathsf{NextStep}}$ based on this computation. In subsequent rounds, $\mathsf{CPU}_{n,w,r}$ is message driven, as before.

Messages sent from $\mathsf{CPU}_{n,w,r}$ to $\mathsf{MEM}_{n,w,r}$ take the same form as in the standard RAM model, and are responded to identically. Messages sent from $\mathsf{CPU}_{n,w,r}$ to $\mathcal{O}^{\mathsf{NextStep}}$ take the form $(\mathsf{inst}, \mathsf{state}, \mathsf{out})$, corresponding to an instruction $\mathsf{inst} \in \{\mathsf{next}, \mathsf{halt}\}$, the current contents $\mathsf{state} \in \{0,1\}^{rw}$ of the work tape of $\mathsf{CPU}_{n,w,r}$, and an intermediate output value $\mathsf{out} \in \{0,1\}^w$. $\mathcal{O}^{\mathsf{NextStep}}$ maintains a local counter i (initialized at the beginning of execution to $i = 0$), and upon receiving such message from $\mathsf{CPU}_{n,w,r}$, does the following.

- If $\mathsf{inst} = \mathsf{next}$, then $\mathcal{O}^{\mathsf{NextStep}}$ determines the $(i+1)$th instruction Π_{i+1} of Π given the received value state, and sends a description Π_{i+1} to $\mathsf{MEM}_{n,w,r}$, who copies it into a designated location in its work tape. In addition, $\mathcal{O}^{\mathsf{NextStep}}$ writes out to the ith position of its output tape.
- If $\mathsf{inst} = \mathsf{halt}$, then $\mathcal{O}^{\mathsf{NextStep}}$ outputs the full contents of its output tape.

The *output* of $\mathsf{onlineRAM}_{n,w,r}$ on input (s, d, Π), denoted $\mathsf{onlineRAM}_{n,w,r}(s, x, \Pi)$, is defined as the output of $\mathcal{O}^{\mathsf{NextStep}}(\Pi)$ when interacting with $\mathsf{CPU}_{n,w,r}(s)$ and $\mathsf{MEM}_{n,w,r}(x)$.

An *online oblivious RAM* is on online RAM satisfying the obliviousness requirement of Definition 2.6, where $\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, y)$ is as in Definition 2.5 (i.e., the sequence of accessed memory addresses, without information on calls to $\mathcal{O}^{\mathsf{NextStep}}$).

Note that the CPU commits itself to the ith step output *before* it gains access to the next instruction Π_{i+1} (through MEM).

Definition 2.10 (*Online* Oblivious RAM Simulation)**.** We say that a probabilistic-$\mathsf{RAM}'_{n',w',r'}$ is an *online oblivious simulation of* $\mathsf{RAM}_{n,w,r}$ if the Correctness, Obliviousness and Non-triviality properties of Definition 2.7 hold in the *Online RAM Model*, as per Definition 2.9.

In any case (standard, offline, etc.), simulation of a RAM by an oblivious RAM incurs certain overhead costs. In this work, we focus on the overhead in *computation* and *bandwidth*.

Definition 2.11 (Overhead of Oblivious Simulations)**.** Suppose that a probabilistic-$\mathsf{RAM}'_{n',w',r'}$ obliviously simulates the computations of $\mathsf{RAM}_{n,w,r}$.

- We say that the *bandwidth overhead of the simulation is at most* g for some function $g : \mathbb{N} \to \mathbb{N}$ if, for every y, at most $g(B) \cdot B$ bits are written by $\mathsf{MEM}'_{n',w',r'}$ to its communication tape throughout the course of the simulation, where B denotes the number of bits written by $\mathsf{MEM}_{n,w,r}$ to its communication tape in the original execution.

- We say that the *computation overhead of the simulation is at most* g for some function $g : \mathbb{N} \to \mathbb{N}$ if, for every y, at most $g(t) \cdot t$ computation steps are taken during the execution of $\mathsf{RAM}'_{n',w',r'}(y)$, where t denotes the number of computation steps taken in the original execution $\mathsf{RAM}_{n,w,r}(y)$.

3 Offline ORAM Lower Bounds Imply Circuit Lower Bounds

Our main theorem demonstrates that in the *offline* case, lifting the "balls and bins" restriction from the Goldreich-Ostrovsky lower bound will require proving yet unknown circuit lower bounds on the size of Boolean sorting circuits. This conclusion is obtained constructively: Given a small sorting circuit, we show how to build a secure offline ORAM with sub-logarithmic overhead.

Theorem 3.1. *Suppose there exists a Boolean circuit ensemble* $C = \{C(n, w)\}_{n,w}$ *of size* $s(n, w)$*, such that each* $C(n, w)$ *takes as input* n *words each of size* w *bits, and outputs the words in sorted order. Then for word size* $w \in \Omega(\log n) \cap n^{o(1)}$ *and constant CPU registers* $r \in O(1)$*, there exists a secure offline ORAM (as per Definition 2.8) with total bandwidth and computation* $O(n \log w + s(2n/w, w))$*.*

In particular, given the existence of any sorting circuit ensemble with size $o(nw \log n)$, we obtain an offline ORAM construction that bypasses the [GO96] lower bound:

Corollary 3.2. *If there exist Boolean sorting circuits* $C = \{C(n, w)\}_{n,w}$ *of size* $s(n, w) \in o(nw \log n)$ *(for* $w \in \Omega(\log n) \cap n^{o(1)}$*), then there exists secure offline ORAM with* $O(1)$ *CPU registers and bandwidth and computation* overhead $o(\log n)$*.*

The total storage requirement of our offline ORAM construction is $O(n + s(2n/w, w))$. For circuits of size $s(m, w) \in o(mw \log m)$, this corresponds to $o(\log n)$ storage overhead. We do not assume much on the computational power of the (compiled) CPU, except that it be able to perform bitwise logical operations and be able to extract parts of a word.

The proof of Theorem 3.1 proceeds via two steps. First, in Section 3.2, we begin by constructing and analyzing an efficient *Oblivious-Access Sort* algorithm: i.e., a (randomized) RAM program for sorting an n-word database, where the distribution of access patterns is statistically close over any two choices of the input database. We remark that this notion of oblivious-access aligns directly with that of ORAM. (In contrast, the term "Oblivious Sort" refers in the literature to sorting algorithms whose access patterns are *fixed*).

Definition 3.3 (Oblivious-Access Sort)**.** An *Oblivious-Access Sort* algorithm for input size n (and word size w) and computation $\mathsf{comp}(n, w)$ is a (possibly randomized) RAM program Π in which the following properties hold:

- Efficiency: The program Π terminates in $\mathsf{comp}(n, w)$ computation steps.
- Correctness: With overwhelming probability in n, at the conclusion of Π, the database contains the n inputs, in sorted order.
- Oblivious Access: There exists a negligible function ν such that for any two inputs x, x' of size n, then $\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, (\Pi, x))$ is $\nu(n)$-close statistically to $\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, (\Pi, x'))$ (see Definition 2.5).

Then, in Section 3.3, this oblivious-access sort algorithm will be used as a black box in order to construct the final offline ORAM compiler.

We first begin in Section 3.1 by introducing some useful notation.

3.1 Notation

Throughout this work, n, w, and $r \in \mathbb{N}$ will be used to denote the (external) memory size, word size, and number of registers (i.e., CPU memory size) of the system. We consider the range in which $w \in \Omega(\log n) \cap n^{o(1)}$ and $r \in O(1)$.

We denote the work tapes of $\mathsf{CPU}_{n,w,r}$ and $\mathsf{MEM}_{n,w,r}$ as arrays $\mathsf{Reg}, \mathsf{Mem}$.

In general, capital letters are used to denote arrays (e.g., $\mathsf{Reg}, \mathsf{Mem}$), lowercase letters to denote words (sometimes interpreted as bit strings). Indexing: for an array (e.g., memory Mem), $\mathsf{Mem}[i]$ denotes word i in the array; $\mathsf{Mem}[i][j]$ denotes the jth bit of $\mathsf{Mem}[i]$ in the array. For a single word, $x[j]$ denotes the jth bit of x. When describing addresses of the head of an array (in particular, to be passed as an argument to a function call), we will denote by $D' :=$ "$D[\times n]$" the array D' for which each $D'[i+1] = D[n \cdot i + 1]$

For $n \in \mathbb{N}$, we denote by S_n the symmetric group on n items.

We consider general binary circuits of fan-in 2. Denote by $G = \{f : \{0,1\}^2 \to \{0,1\}\}$ the set of all possible boolean gate functions on two inputs, indexed by an integer in $\{1, \ldots, 16\}$.

Notation 3.4 (Boolean Circuit Model)**.** A (fan-in 2) boolean circuit C with n-bit inputs, m-bit outputs, and size $|C| = s$ is a collection of s gates g_i of the following form:

- Input gates: $\{g_i\}_{1 \le i \le n}$ each directly associated with bit i of the input bit string.
- Computation gates: $\{g_i\}_{n < i \le s-m}$ each specified by a triple (f, i_L, i_R), where $f \in G$ (see above), and $1 \le i_L, i_R < i$.
- Output gates: $\{g_i\}_{s-m < i \le s}$ each specified by a single index $i_{\mathsf{out}} < i$.

3.2 From Sorting Circuits to Oblivious-Access Sort

For simplicity of exposition, we treat the case of oblivious-access sorting of *distinct* data items. This will suffice for our Offline ORAM application. However, a few modifications to the algorithm will enable sorting of non-distinct items.

Proposition 3.5 (Oblivious-Access Sort)**.** *If there exist Boolean sorting circuits $C = \{C(n,w)\}_{n,w}$ of size $s(n,w)$, then there exists an Oblivious-Access Sort algorithm for n* distinct *elements using $O(1)$ CPU resisters, with total bandwidth and computation complexity each $O(n \log w + s(2n/w, w))$, and probability of error $e^{-n^{\Omega(1)}}$.*

In order to avoid the factor of w overhead in directly emulating the circuit by RAM, in our sorting algorithm the CPU will manipulate the bit structure of words stored in memory, so that we can "make progress" on *all* bits within words pulled into memory. The governing observation is that the above circuit-emulation approach is not too costly when executing independently *on w groups of only n/w words in parallel*, taking a bit-slicing/SIMD approach.

At a high level, our Oblivious-Access Sort RAM program OASort takes the following form. Phases 1 and 2 are depicted visually in Figure 1; Phases 3 and 4 are depicted in Figure 2.

1. First, we perform a random public shuffle of database items. Since the permutation may be public (without attempting to hide the effective permutation), this step may be executed straightforwardly, e.g. via Knuth (Fisher-Yates) shuffle [Knu97]. The purpose of this initial shuffle is to reduce the problem of worst-case sorting to that of sorting a list in random order. Note, however, that no *secret* randomness is required.

2. Second, we sort w separate groups of only n/w words each, *in parallel*. This requires the algorithm to first "repack" words to contain consistent bits from each of the w parallel executions (corresponding to transpose of bits in memory), then to emulate the circuit on "packed" words SIMD-style, and finally to transpose the words back to original form.

   At the conclusion of this step, we have w interleaved sets of n/w sorted words. Our remaining task is to merge these sorted lists into a single sorted list of n words.

3. Because of the random shuffle in Step 1, we are guaranteed with overwhelming probability that no element's current position is too far from its position in the final sorted list. We may thus split the list into new blocks of size $n/w \in \omega(\sqrt{n})$ and sort each block independently, in parallel. To handle elements near the boundary of these blocks, we extend the "window" of each block by an extra $n/(2w)$ on either side, introducing overlaps between blocks.

   At the conclusion of this step, we have a database of size $2n$ (because of overlaps), where each ith consecutive block of $2n/w$ words is individually sorted, and is guaranteed to contain as a subsequence the ith set of n/w words in the *final* sorted list. The goal is now to remove duplicated words, leaving behind this complete sorted subsequence.

4. Removing duplicates takes place via three sub-steps. First, the duplicate items are *identified* (and replaced by $-\infty$ symbols) by a one-pass over the data. In order to remove them *obliviously* (since their location is input-dependent), we re-sort each of the $2n/w$-sized blocks, allowing the $-\infty$ garbage items to sift to the first n/w positions of each $2n/w$-block. Then, we may deterministically remove garbage items and compress back to a sorted list of size n.

In Appendix B.1, we provide and analyze useful sub-algorithms RandPerm, Transpose, EmulateCircuit, and RemRedundant. RandPerm(D, n) shuffles the n words in D to random order. Transpose(D, n, w) implements a bitwise transpose within each block of w words within the size-n database D. EmulateCircuit(D, C, s, n, m) performs a *SIMD* execution of the circuit C on the data held in D (i.e., component-wise for each bit in the RAM words). RemRedundant(E, n) steps through a database sorted in blocks with overlap, and identifies and zeroes out redundant items.

Oblivious-Access Sort. A description of our procedure OASort is given in Figure 3.

Proof of Proposition 3.5. The complexity (both computation and bandwidth) of each step in OASort is indicated in grey below each arrow within Figures 1 and 2, yielding a total complexity of $O(n \log w + s(2n/w, w))$. These individual values are derived from the complexities of the underlying sub-algorithms RandPerm, Transpose, etc. constructed and analyzed in Appendix B.1.

The correctness and obliviousness of the OASort procedure (Figure 3) hold via the following sequence of intermediate claims.

The first claim shows that, in a slightly tweaked version of the OASort Steps 1-3, the items of the n/w individually sorted lists will *not* appear too far from their final positions in the complete n-sorted list, with overwhelming probability over the randomness of the initial shuffle. In the experiment below, the values stored in D represent the *indices* of the distinct values to be sorted, with respect to their correct sorted order (i.e., smallest is 1, largest is n). (The difference between this experiment and the OASort Steps 1-3 is that here we begin with elements already in sorted order $D[i] = i$, and then apply a random permutation, whereas in OASort the values begin in arbitrary unsorted order). The experiment fails in abort if after randomly permuting and then sorting the interleaved groups (each of size $\ell := n/w$), any position of D holds a value that is "far" from its target value (specifically, if it falls outside the 2ℓ-size region assigned to the ℓ-size region in which it belongs).

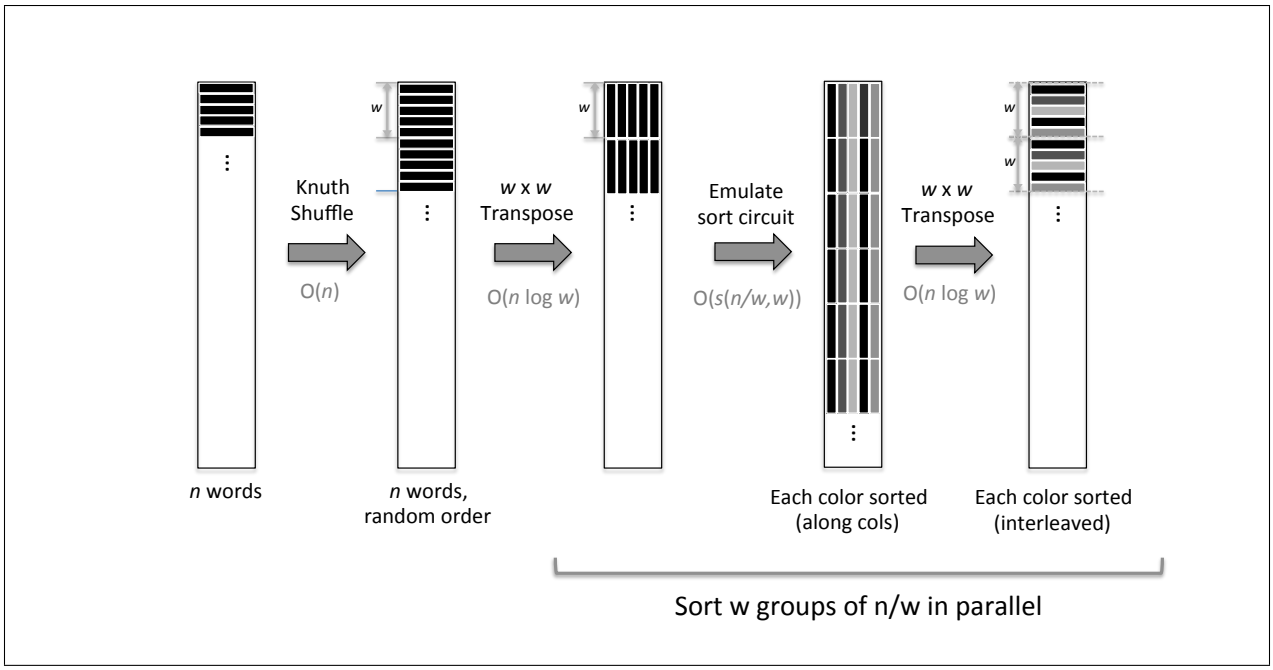


Figure 1: First phases of OASort.

Claim 3.6. *Fix $\ell := n/w \in \omega(\sqrt{n})$. (Recall $w \in n^{o(1)}$). With overwhelming probability in n (over $\sigma \leftarrow S_n$), the following experiment does* not *end in* abort*:*

1. *Sample random $\sigma \leftarrow S_n$; $\forall i \in [n]$, set $D[\sigma(i)] \leftarrow i$.*
2. *Individually sort each of the w interleaved groups of (n/w) words of D: i.e., $\forall i \in [w]$, $D[i] \le D[w+i] \le \cdots \le D[(n/w-1)w+i]$. (Indexing as in Figure 1).*
3. *If for any $(j,k) \in [w] \times [n/w]$ (now indexing as in Figure 2) the following holds, the experiment ends in* abort *(recall that in perfect sorted order, $D[j\ell+k] = j\ell+k$):*

$$j\ell + k \notin \{D[(j-1/2)\ell], \ldots, D[(j+3/2)\ell]\}.$$

That is, the value that should be sorted to the kth word in jth n/w-block is currently located somewhere outside *the specified $2n/w$-window spanning this location (see Figure 2).*

Proof. Refer to the w interleaved groups of (n/w) words defined in Step 2 by "color" $C_1, \ldots, C_w$. Namely, $C_i := \{i, w+i, \ldots, (n/w-1)w+i\}$. This aligns with the colors of the right-most diagram in Figure 1.

Fix one value of $(j,k) \in [w] \times [n/w]$. The event that $j\ell + k = D[\mathsf{pos}]$ for some $\mathsf{pos} < (j-1/2)\ell$ corresponds to a case where significantly fewer than the expected number of values smaller than $j\ell+k$ ended up in the same color (and thus $j\ell+k$ appears unexpectedly early). Formally, this probability of this event is bounded by the probability that, in $\sigma \leftarrow S_n$, *strictly fewer* than $\frac{1}{w}(j-1/2)\ell$ of the first $j\ell+k$ values $\sigma(1), \ldots, \sigma(j\ell+k)$ will have the same color as $j\ell+k$ (in particular, $C^* := C_{j\ell+k \mod w}$). Note that in expectation, $\frac{1}{w}(j\ell+k)$ of these values should receive the same color. Consider the equivalent experiment of sampling $j\ell+k$ values X_i without replacement from a list $\mathcal{X}$ of n values X_i, where n/w of the X_i are equal to 1 (corresponding to those with color C^*)

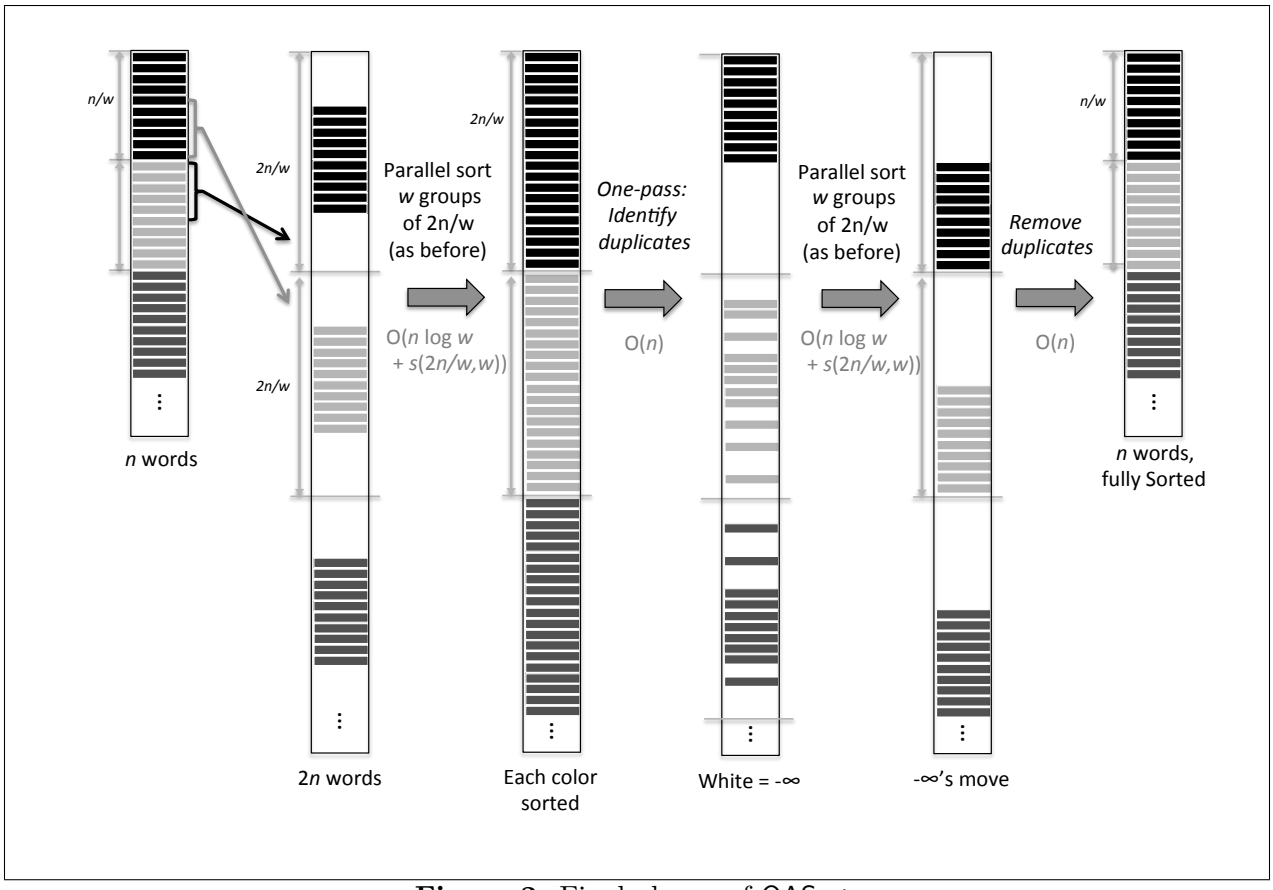


Figure 2: Final phases of OASort.

and the remaining $n - n/w$ are equal to 0. By Hoeffding's inequality,[7]

$$\Pr\left[\sum_{i=1}^{j\ell+kj} X_i < \frac{(j-1/2)\ell}{w}\right] \le \Pr\left[\sum_{i=1}^{j\ell} X_i < \frac{(j-1/2)\ell}{w}\right] \le \Pr\left[\left|\sum_{i=1}^{j\ell} X_i - \frac{j\ell}{w}\right| > \frac{\ell}{2w}\right] \le 2e^{-2(\ell/2w)^2},$$

which (for $w \in n^{o(1)}$) is $\le 2e^{-n^{\Omega(1)}}$.

Similarly, the event that $j\ell+k = D[\mathsf{pos}]$ for $\mathsf{pos} > (j+3/2)\ell$ is bounded by the probability that, in $\sigma \leftarrow S_n$, *strictly more* than $\frac{1}{w}(j+3/2)\ell$ of the first $j\ell+k$ values $\sigma(1), \ldots, \sigma(j\ell+k)$ will have color C^* (as above). Consider an experiment of sampling $j\ell+k$ values without replacement from the set $\mathcal{X}$, as above. Then sampling $(j+3/2)\ell/w$ or more "1" values X_i departs significantly from the expected number of such appearances $(j\ell+k)/w \le (j\ell+\ell)/w$, and we bound the probability of this via Hoeffding's inequality:

$$\Pr\left[\sum_{i=1}^{j\ell+k} X_i > \frac{(j+3/2)\ell}{w}\right] \le \Pr\left[\sum_{i=1}^{j\ell+\ell} X_i > \frac{(j+3/2)\ell}{w}\right] \le \Pr\left[\left|\sum_{i=1}^{(j+1)\ell} X_i - \frac{(j+1)\ell}{w}\right| > \frac{\ell}{2w}\right] \le 2e^{-2(\ell/2w)^2},$$

which again is $\le 2e^{-n^{\Omega(1)}}$.

[7] Hoeffding's inequality, sampling without replacement, special case of 0/1 values: $\Pr\left[|S_m - E[S_m]| \ge t\right] \le 2e^{-2t^2}$, where S_m denotes the sum of m values sampled from $\mathcal{X}$ [Hoe63].

Oblivious-Access Sort $\mathsf{OASort}(D, n)$
$\mathsf{RandPerm}, \mathsf{Transpose}, \mathsf{EmulateCircuit}, \mathsf{RemRedundant}$ are given in Algorithms 1-4.

1. $\mathsf{RandPerm}(D, n)$. Perform a random public shuffle of the database items.
2. Sort in parallel w blocks of size n/w (see Figure 1):
   (a) $\mathsf{Transpose}(D, n, w)$. Transpose data to SIMD form.
   (b) $\mathsf{EmulateCircuit}(D[\times w], C_{\mathsf{sort}}, s(n), n, n)$. SIMD emulate boolean circuit C_{sort} on D.
   (c) $\mathsf{Transpose}(D, n, w)$. Transpose data back to (standard) word form.

   Result: w interleaved groups of n/w words, each individually sorted.
3. Merge blocks into 1 sorted list *with overlaps*, by sorting (overlapping) blocks in parallel.
   (a) Arrange data to appropriate form (see Figure 2). Namely, let $\ell = n/w$.
   Define $2n$-size database E: For each $i \in \{0, \ldots, w-1\}$ and $j \in \{-\ell/2+1, \ldots, 3\ell/2\}$, let $E[2i\ell + \ell/2 + j] \leftarrow D[i\ell + j]$, where $D[i] := -\infty$ (special min elmt) for $i < 1$ and $i > n$.
   (b) Sort 2ℓ-size blocks of E in parallel (see Figure 2):
      i. Reorder words: Define temp array $E'[iw+j+1] \leftarrow E[2\ell j+i+1]$ for $i \in \{0, \ldots, 2n/w-1\}$, $j \in \{0, \ldots, w-1\}$. (Recall $\ell = n/w$).
      ii. $\mathsf{Transpose}(E', 2n, w)$. Transpose E' bitwise to SIMD form.
      iii. $\mathsf{EmulateCircuit}(E', C_{\mathsf{sort}}, s(2n), 2n, 2n)$. SIMD emulate boolean circuit C_{sort} on w independent blocks of E'.
      iv. $\mathsf{Transpose}(E', 2n, w)$. Transpose data back to (standard) word form.
      v. Reorder words: Return data as $E[2\ell j+i+1] \leftarrow E'[iw+j+1]$ for $i \in \{0, \ldots, 2n/w-1\}$, $j \in \{0, \ldots, w-1\}$. (Recall $\ell = n/w$).

   Result: $2n$-size database E contains all n items of D in sorted order, but with overlaps.
4. Remove redundant items (see Figure 2).
   (a) $\mathsf{RemRedundant}(E, 2n)$. One-pass to identify redundant items, setting to $-\infty$:
   (b) Sort each block again, repeating Step 3b.
   (c) One pass: Compress E back to D, removing $-\infty$ values. Namely, for each $i \in \{0, \ldots, w-1\}$ and $j \in \{1, \ldots, \ell\}$, set $D[i\ell + j] \leftarrow E[2i\ell + j + \ell]$.

Figure 3: Oblivious-access sorting algorithm for *distinct data items*.

Finally, combining the probability of these two bad events, and taking a union bound over the n possible choices of (j, k), the claim holds. □

We now adjust the previous experiment to match that of OASort Steps 1-3. Namely, we do not assume the items in D begin in sorted order, but rather in arbitrary permuted order π.

Claim 3.7. *For any fixed $\pi \in S_n$, Claim 3.6 holds also if the assignment in Step 1 is replaced by "Set $D[\sigma(i)] \leftarrow \pi(i)$."*

Proof. Follows directly: $D[\sigma(i)] \leftarrow \pi(i)$ is equivalent to $D[\sigma(\pi^{-1}(i))] \leftarrow i$, and for fixed π, the uniform distribution $\{\sigma \leftarrow S_n\}$ is identical to $\{\sigma \circ \pi^{-1} : \sigma \leftarrow S_n\}$. □

In Appendix B.2 we use these claims in order to show correctness of OASort:

Claim 3.8 (OASort Correctness)**.** *For any sequence of n* distinct *initial values $D = (D[1], \ldots, D[n])$, then with probability $1 - e^{-n^{\Omega(1)}}$, $\mathsf{OASort}(D, n)$ outputs the values in sorted order $D[1] \leq \cdots \leq D[n]$.*

Claim 3.9 (Perfect Obliviousness)**.** *For any two sequences of n* distinct *values $D := (D[1], \ldots, D[n])$, $D' := (D'[1], \ldots, D'[n])$, it holds that the distribution of memory access patterns of* OASort *on input D and D' are identical: $\mathscr{A}ccess[\mathsf{OASort}(D, n)] \equiv \mathscr{A}ccess[\mathsf{OASort}(D', n)]$.*

Proof. Aside from the initial random shuffle, which induces a fixed input-independent distribution over accesses, all steps of OASort (i.e., Transpose, EmulateCircuit, RemRedundant), in fact have *fixed* access structure. (Note the accesses of EmulateCircuit are determined by the fixed sorting circuit topology). Perfect obliviousness thus follows immediately. □

This concludes the proof of OASort. □

3.3 Offline ORAM from Oblivious-Access Sorting

Now, suppose there exists an oblivious-access sort algorithm OASort, as per Definition 3.3. We now use such an algorithm as a black box in order to construct the desired offline ORAM simulation, with only constant multiplicative cost in bandwidth and computation over the corresponding values for OASort. This procedure is specified in Figure 4. For notational simplicity, we describe the case where the program proceeds in n time steps (this is extended simply by considering longer request sequence arrays S of length $t > n$).

Recall that to obtain an offline ORAM, we must be able to obliviously simulate for programs consisting of data access instructions Access(addr, val) where either $\mathsf{val} = \emptyset$ (for read) or a fixed and explicitly specified val (see Definition 2.8).

Our transformation begins by making a single pass through the input $y = (\Pi, x)$, and labeling each item with a triple (index, time, value). Words in the data portion x, denoted by $D[i]$ in Figure 4, will be labeled with: index corresponding to their address i, time = 0, and their listed value. Words in the program portion of the input Π, denoted by $S[j]$ in Figure 4 (for request *sequence*), and corresponding to a Read/Write request (addr, command), will be labeled with: index addr, time j (i.e., the jth request in time), and value command. In each execution of OASort on these triples, we will sort only with respect to *two* of the three words, and will carry the third word simply as a "payload."[8]

We next sort the *entire* program-data array (denoted M in Figure 4) with respect to key value (index, time), using an execution of OASort. As a result, the array M is now ordered in blocks of

[8] Note that in any case, sorting on words of length $3w$ instead of w incurs only constant complexity overhead.

words (of varying sizes), where each block corresponds to a separate index $\mathsf{index} \in [n]$. The first item of each block is the data payload itself, word $D[\mathsf{index}]$. Following this, in chronological order, will be the sequence of program requests accessing location index. In the restricted offline setting, we are guaranteed that access requests are limited to either $\mathsf{Access}(\mathsf{index}, \mathsf{val})$, where val is either $\emptyset$ (for read) or an explicitly specified value. We can thus satisfy each request by making a single pass through M, with a single look-back at each step: Each $\mathsf{Access}(\mathsf{index}, \mathsf{val})$ is assigned to val, and each $\mathsf{Access}(\mathsf{index}, \emptyset)$ is "filled in" with the value held in the location one previous.

As the final step, we re-sort the elements of M with respect to key $(\mathsf{time}, \mathsf{index})$, so as to return them to their original locations. The desired output sequence is now contained within the program portion of memory, S.

Proposition 3.10. *Suppose there exists an Oblivious-Access Sorting algorithm for sorting n words of size $w \in \Omega(\log n) \cap n^{o(1)}$ with computation/bandwidth complexity each $\mathsf{comp}(n, w)$. Then there exists an offline ORAM simulating programs of time $t \geq n$ with computation/bandwidth $O(\mathsf{comp}(n+t, 3w))$.*

In particular, if there exists OASort with cost $\mathsf{comp}(n, w) \in o(n \log n)$, as is guaranteed by Proposition 3.5 if there exist Boolean sorting circuits of size $o(nw \log n)$, then this yields an offline ORAM with cost $o((n+t)\log(n+t))$: i.e. (for $t \geq n$), with *sub-logarithmic overhead*.

Offline ORAM Compiler $\mathsf{OfflineORAM}(D, S, n)$
Inputs: starting address of database D, starting address of request sequence S, length n.

1. Preprocessing: Define a new array $M := D||S$, with appropriate labeling.
   (a) For each $i \in [n]$: Parse $D[i] = v$; set $M[i] \leftarrow (i, 0, D[i])$.
   (b) For each $j \in [n]$: Parse $S[j] = (\mathsf{index}, \mathsf{command})$; set $M[n+j] \leftarrow (\mathsf{index}, j, \mathsf{command})$.

   We now parse all entries $M[i]$ as $(\mathsf{index}, \mathsf{time}, v)$ where $\mathsf{index} \in [n]$, $\mathsf{time} \in [n] \cup \{0\}$, and $v \in \{0,1\}^w \cup \{\emptyset\}$.
2. Execute $\mathsf{OASort}(M, 2n)$, w.r.t. key value $(\mathsf{index}, \mathsf{time})$ (i.e., index is more significant).
3. Fill out data requests via one pass:

   Read $\mathsf{Reg}[1] \leftarrow M[1]$, parse as (index, i, v), and set $\mathsf{prevvalue} \leftarrow v$.
   For $i = 2$ to $2n$:
   (a) Read $\mathsf{Reg}[1] \leftarrow M[i]$, and parse as (index, i, v).
   (b) If $v = \emptyset$, then write $M[i] \leftarrow (\mathsf{index}, i, \mathsf{prevvalue})$.
   (c) Else, set $\mathsf{prevvalue} \leftarrow v$.
4. Execute $\mathsf{OASort}(M, 2n)$ w.r.t. key value $(\mathsf{time}, \mathsf{index})$ (i.e., time is more significant).
5. Output: starting address S to length-n array from $M[n+1]$ to $M[2n]$.

Figure 4: Offline ORAM, assuming oblivious-access sort procedure OASort.

Proof. The desired offline ORAM compiler $\mathsf{OfflineORAM}$ is given in Figure 4. We defer the proof of correctness to Appendix B.2.

Consider the complexity of the $\mathsf{OfflineORAM}$ steps. Steps 1 and 3 each incur a single pass of the database and simple manipulation, taking $O(n)$ computation. Steps 2 and 4 correspond to

executions of OASort on a database of size $n+t$ with word size $3w$ (corresponding to index-time-value triples), requiring time and bandwidth each $\mathsf{comp}(n+t, 3w)$. The claim follows.

Obliviousness of $\mathsf{OfflineORAM}$ follows directly from the obliviousness of OASort (Proposition 3.5), since all remaining steps (namely, preprocessing $M := D||S$ and filling out data requests via one pass over M) have fixed access structure, independent of the values of D and S.

This concludes the proof of Proposition 3.10. □

Simulating a PRAM: Getting now a PRAM simulation is simple. We assume that the PRAM has n memory cells and n processors (with $O(1)$ registers as internal memory per processor), and at every step each processor can access any cell, perform some computation involving its internal registers, and update any cell. That is, a PRAM program Π is a sequence of the following steps:

1. **Local computation (prepare read):** Each processor performs some local computation on its $O(1)$ registers. At the conclusion of computation, each processor identifies an address $\mathsf{addr} \in [n]$ within memory to read.
2. **Read memory:** Each CPU begins with an address $\mathsf{addr}_i \in [n]$ in memory to read. At the conclusion of this step, each CPU learns the value $\mathsf{Mem}[\mathsf{addr}_i]$.
3. **Local computation (prepare write):** Each processor performs some local computation on its $O(1)$ registers. As the result of computation, each processor identifies an address $\mathsf{addr} \in [n]$ and word $\mathsf{val} \in \{0,1\}^w$ to write to this location in memory.
4. **Write to memory:** Each CPU begins with an address $\mathsf{addr}_i \in [n]$ and an *explicit* value val to write in this location. At the conclusion of this step, each write instruction is implemented within memory. Conflicts in the values written are resolved by priority (say the value of the highest numbered processor; see [KR88, Vis15] for a survey).

The observation is that each of these steps itself has the form of a Fixed-Access program. Indeed, consider a (single-CPU) simulation of Π where each PRAM processor's local registers are written in designated portions of memory. Each local computation step can be simulated directly: Namely, for each of the PRAM processors, the simulation will read into memory all $O(1)$ of its local registers, perform the dictated local computation, and write the updated state back to memory, where the resulting Read/Write request information is written in a fixed location. For each Read or Write operation, the simulation will execute the above-described Offline ORAM procedure, where the length-n request sequence is specified in (fixed) locations corresponding to the n processors (instead of n sequential time steps). The priority writing is obtained automatically from the nature of the simulation, which sorts CPU requests chronologically, and assuming that higher numbered processors are later in the program than the lower ones. The computational power required from our simulating CPU is essentially equivalent to that of the original PRAM CPUs, requiring (in addition) only bitwise logical operations on words and extracting parts of a word.

We thus obtain an oblivious simulation of n-processor PRAM with computation/bandwidth overhead equal to that of our Offline ORAM simulation. In particular, if there exist Boolean sorting circuits of size $o(nw \log n)$, then (combining Propositions 3.5 and 3.10) for any t-step, n-processor PRAM, our simulation requires computation and bandwidth $o(tn \log n)$. Note that this is an asymptotic improvement over any known oblivious PRAM simulation in the standard model [BCP14, CLT15]. However, we simulate the PRAM by a *sequential* oblivious RAM, in contrast to the Oblivious PRAM setting considered in these works, where a PRAM is simulated on an oblivious PRAM. An interesting question is to what extent our construction may be parallelized to fit within this setting.

4 Acknowledgements

We thank Uri Zwick, Peter Bro Miltersen, Emanuele Viola and Ryan Williams for discussing on the state of the art of circuit bounds for sorting.

References

[ACS87] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science*, pages 204–216, 1987.

[AHNR95] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 427–436, 1995.

[Ajt10] Miklós Ajtai. Oblivious RAMs without cryptographic assumptions. In *STOC*, pages 181–190, 2010.

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

[AKST14] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. Cryptology ePrint Archive, Report 2014/153, 2014.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM. *IACR Cryptology ePrint Archive*, 2014:594, 2014.

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

[Bih97] Eli Biham. A fast new DES implementation in software. In *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, pages 260–272, 1997.

[BM11] Paul Beame and Widad Machmouchi. Making branching programs oblivious requires superlogarithmic overhead. In *Proceedings of the 26th Annual IEEE Conference on Computational Complexity, CCC 2011, San Jose, California, June 8-10, 2011*, pages 12–22, 2011.

[CKW13] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT*, pages 279–295, 2013.

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

[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. In *TCC*, pages 144–163, 2011.

[DSLP+15] Dana Dachman-Soled, Chang Liu, Charalampos Papamanthou, Elaine Shi, and Uzi Vishkin. Oblivious network ram. Cryptology ePrint Archive, Report 2015/073, 2015.

[DvDF+15] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious RAM. Cryptology ePrint Archive, Report 2015/005, 2015.

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

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99*, pages 285–298, 1999.

[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. In *Privacy Enhancing Technologies*, pages 1–18, 2013.

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

[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. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524, 2012.

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

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

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

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

[Goo14] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in O(n log n) time. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 684–693, 2014.

[Han04] Yijie Han. Deterministic sorting in O($n$loglog$n$) time and linear space. *J. Algorithms*, 50(1):96–105, 2004.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

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

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[KR88] Richard M. Karp and Vijaya Ramachandran. A survey of parallel algorithms for shared-memory machines. University of California, Berkeley Technical Report No. UCB/CSD-88-408 March 1988, 1988. http://www.eecs.berkeley.edu/Pubs/TechRpts/1988/CSD-88-408.pdf.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396, 2013.

[MS11] Muriel Medard and Alex Sprintson. *Network Coding: Fundamentals and Applications*. Elsevier, Amsterdam, 2011.

[Nao09] Moni Naor. On recycling encryption schemes or achieving resistance to cache attacks via low bandwidth encryption. Lecture Slides, MIT Workshop on Computing in the Cloud, 2009. people.sail.mit.edu/joanne/recycling_clouds.ppt.

[NR95] Moni Naor and Ron M. Roth. Optimal file sharing in distributed networks. *SIAM J. Comput.*, 24(1):158–183, 1995.

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

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[Ren14] Ling Ren. Unified RAW path oblivious RAM. Master's thesis, Massachusetts Institute of Technology, 2014.

[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*, 2014:997, 2014.

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

[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. In *ACM Conference on Computer and Communications Security*, pages 299–310, 2013.

[Vis15] Uzi Vishkin. Explicit multi-threading (xmt): A pram-on-chip vision. Online notes. http://www.umiacs.umd.edu/∼vishkin/XMT/index.shtml, Visited August 2015.

[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*, 2014:672, 2014.

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

A The [GO96] Lower Bound

The Goldreich-Ostrovsky bound assumes a "balls and bins" model of data manipulation, where the CPU may shuffle data items around, but cannot perform any further computation. Recall $n, w, r \in \mathbb{N}$ denote external memory size, word size, and local CPU memory size (i.e., number of local registers).

Definition A.1 (Balls and Bins Model)**.** A (probabilistic)-$\mathsf{RAM}_{n,w,r}$ operates in the *balls and bins model* if CPU registers begin empty and $\mathsf{CPU}_{n,w,r}$ operations are restricted to the following:

- *Move ball to Memory.* For some $i \in [r]$ and $\mathsf{addr} \in [n]$, write the tuple $(\mathsf{store}, \mathsf{addr}, \mathsf{Reg}[i])$ to outgoing communication tape, and erase $\mathsf{Reg}[i] \leftarrow \emptyset$.
- *Request ball.* For an $\mathsf{addr} \in [n]$, write tuple $(\mathsf{fetch}, \mathsf{addr}, \perp)$ to outgoing communication tape.
- *Move ball to register.* For some empty register $i \in [r]$ (i.e., for which $\mathsf{Reg}[i] = \emptyset$), set $\mathsf{Reg}[i] \leftarrow \mathsf{val}$, where val is the word currently on incoming communication tape. Erase val from the tape.

We present an adaptation of the [GO96] lower bound directly for the weakest *offline* simulation (corresponding to a stronger lower bound).

Theorem A.2 ([GO96] Theorem 6.2, rephrased)**.** *Every offline oblivious simulation of* $\mathsf{RAM}_{n,w,r}$ *by probabilistic-*$\mathsf{RAM}'_{n',w',r'}$ *operating in the Balls and Bins Model, for any* $w = w' \in \Omega(\log n)$*, must make at least* $\max\{|y|, \Omega(t \cdot \log t / \log r')\}$ *accesses on any input* $y = (\Pi, x)$*, where* $t = |\mathscr{A}ccess(\mathsf{RAM}_{n,w,r}, y)|$.

Proof. Suppose $\mathsf{RAM}'_{n',w',r'}$ is an offline oblivious simulation of $\mathsf{RAM}_{n,w,r}$. For sake of cleanliness, we omit the n, w, r subscripts and write simply RAM' and RAM. We begin by considering a simplified version, in which the simulation has:

- Perfect security: for any inputs y_1, y_2, we have $\mathscr{A}ccess(\mathsf{RAM}', y_1) \equiv \mathscr{A}ccess(\mathsf{RAM}', y_2)$.
- Perfect correctness: $\forall y$, $\Pr[\mathsf{RAM}'(y) = \mathsf{RAM}(y)] = 1$, over $\mathsf{CPU}'_{n',w',r'}$ randomness.
- Fixed bandwidth overhead: $|\mathscr{A}ccess(\mathsf{RAM}', y)|$ is a fixed function of $|\mathscr{A}ccess(\mathsf{RAM}, y)|$, for any y and any choice of randomness made by CPU'.

Consider the following collection of RAM programs that simply query and output a specified length-t sequence from the n data items, for fixed $t \geq n$. That is, for each length-t query sequence $Q = (q_1, \ldots, q_t) \in [n]^t$, we define program Π_Q *represented by the explicit sequence* $q_1, \ldots, q_t$. Note that these programs each fall within the category of *Fixed-Access* programs, and thus apply to the setting of *offline* oblivious simulation (see Definition 2.8). Further, for any x, the straightforward execution (without simulation) satisfies $|\mathscr{A}ccess(\mathsf{RAM}, (\Pi_Q, x))| = t$.

By assumption (3), for any data $x \in (\{0,1\}^w)^n$, and any such program Π_Q, we thus have that $|\mathscr{A}ccess(\mathsf{RAM}', (\Pi_Q, x))|$ is a fixed function $N(t)$ of t. Fix an arbitrary "*visible*" access sequence $V = (v_1, \ldots, v_N)$ in the support of $\mathscr{A}ccess(\mathsf{RAM}', (\Pi_{Q^*}, x))$, for some Q^*, x. By assumption (1), V must be in the support of $\mathscr{A}ccess(\mathsf{RAM}', (\Pi_Q, x))$ for *every* $Q \in [n]^t$. By assumption (2), it must be that RAM' produces the correct output $\mathsf{RAM}(\Pi_Q, x)$ conditioned on access pattern V. Namely, by making accesses specified by V, the simulation is able to correctly output the specified sequence of queried values $x[q_1], \ldots, x[q_t]$. The lower bound proceeds by a counting argument, showing that N must be sufficiently large in order for all n^t choices of Q to be answerable by the visible access pattern V.

For each visible access pattern (e.g., $V = (v_1, \ldots, n_N)$), consider the collection of possible "*hidden*" actions $H = (h_1, \ldots, h_N)$ that can be taken by CPU'. Recall that CPU' operates in the *balls and bins model*, as in Definition A.1. Each "Move ball to memory" operation $(\mathsf{store}, \mathsf{addr}, \mathsf{Reg}[i])$ and "Request ball" operation $(\mathsf{fetch}, \mathsf{addr}, \perp)$ made by CPU' induces a new visible access addr. Since the visible access sequence of addresses is fixed, each choice of addr is fixed. This leaves $(r+1)$ distinct choices for these two operations: r choices of registers $\mathsf{Reg}[i]$ from which to move a ball to memory, and a single choice for requesting the ball currently stored at addr. Between any two executions of "Move ball to memory" or "Request ball," CPU' further has the option to execute *once* the operation "Move ball to register." This allows $2(r+1)$ possible hidden actions h_i for each visible access v_i, resulting in $2^N(r+1)^N$ possible complete hidden action sequences H for any fixed visible N-access pattern V.

Finally, for any resulting hidden action sequence, consider the set of possible programs Π_Q (equivalently, t-query sequences $Q \in [n]^t$) that can be correctly simulated. We say that an action sequence $(v_1, h_1), \ldots, (v_N, h_N)$ *satisfies* a query sequence $q_1, \ldots, q_t$ if there exists a sequence $1 \leq j_1 \leq \ldots \leq j_t = q$ so that, for every "round" i $(1 \leq i \leq t)$, after actions $(v_1, h_1), \ldots, (v_{j_i}, h_{j_i})$, the ball corresponding to the ith queried data item $x[q_i]$ currently resides in one of the registers of CPU'. One can then see that a fixed sequence of actions $(v_1, h_1), \ldots, (v_N, h_N)$ can satisfy at most r^N request sequences $Q \in [n]^t$.

Putting these steps together, we see that any given visible access sequence $V = (v_1, \ldots, v_N)$ may satisfy *at most* $(2^N(r+1)^N)r^N$ request sequences $Q \in [n]^t$. Thus, in order to satisfy *all* such sequences (as required), it must be that

$$2^N(r+1)^N r^N \geq n^t.$$

That is, $N \in \Omega(t \log n / \log r)$.

Removing the above simplifying assumptions amounts to a slightly more detailed probability analysis. Perfect security (obliviousness) and correctness can be relaxed even to have *constant* probability of failure δ, because all we need is that there will be $O(n^t)$ many request sequences $Q \in [n]^t$ that share the same visible access pattern (with correctness). And, fixed bandwidth overhead can be directly relaxed, by instead analyzing expected overhead.

□

Simple Extensions of the [GO96] Bound

- *No repeat lookups.* A nearly identical bound holds even when restricting to programs that access each data item a single time in some permutation on $[n]$, resulting from $2^N(r+1)^N \geq n!$.
- *Given metadata "for free."* As noted in the original work [GO96], the lower bound holds even if the CPU is given oracle access to the addresses of each data ball in memory at no cost.

- *"Balls and Bins and Garbage."* The bound extends to a relaxed model in which the CPU may additionally copy and delete balls, and may spawn balls of a different "color" storing arbitrary information, as long as the final output contains only balls of the original color. Note that existing positive results in ORAM literature use such an approach, where helper non-data info is stored in memory to help later locate items (specifically, in recursive solutions).
- *"Server-side computation" in balls and bins.* Interestingly, the bandwidth bound extends also to the case where MEM' may perform local computation before communicating to CPU', if we demand statistical security, and if this computation is *also restricted to the balls and bins model* (Definition A.1). To be meaningful, we consider a natural extension of the Oblivious RAM model, where the sequence of addresses accessed by MEM' during its computations is also included within adversarial view. In such case, "server-side" shuffling of balls around in bins has absolutely no effect, since MEM' is still limited to sending intact memory values, and the adversary knows precisely which balls are eventually sent.

  This observation implies that the recent results of [AKST14, DvDF+15] are able to bypass the [GO96] ORAM bandwidth lower bound not only because they allow server-side computation, but rather because they also violate balls and bins (and obtain computational security).

B Omitted Material

This section contains discussions and proofs omitted from the body of the paper.

B.1 Useful Sub-Algorithms

Random Public Shuffle. We make use of the Knuth (or Fisher-Yates) Shuffle [Knu97], given as RandPerm in Algorithm 1.

Claim B.1 (RandPerm)**.** *[Knu97] The (randomized) algorithm* $\mathsf{RandPerm}(D, n)$ *terminates in* $O(n)$ *computation steps, and implements a random permutation* $\pi \leftarrow S_n$ *on the elements* $D[1], \ldots, D[n]$*.*

Algorithm 1 (Public) $\mathsf{RandPerm}(D, n)$

1: Inputs: start address to D, database size n.
2: **for** $i = 1, \ldots, n$ **do**
3: Sample $j \xleftarrow{\$} \{i, \ldots, n\}$.
4: Read $\mathsf{Reg}[1] \leftarrow D[i]$ into local memory.
5: Read $\mathsf{Reg}[2] \leftarrow D[j]$ into local memory.
6: Write $D[i] \leftarrow \mathsf{Reg}[2]$ and $D[j] \leftarrow \mathsf{Reg}[1]$.
7: **end for**

Bitwise Transpose. Algorithm 2 describes a basic recursive Transpose algorithm assuming $\Theta(1)$ CPU registers (see, e.g., [ACS87, FLPR99]). The algorithm performs each w-word transpose recursively, in $\log w$ iterations, each corresponding to swapping blocks of size $n/2^i$. Each recursion level requires reading and rewriting every word of data once, since words only affect one another in pairs at any given recursion level. The total computation and bandwidth of the algorithm are thus both $O(n \log w)$.

Claim B.2 (Transpose)**.** *Assuming $O(1)$ local CPU registers, the algorithm* Transpose(D, n, w) *makes $O(n \log w)$ memory word accesses, performs $O(n \log w)$ computation, and implements a bitwise transpose within each block of w words of the input database D.*

Proof. It suffices to argue correctness and $O(w \log w)$ complexity of each $w \times w$ transpose (Lines 4-13 of Algorithm 2). The complexity claim is clear: within each $\log w$ level of iteration, $O(w)$ work is performed, pulling words into local memory in pairs and performing simple bit swap operations on the words. To see correctness, note that after each level of iteration $j \in \{0, \ldots, \log w - 1\}$, the (ℓ, m)th bit in the original $w \times w$ array (i.e., $D[\ell][m]$) for which $\ell = \ell^{top}||\ell^{bot}$ and $m = m^{top}||m^{bot}$ (split at bit position j) now resides in position $D[\ell^{top}||m^{bot}][m^{top}||\ell^{bot}]$. Thus, after the final $(\log w - 1)$th iteration, each original bit $D[\ell][m]$ will lie in position $D[m][\ell]$, as desired. □

Remark B.3 (Transpose with $\Theta(w)$ CPU registers)**.** Note that if the CPU has $\Theta(w)$ local registers, then we may simplify the iterated transpose procedure: For each $w \times w$ transpose, read *all* w words into local CPU memory at once and transpose locally. This reduces the total bandwidth required in this step from $O(n \log w)$ down to $O(n)$ (but maintains $O(n \log w)$ computation).

Algorithm 2 Transpose(D, n, w) (with $O(1)$ local CPU registers)

```
1: Inputs: start address to D, database size n, word size w.
2: for i = 1 to n/w do    //Over all w-word blocks, transpose each block
3:     Let start ← wj;
4:     for j = 0 to (log w − 1) do    //Iteratively compute 2^j-square transposes
5:         for ℓ = 1 to w/2 do    //Pull words into memory & modify in pairs
6:             Read Reg[1] ← D[start + ℓ];
7:             Read Reg[2] ← D[start + ℓ + 2^j];
8:             Store temp copy of 1: temp ← Reg[1];
9:             For every bit index m ∈ [w] for which (m ∧ 2^j == 1), set Reg[1][m] ← Reg[2][m − 2^j];
10:            For every bit index m ∈ [w] for which (m ∧ 2^j == 0), set Reg[2][m] ← temp[m + 2^j];
11:            Write D[start + ℓ] ← Reg[1] back to memory
12:            Write D[start + ℓ + 2^j] ← Reg[2] back to memory
13:        end for
14:    end for
15: end for
```

SIMD Circuit Emulation via RAM. EmulateCircuit is given in Algorithm 3.

Note that the circuit description is written to data as an array C, where $C[i]$ is initialized with the information defining gate i (see Notation 3.4). EmulateCircuit takes two addresses, identifying the start of of the database D and of the circuit description array C.

Claim B.4 (EmulateCircuit)**.** *The algorithm* EmulateCircuit(D, C, s, n, m) *terminates in $O(s)$ time steps; at the conclusion of execution, the m data words* output $:= (C[s-m+1], \ldots, C[s])$ *correspond to the output of circuit C computed* bitwise *on n-word input* input $:= (D[1], \ldots, D[n])$*. That is,* output$_i = C($input$)$.

Proof. The time complexity claim follows by inspection: the first loop counts through 1 to n; the second through $(n + 1)$ to $(s - m)$, and the final through $(s - m + 1)$ to s. The first loop copies the n input words into the first n components of the array C. The second loop executes each

boolean gate of C bitwise on the pair of values generated so far in the partial computation at the appropriate indices i_L, i_R. Finally, the remaining loop copies the output values from their current locations within the computation to the final m indices of C. That is, the claim holds. □

Algorithm 3 $\mathsf{EmulateCircuit}(D, C, s, n, m)$

Inputs: start address to database D, start address to circuit description C, circuit size s, input length n, output length m.
for $i = 1$ to n **do** //Copy input
    Read $\mathsf{Reg}[1] \leftarrow D[i]$.
    Write $C[i] \leftarrow \mathsf{Reg}[1]$.
end for
for $i = (n+1)$ to $(s-m)$ **do** //Evaluate circuit
    Read $\mathsf{Reg}[1] \leftarrow C[i]$. Parse $\mathsf{Reg}[1] = (f, i_L, i_R)$.
    Read $\mathsf{Reg}[2] \leftarrow C[i_L]$.
    Read $\mathsf{Reg}[3] \leftarrow C[i_R]$
    Write $C[i] \leftarrow f(\mathsf{Reg}[2], \mathsf{Reg}[3])$. //Boolean gate f performed bitwise on words
end for
for $i = (s-m+1)$ to s **do** //Copy output
    Read $\mathsf{Reg}[1] \leftarrow C[i]$. Parse $\mathsf{Reg}[1] = i_{\mathsf{out}}$.
    Read $\mathsf{Reg}[2] \leftarrow C[i_{\mathsf{out}}]$.
    Write $C[i] \leftarrow \mathsf{Reg}[2]$.
end for

Remove Redundant Items in Sorted List. In the final stage of the OASort algorithm, we reach an array E that contains all items of the original database in sorted order, but with redundant overlapping items. The following procedure $\mathsf{RemRedundant}(E, n)$, given in Algorithm 4, provides a means for identifying and zeroing out these redundant items (to be removed in a proceeding step). Note that this process is facilitated when the items are guaranteed distinct; however, a modification of the protocol will enforce this generically even when beginning with a list of non-distinct items.

Algorithm 4 $\mathsf{RemRedundant}(E, n)$ One-pass: Remove redundant items

```
currentval ← −∞.
for i = 1 to w − 1 do      // For each (2n/w)-size block
    blockctr ← 0.
    for j = 1 to 2ℓ do      // Step through the block
        Read Reg[1] ← D[2iℓ + j].
        if (Reg[1] > currentval) && (blockctr < ℓ) then
            currentval ← Reg[1].
            blockctr++.
        else
            Zero out D[iℓ + j] ← −∞
        end if
    end for
end for
```

B.2 Omitted Correctness Proofs

Claim B.5 (OASort Correctness)**.** *For any sequence of* n distinct *initial values* $D = (D[1], \ldots, D[n])$*, then with probability* $1 - e^{-n^{\Omega(1)}}$*,* $\mathsf{OASort}(D, n)$ *outputs the values in sorted order* $D[1] \leq \cdots \leq D[n]$*.*

Proof. Consider the steps of OASort.

1. $\mathsf{RandPerm}(D, n)$. By Claim B.1, the n items of D are permuted via random $\pi \leftarrow S_n$.
2. Sorting in parallel. (a) $\mathsf{Transpose}(D, n, w)$: By Claim B.2, every block of w words of D is individually bitwise transposed. That is, within each w-block, the ith word ($i \in [w]$) now contains the concatenation of ith bits from each of the corresponding w words. (b) $\mathsf{EmulateCircuit}(D[\times w], C_{\mathsf{sort}}, s(n), n, n)$: By Claim B.4, together with the assumption that C_{sort} is a valid sorting circuit, we have at the conclusion of execution that D satisfies the following: for each bit position $i \in [w]$ within a word, and each $j \in [n/w]$, defining $D^{(i)}[j] := (D[wj+1]_i || D[wj+2]_i || \ldots || D[wj+w]_i)$, then $D^{(i)}[1] \leq D^{(i)}[2] \leq \cdots \leq D^{(i)}[n/w]$. (c) $\mathsf{Transpose}(D, n, w)$: By Claim B.2, we now have for each $i \in [w]$ that $D[i] \leq D[i+w] \leq D[i+2w] \leq \cdots \leq D[i+(n/w-1)w]$.
3. Merging blocks (with overlap). In (a), data is copied from D into the size-$2n$ array E with overlaps as illustrated in Figure 2. Step (b) directly mimics the parallel sort analyzed above, with the modification that words are first reordered from "same-colored" blocks to be interleaved, and then this is reversed at the very end. From the same analysis as before, together with the added reordering, this means that in effect each of the w contiguous "same-colored" $2n/w$-blocks is individually sorted: i.e., for every $i \in [w]$, we have $E[i\ell] \leq E[i\ell+1] \leq E[i\ell+2] \leq \cdots \leq E[i\ell+(\ell-1)]$ (recall $\ell = n/w$).

   We further claim the following property of E with respect to the original database D. Denote by $\tilde{D}$ the *correctly sorted* version of D (i.e., the target output), and recall that $\ell = n/w \in \omega(\sqrt{n})$. Then each (sorted) ℓ-block of $\tilde{D}$ is contained as a subsequence of the corresponding 2ℓ-block of E. That is, there exists a subsequence of indices $i_1 < i_2 < i_3 < \cdots < i_n \in [2n]$ for which $i_1, \ldots, i_\ell \leq 2\ell \leq i_{\ell+1}, \ldots, i_{2\ell} \leq 4\ell \leq i_{2\ell+1}$, etc, and $E[i_j] = \tilde{D}[j]\ \forall j \in [n]$.

   This follows directly from Claims 3.6 and 3.7, with the required error probability over the execution of RandPerm.
4. Remove redundant items.

   Since the items of the original data set D are assumed to be distinct, the one-pass procedure of $\mathsf{RemRedundant}(E, 2n)$ in Step (a) will necessarily identify *some* index sequence $i_1 < i_2 < \cdots < i_n \in [2n]$ satisfying the properties above, and set all elements of E outside of these indices to $-\infty$. Indeed, RemRedundant keeps precisely the first ℓ distinct values of each 2ℓ-block of E that have not appeared yet in the sequence.

   From the same analysis as above, the parallel block sort in Step 4(b) (repeating the procedure of Step 3(b)) will precisely sort each of the "same-colored" contiguous 2ℓ-blocks of E, in effect placing the ℓ elements of each 2ℓ block that were assigned to $-\infty$ during RemRedundant to the left-most ℓ positions of the block.

   Finally, this means that in the final one-pass step of Step 4(c), these $-\infty$ items are removed, leaving behind precisely the sorted elements of D.

□

Claim B.6 (OfflineORAM Correctness)**.** OfflineORAM *satisfies the correctness property of an offline ORAM compiler, for Fixed-Access programs.*

Proof. By the correctness of OASort, the following will happen with overwhelming probability in n. At the conclusion of Step 2, all entries of the matrix $M = D||S$ will be correctly sorted with respect to the key value $(\mathsf{index}, \mathsf{time})$: i.e., for each index $i \in [n]$, there will be a sequence beginning with the original database value entry $D[i]$, and proceeding with the subsequence of data requests $S[j]$ of the form $(\mathsf{index}, \mathsf{command})$ for $\mathsf{index} = i$, in order of increasing j (corresponding to the time order of the requests). Thus, at the conclusion of Step 3, each data request entry $S[j]$ will be fulfilled with the *correct* requested value (note this relies on the $Fixed - Access$ structure of the compiled program). Finally, after Step 4, the array M will be correctly sorted by key $(\mathsf{time}, \mathsf{index})$, returning it to the original order $M = D||S$ (since all items of D have $\mathsf{time} = 0$ and index equal to its originating index, and all items of S have sequential time values in $[n]$). Thus, the values $M[n+1]$ to $M[2n]$ will hold precisely the fulfilled requests from S. □