Foundations of Differentially Oblivious Algorithms*

T-H. Hubert Chan† Kai-Min Chung‡ Bruce M. Maggs§ Elaine Shi¶

Abstract

It is well-known that a program's memory access pattern can leak information about its input. To thwart such leakage, most existing works adopt the technique of oblivious RAM (ORAM) simulation. Such an obliviousness notion has stimulated much debate. Although ORAM techniques have significantly improved over the past few years, the concrete overheads are arguably still undesirable for real-world systems — part of this overhead is in fact inherent due to a well-known logarithmic ORAM lower bound by Goldreich and Ostrovsky. To make matters worse, when the program's runtime or output length depend on secret inputs, it may be necessary to perform worst-case padding to achieve full obliviousness and thus incur possibly super-linear overheads.

Inspired by the elegant notion of differential privacy, we initiate the study of a new notion of access pattern privacy, which we call "(ϵ, δ)-differential obliviousness". We separate the notion of (ϵ, δ)-differential obliviousness from classical obliviousness by considering several fundamental algorithmic abstractions including sorting small-length keys, merging two sorted lists, and range query data structures (akin to binary search trees). We show that by adopting differential obliviousness with reasonable choices of ϵ and δ, not only can one circumvent several impossibilities pertaining to full obliviousness, one can also, in several cases, obtain meaningful privacy with little overhead relative to the non-private baselines (i.e., having privacy "almost for free"). On the other hand, we show that for very demanding choices of ϵ and δ, the same lower bounds for oblivious algorithms would be preserved for (ϵ, δ)-differential obliviousness.

1 Introduction

Suppose that there is a database consisting of sensitive user records (e.g., medical records), and one would like to perform data analytics or queries over this dataset in a way that respects individual users' privacy. More concretely, we imagine the following two scenarios:

1. The database is encrypted and outsourced to an untrusted cloud server that is equipped with a trusted secure processor such as Intel's SGX [34, 2], such that only the secure processor can decrypt and compute over the data.

2. The database is horizontally partitioned across multiple nodes, e.g., each hospital holds records for their own patients.

To provide formal and mathematical guarantees of users' privacy, one natural approach is to require that *any information about the dataset that is disclosed during the computation must satisfy differential privacy* (DP). Specifically, differential privacy is a well-established notion first proposed in the ground-breaking work by Dwork et al. [14]. Naturally, in the former scenario, we can have the secure processor compute differentially private statistics to be released or differentially private answers to analysts' queries. In the latter scenario, since the data is distributed, we can rely on multi-party computation (MPC) [19, 45] to emulate a secure CPU, and compute a differentially private mechanism securely (i.e., revealing only the differentially private answer but nothing else). The above approaches (assuming that the program is executed in the RAM-model) indeed ensure that the statistics computed by the secure processor or the MPC protocol are safe to release. However, this is not sufficient for privacy: specifically, the program's execution behavior (in particular, *memory access patterns*) can nonetheless leak sensitive information.

Classical notion of access pattern privacy: full obliviousness. To defeat access pattern leakage, a line of work has focused on oblivious algorithms [23, 17, 32] and Oblivious RAM (ORAM) constructions [20, 18]. These works adopt "full obliviousness" as a privacy notion, i.e., the program's memory access patterns (including the length of the access sequence) must be indistinguishable regardless of the secret database or inputs to the program. Such a full obliviousness notion has at least the following drawbacks:

1. First, to achieve full obliviousness, a generic

*The full version of this paper is available online [10].

†The University of Hong Kong. hubert@cs.hku.hk.
‡Academia Sinica. kmchung@iis.sinica.edu.tw
§Duke University and Akamai Technologies. bmm@cs.duke.edu
¶Cornell University. runting@gmail.com

approach is to apply an Oblivious RAM (ORAM) compiler, an elegant algorithmic technique originally proposed by Goldreich and Ostrovsky [20, 18]. Although ORAM constructions have significantly improved over the past few years [38, 39, 43], their concrete performance is still somewhat undesirable — and some of this overhead is, in fact, inherent due to the well-known *logarithmic* ORAM lower bound by Goldreich and Ostrovsky [20, 18].

2. Second, to make matters worse, in cases where the program's output length or runtime also depends on the secret input, it may be necessary to pad the program's output length and runtime to the maximum possible to achieve full obliviousness. Such padding can sometimes incur even super-linear overhead, e.g., see our range query database example later in the paper.

Our new notion: differential obliviousness. Recall that our final goal is to achieve a notion of "end-to-end differential privacy", that is, any information disclosed (including any statistics explicitly released as well as the program's execution behavior) must be differentially private. Although securely executing an *oblivious* DP-mechanism would indeed achieve this goal, the full obliviousness notion appears to be an overkill. In this paper, we formulate a new notion of access pattern privacy called *differential obliviousness*. Differential obliviousness requires that if the memory access patterns of a program are viewed as a form of statistics disclosed, then such "statistics" must satisfy differential privacy too. Note that applying standard composition theorems of DP [16], the combination of statistics disclosed and access patterns would jointly be DP too (and thus achieving the aforementioned "end-to-end DP" goal).

Our differential obliviousness notion can be viewed as a relaxation of full obliviousness (when both are defined with information theoretic security). Clearly, such a relaxation is only interesting if it allows for significantly smaller overheads than full obliviousness. Indeed, with this new notion, we can hope to overcome both drawbacks for full obliviousness mentioned above. First, it might seem natural that with differential obliviousness, we can avoid worst-case padding which can be prohibitive. Second, even when padding is a non-issue (i.e., when the program's runtime and output length are fixed), an exciting question remains:

Can we asymptotically outperform full obliviousness with this new notion? In other words, *can we achieve differential obliviousness without relying on full obliviousness as a stepping stone*?

The answer to this question seems technically challenging. In the classical DP literature, we typically achieve differential privacy by adding noise to intermediate or output statistics [14]. To apply the same techniques here would require adding noise to a program's memory access patterns — this seems counter-intuitive at first sight since access patterns arise almost as a side effect of a program's execution.

Our results and contributions. Our paper shows non-trivial lower- and upper-bound results establishing that differential obliviousness is an interesting and meaningful notion of access pattern privacy, and can significantly outperform full obliviousness (even when padding is a non-issue). We show results of the following nature:

1. *New lower bounds on full obliviousness.* On one hand, we show that for several fundamental algorithmic building blocks (such as sorting, merging and range query data structures), any oblivious simulation must incur at least $\Omega(\log N)$ overhead where N is the data size. Our oblivious algorithm lower bounds can be viewed as a strengthening of Goldreich and Ostrovsky's ORAM lower bounds [20, 18]. Since the logarithmic ORAM lower bounds do not imply a logarithmic lower bound for any specific algorithm, our lower bounds (for specific algorithms) are necessary to show a separation between differential obliviousness and full obliviousness.

2. *Almost-for-free differentially oblivious algorithms.* On the other hand, excitingly we show for the first time that for the same tasks mentioned above, differentially oblivious algorithms exist which incur only $O(\log \log N)$ overhead (we sometimes refer to these algorithms as "almost-for-free").

3. *Separations between various definitional variants.* We explore various ways of defining differential obliviousness and theoretical separations between these notions. For example, we show an intriguing separation between ϵ-differential obliviousness and (ϵ, δ)-differential obliviousness. Specifically, just like ϵ-DP and (ϵ, δ)-DP, a non-zero δ term allows for a (negligibly) small probability of privacy failure. We show that interestingly, permitting a non-zero but negligibly small failure probability (i.e., a non-zero δ) turns out to be crucial if we would like to outperform classical full obliviousness! Indeed, our "almost-for-free" differential oblivious algorithms critically make use of this non-zero δ term.

   Intuitively, in ϵ-differential privacy, very little privacy is preserved for large values of ϵ. Hence,

it is surprising that most of our logarithmic full obliviousness lower bounds will still apply, even if we allow arbitrarily large ϵ for ϵ-differential obliviousness.

Both our lower bounds and upper bounds require novel techniques. Our lower bounds draw connections to the complexity of shifting graphs [37] that were extensively studied in the classical algorithms literature. For upper bounds, to the best of our knowledge, our algorithms show for the first time how to combine oblivious algorithms techniques and differential privacy techniques in non-blackbox manners to achieve non-trivial results. Our upper bounds also demonstrate a new algorithmic paradigm for constructing differentially oblivious algorithms: first we show how to make certain DP mechanisms oblivious and we rely on these oblivious DP mechanisms to compute a set of intermediate DP-statistics. Then, we design algorithms whose memory access patterns are "simulatable" with knowledge of these intermediate DP statistics — and here again, we make use of oblivious algorithm building blocks.

1.1 Differential Obliviousness We formulate differential obliviousness for random access machines (RAMs) where a trusted CPU with $O(1)$ registers interacts with an untrusted memory and performs computation. We assume that the adversary is able to observe the memory addresses the CPU reads and writes, but is unable to observe the contents of the data (e.g., the data is encrypted or secret-shared by multiple parties). This abstraction applies to both of the motivating scenarios described at the beginning of our paper.

Differential obliviousness can be intuitively interpreted as differential privacy [14, 41], but now the observables are access patterns. Informally, we would like to guarantee that an adversary, after having observed access patterns to (encrypted)[1] dataset stored on the server, learns approximately the same amount of information about an individual or an event as if this individual or event were not present in the dataset.

Basic definition of differential obliviousness. Let M be an algorithm that is expressed as a RAM program. We say that two input databases I and I' are neighboring iff they differ only in one entry. The algorithm M is said to be (ϵ, δ)-differentially oblivious, iff for any two *neighboring* input databases I and I', for any set S of access patterns, it holds that

(1.1)
$$\Pr[\mathbf{Accesses}^M(I) \in S] \leq e^\epsilon \cdot \Pr[\mathbf{Accesses}^M(I') \in S] + \delta,$$

where $\mathbf{Accesses}^M(I)$ denotes the ordered sequence of memory accesses made by the algorithm M upon receiving the input I. Therefore, (ϵ, δ)-differential obliviousness can be thought of as (ϵ, δ)-DP but where the observables are the access patterns.

The term δ can be thought of as a small probability of privacy failure that we are willing to tolerate. For all of our upper bounds, we typically require that δ be *negligibly small in some security parameter* λ. When $\delta = 0$, we also say that M satisfies *ϵ-differential obliviousness*.

Comparison with full obliviousness. It is interesting to contrast the notion of differential obliviousness with the classical notion of full obliviousness [20, 18]. An algorithm M (expressed as a RAM program) is said to be (statistically) δ-oblivious iff for any input databases I and I' of equal length, it holds that $\mathbf{Accesses}^M(I) \stackrel{\delta}{\equiv} \mathbf{Accesses}^M(I')$ where $\stackrel{\delta}{\equiv}$ denotes that the two distributions have statistical distance at most δ. When $\delta = 0$, we say that the algorithm M satisfies *perfect* obliviousness. Note that to satisfy the above definition requires that the length of the access sequence be identically distributed or statistically close for any input of a fixed length — as mentioned earlier, one way to achieve this is to pad the length/runtime to the worst case.

It is not difficult to observe that (ϵ, δ)-differential obliviousness is a relaxation of δ-obliviousness; and likewise ϵ-differential obliviousness is a relaxation of perfect obliviousness. Technically the relaxation arises from the following aspects:

1. First, differential obliviousness requires that the access patterns be close only for *neighboring* inputs; as the inputs become more dissimilar, the access patterns they induce are also allowed to be more dissimilar. By contrast, full obliviousness requires that the access patterns be close for any input of a fixed length.

2. Differential obliviousness permits a multiplicative e^ϵ difference in the distribution of the access patterns incurred by neighboring inputs (besides the δ failure probability); whereas full obliviousness does not permit this e^ϵ relaxation.

Later in the paper, we shall see that although ϵ-differential obliviousness seems much weaker than obliviousness, surprisingly the same logarithmic lower bounds pertaining to full obliviousness carry over

[1] Our differentially oblivious definitions do not capture the encryption part, since we consider only the access patterns as observables. In this way all of our guarantees are information theoretic in this paper.

to ϵ-differential obliviousness for several algorithmic abstractions we are concerned with. However, by additionally permitting a non-zero (but negligibly small) failure probability δ, we can achieve almost-for-free differentially oblivious algorithms.

Definition of differential obliviousness for stateful algorithms. We will also be concerned about *stateful* algorithms where the memory stores persistent state in between multiple invocations of the algorithm. Concretely, we will consider range-query data structures (akin to binary search trees), where the entries of a database can be inserted dynamically over time, and range queries can be made in between these insertions. In such a dynamic database setting, we will define an *adaptive* notion of differential obliviousness where the adversary is allowed to adaptively choose both the entries inserted into the database, as well as the queries — and yet we require that the access patterns induced be "close" by Equation (1.1) for any two neighoring databases (inserted dynamically).[2] Our notion of adaptive differential obliviousness is akin to the standard adaptive DP notion for dynamic databases [16], but again our observables are now memory access patterns rather than released statistics. We defer the full definition to later technical sections.

1.2 Our Results Equipped with the new differentially oblivious notion, we will now try to understand the following questions: 1) does differential obliviousness permit asymptotically faster algorithms than full obliviousness? 2) how do the choices of ϵ and δ affect the asymptotical performance of differentially oblivious algorithms? To this end, we consider a few fundamental algorithmic abstractions including sorting, merging, and data structures — these algorithmic abstractions were not only extensively studied in the algorithms literature, but also heavily studied in the ORAM and oblivious algorithms literature as important building blocks.

1.2.1 Sorting We consider (possibly non-comparison-based) sorting in the *balls-and-bins* model: imagine that there are N balls (i.e., records) each tagged with a k-bit key. We would like to sort the balls based on the relative ordering of their keys.[3] If how an algorithm moves elements is based only on the relative order (with respect to the keys) of the input elements, we say that the algorithm is *comparison-based*; otherwise it is said to be *non-comparison-based*. Unlike the keys, the balls are assumed to be opaque — they can only be moved around but cannot be computed upon. A sorting algorithm is said to be *stable* if for any two balls with identical keys, their relative order in the output respects that in the input.

First, even without privacy requirements, it is understood that 1) any *comparison-based* sorting algorithm must incur at least $\Omega(N \log N)$ comparison operations — even for sorting 1-bit keys due to the well-known 0-1 principle; and 2) for special scenarios, *non-comparison-based* sorting techniques can achieve linear running time (e.g., radix sort, counting sort, and others [3, 29, 25, 24, 40]) — and a subset of these techniques apply to the balls-and-bins model. A recent manuscript by Lin, Shi, and Xie [31] showed that interesting barriers arise if we require full obliviousness for sorting:

FACT 1.1. (BARRIERS FOR OBLIVIOUS SORTING [31]) *Any oblivious 1-bit stable sorting algorithm in the balls-and-bins model, even non-comparison-based ones, must incur at least $\Omega(N \log N)$ runtime (even when allowing a constant probability of security or correctness failure). As a direct corollary, any general oblivious sorting algorithm in the balls-and-bins model, even non-comparison-based ones, must incur at least $\Omega(N \log N)$ runtime.*

We stress that the above oblivious sorting barrier is applicable only in the balls-and-bins model (otherwise without the balls-and-bins constraint, the feasibility or infeasibility of $o(n \log n)$-size circuits for sorting remains open [7]). Further, as Lin, Shi, and Xie showed [31], for small-length keys, the barrier also goes away if the stability requirement is removed (see Section 1.3).

Differentially oblivious sorting. Can we use the differential obliviousness relaxation to overcome the above oblivious sorting barrier (in the balls-and-bins model)? We show both upper and lower bounds. For upper bounds, we show that for choices of ϵ and δ that give reasonable privacy, one can indeed sort small-length keys in $o(N \log N)$ time and attain (ϵ, δ)-differential obliviousness. As a typical parameter choice, for $\epsilon = \Theta(1)$ and δ being a suitable negligible function in N, we can *stably* sort N balls tagged with 1-bit keys in $O(N \log \log N)$ time. Note that in this case, the best non-private algorithm takes linear time, and thus we show that privacy is attained "almost for free" for 1-bit stable sorting. More generally, for any $k = o(\log N / \log \log N)$, we can stably sort k-bit keys in $o(N \log N)$ time — in other words, for small-length keys

[2] Roughly, we say two dynamic databases are neighboring if the sequence of insert and query operations differ in at most one position; see Section 6 for the precise definition of neighboring.

[3] For example, if the key is 1-bit, a non-balls-and-bins algorithm could just count the number of 0s and 1s and write down an answer; but a balls-and-bins algorithm would have to sort the balls themselves.

we overcome the $\Omega(N \log N)$ barrier of oblivious sorting.

We state our result more formally and for generalized parameters:

THEOREM 1.1. ((ϵ, δ)-DIFFERENTIALLY OBLIVIOUS STABLE k-BIT SORTING) *For any $\epsilon > 0$ and any $0 < \delta < 1$, there exists an (ϵ, δ)-differentially oblivious k-bit stable sorting algorithm that completes in $O(kN(\log \frac{k}{\epsilon} + \log\log N + \log\log \frac{1}{\delta}))$ runtime. As a special case, for $\epsilon = \Theta(1)$, there exists an $(\epsilon, \mathsf{negl}(N))$-differentially oblivious stable 1-bit sorting algorithm that completes in $O(N \log\log N)$ runtime for some suitable negligible function $\mathsf{negl}(\cdot)$, say, $\mathsf{negl}(N) := \exp(-\log^2 N)$.*

Note that the above upper bound statement allows for general choices of ϵ and δ. Interestingly, we show that our upper bound result is *optimal* up to $\log\log$ factors for a wide parameter range. We present our lower bound statement for general parameters first, and then highlight several particularly interesting parameter choices and discuss their implications. Note that our lower bound below is applicable even to non-comparison-based sorting:

THEOREM 1.2. (LIMITS OF (ϵ, δ)-DIFFERENTIALLY OBLIVIOUS STABLE SORTING IN THE BALLS-AND-BINS MODEL) *For any $0 < s \le \sqrt{N}$, any $\epsilon > 0$, and any $0 \le \delta \le e^{-(2\epsilon s + \log^2 N)}$, any (ϵ, δ)-differentially oblivious stable 1-bit sorting algorithm in the balls-and-bins model must incur, on some input, at least $\Omega(N \log s)$ memory accesses with high probability.*[4]

As a corollary, under the same parameters, any (ϵ, δ)-differentially oblivious $\Omega(\log N)$-bit-key balls-and-bins sorting algorithm, even a non-stable one, must incur, on some input, at least $\Omega(N \log s)$ memory accesses with high probability.

First note that the lower bound tightly matches the upper bound (up to $\log\log$ factors) for $\epsilon = \Theta(1)$ and typical choices of δ, e.g., $\delta = \exp(-\log^2 N)$ or $\delta = \exp(-N^{0.1})$. Second, the lower bound allows a tradeoff between ϵ and δ. For example, if $\epsilon = \Theta(\frac{1}{\sqrt{N}})$, then we rule out $o(N \log N)$ stable 1-bit sorting for even $\delta = \exp(-\Omega(\log^2 N))$.

The case of $\delta = 0$ is more interesting: if δ is required to be 0, then *even when ϵ may be arbitrarily large*, any ϵ-differentially oblivious stable sorting algorithm must suffer from the same lower bounds as oblivious sorting (in the balls-and-bins model)! This is a surprising conclusion because in some sense, very little privacy (or almost no privacy) is attained for large choices of ϵ — and yet if δ must be 0, the same barrier for full obliviousness carries over!

[4] All lower bounds in this paper can be extended to handle imperfect correctness as we show in the full verison [10].

1.2.2 Merging Two Sorted Lists

Merging is also a classical abstraction and has been studied extensively in the algorithms literature (e.g., [30]). Merging in the balls-and-bins model is the following task: given two input sorted arrays (by the keys) which together contain N balls, output a merged array containing balls from both input arrays ordered by their keys. Without privacy requirements, clearly merging can be accomplished in $O(N)$ time. Interestingly, Pippenger and Valiant [37] proved that *any oblivious algorithm must (in expectation) incur at least $\Omega(N \log N)$ ball movements to merge two arrays of length N* — even when $O(1)$ correctness or security failure is allowed[5].

Differentially oblivious merging. Since merging requires that the input arrays be sorted, we clarify the most natural notion of "neighboring": by the most natural definition, two inputs (I_0, I_1) and (I'_0, I'_1) are considered neighboring if for each $b \in \{0, 1\}$, $\mathsf{set}(I_b)$ and $\mathsf{set}(I'_b)$ differ in exactly one record. Given this technical notion of neighboring, differential obliviousness is defined for merging in the same manner as before.

We show similar results for merging as those for 1-bit stable sorting as stated in the following informal theorems.

THEOREM 1.3. (LIMITS OF (ϵ, δ)-DIFFERENTIALLY OBLIVIOUS MERGING IN THE BALLS-AND-BINS MODEL) *For any $0 < s \le \sqrt{N}$, any $\epsilon > 0$, and any $0 \le \delta \le e^{-(2\epsilon s + \log^2 N)}$, any (ϵ, δ)-differentially oblivious merging algorithm in the balls-and-bins model must incur, on some input, at least $\Omega(N \log s)$ memory accesses with high probability.*

THEOREM 1.4. ((ϵ, δ)-DIFFERENTIALLY OBLIVIOUS MERGING) *For any $\epsilon > 0$ and any $0 < \delta < 1$, there exists an (ϵ, δ)-differentially oblivious merging algorithm that completes in $O(N(\log \frac{1}{\epsilon} + \log\log N + \log\log \frac{1}{\delta}))$ runtime. As a special case, for $\epsilon = \Theta(1)$, there exists an $(\epsilon, \mathsf{negl}(N))$-differentially oblivious merging algorithm that completes in $O(N \log\log N)$ runtime for some suitable negligible function $\mathsf{negl}(\cdot)$.*

The above theorems are stated for general choices of ϵ and δ, below we point out several notable special cases:

[5] Pippenger and Valiant's proof [37] is in fact in a balls-and-bins circuit model, but it is not too difficult, using the access pattern graph approach in our paper, to translate their lower bound to the RAM setting.

1. First, assuming $\epsilon = \Theta(1)$, if δ must be subexponentially small, then the same lower bound for oblivious merging will be preserved for (ϵ, δ)-differentially oblivious merging.
2. Second, for $\epsilon = \Theta(1)$ and δ negligibly small (but not subexponentially small), we can achieve (ϵ, δ)-differentially oblivious merging in $O(N \log \log N)$ time — yet another example of having privacy "almost-for-free".
3. Third, just like the case of 1-bit stable sorting, both our upper and lower bounds are (almost) tight for a wide parameter range that is of interest.
4. Finally, when $\delta = 0$, surprisingly, the $\Omega(N \log N)$ barrier for oblivious merging will be preserved no matter how large ϵ is (and how little privacy we get from such a large ϵ).

1.2.3 Data Structures Data structures are *stateful* algorithms, where memory states persist across multiple invocations. Data structures are also of fundamental importance to computer science. We thus investigate the feasibilities and infeasibilities of efficient, differentially oblivious data structures. Our upper bounds work for *range query* data structures: in such a data structure, one can make insertion and range queries over time, where each insertion specifies a record tagged with a numerical key, and each range query specifies a range and should return all records whose keys fall within the range. Our lower bounds work for *point query* data structures that are basically the same as range query data structures but each range query must be "equality to a specific key" (note that restricting the queries makes our lower bounds stronger).

The technical definition of differential obliviousness for stateful algorithms is similar to the earlier notion for stateless algorithms. We shall define a *static* notion and an *adaptive* notion — the static notion is used in our lower bounds and the adaptive notion is for our upper bounds (this makes both our lower and upper bounds stronger):

- *Static notion*: here we assume that the adversary commits to an insertion and query sequence upfront;
- *Adaptive notion*: here we assume that the adversary can adaptively choose insertions and range queries over time after having observed previous access pattern to the data structure. Our adaptive notion is equivalent to the standard adaptive DP notion for dynamic datasets [16] except that in our case, the observables are memory access patterns.

We defer the full definitions to the main technical sections. We note that for both the static and adaptive versions, as in the standard DP literature, we assume that the data records are private and need to be protected but the queries are public (in particular the standard DP literature considers the queries as part of the DP mechanisms [16]).

The issue of length leakage and comparison with oblivious data structures. Recall for the earlier sorting and merging abstractions, the output length is always fixed (assuming the input length is fixed). For range query data structures, however, an additional issue arises, i.e., the number of records returned can depend on the query and the database itself. Such length disclosure can leak secret information about the data records.

In the earlier line of work on oblivious data structures [44, 28, 35] and ORAM [20, 39, 43, 22, 18], this length leakage issue is somewhat shoved under the rug. It is understood that to achieve full obliviousness, we need to pad the number of records returned to the maximum possible, i.e., as large as the database size — but this will be prohibitive in practice. Many earlier works that considered oblivious data structures [44, 28, 35, 20, 39, 43, 22, 18] instead allow length leakage to avoid worst-case padding.

In comparison, in some sense our differential obliviousness notion gives a way to reason about such length leakage. By adopting our notion, one can achieve meaningful privacy by adding (small) noise to the output length, and without resorting to worst-case padding that can cause linear blowup.

Upper bound results. As mentioned, our upper bounds work for range query data structures that support *insertion* and *range queries*. Besides the standard overhead metrics, here we also consider an additional performance metric, that is, *locality* of the data accesses. Specifically we will use the number of discontiguous memory regions required by each query to characterize the locality of the data structure, a metric frequently adopted by recent works [9, 5, 4].

As a baseline, without any privacy requirement, such a range query data structure can be realized with a standard binary search tree, where each insertion incurs $O(\log N)$ time where N is an upper bound on the total records inserted; and each range query can be served in $O(\log N + L)$ time and accessing only $O(\log N)$ discontiguous memory regions where L denotes the number of matching records. We show the following results (stated informally).

THEOREM 1.5. ((ϵ, δ)-DIFFERENTIALLY OBLIVIOUS DATA STRUCTURES) *Suppose that* $\epsilon = \Theta(1)$ *and that* $\mathsf{negl}(\cdot)$ *is a suitable negligible function. There is an* $(\epsilon, \mathsf{negl}(N))$*-differentially oblivious data structure supporting insertions and range queries, where each of*

the N insertions incurs amortized $O(\log N \log\log N)$ runtime, and each query costs $O(\text{poly}\log N + L)$ runtime where L denotes the number of matching records, and requires accessing only $O(\log N)$ discontiguous memory regions regardless of L.

The best way to understand our upper bound results is to contrast with oblivious data structures [44, 28, 35] and the non-private baseline:

1. We asymptotically outperform known oblivious data structures that have logarithmic (multiplicative) overheads [44, 28, 35] (even when length leakage is permitted). Our algorithms are again "almost-for-free" in comparison with the non-private baseline mentioned earlier for both insertions and for queries that match sufficiently many records, i.e., when $L \geq \text{poly}\log N$.

2. We address the issue of length leakage effectively by adding polylogarithmic noise to the number of matching records; whereas full obliviousness would have required padding to the maximum (and thus incurring linear overhead).

3. Our constructions achieve logarithmic locality for range queries whereas almost all known oblivious data structures or ORAM techniques require accessing $\Omega(L)$ discontiguous regions of memory if the answer is of size L.

4. Finally, although not explicitly stated in the above theorem, it will be obvious later that our constructions are also *non-interactive* when applied to a client-server setting (assuming that the server is capable of performing computation). By contrast, we do not know of any oblivious data structure construction that achieves *statistical* security and non-interactivity at the same time.

In our detailed technical sections we will also discuss applications of our differentially oblivious data structures in designated-client and public-client settings.

Lower bounds. In the context of data structures, we also prove lower bounds to demonstrate the price of differential obliviousness. As mentioned, for our lower bounds, we consider point queries which is a special case of range queries; further, we consider static rather than adaptive differential obliviousness — these make our lower bound stronger. We prove the following theorem.

THEOREM 1.6. (LIMITS OF (ϵ, δ)-DIFFERENTIALLY OBLIVIOUS DATA STRUCTURES) *Suppose that $N = \text{poly}(\lambda)$ for some fixed polynomial $\text{poly}(\cdot)$. Let the integers $r < s \leq \sqrt{N}$ be such that r divides s; furthermore, let $\epsilon > 0$ and $0 \leq \delta \leq e^{-(2\epsilon s + \log^2 N)}$. Suppose that $\mathbb{DS}$ is a perfectly correct and (ϵ, δ)-differentially oblivious data structure supporting point queries. Then, there exists an operational sequence with N insertion and $k := \frac{N}{r}$ query operations interleaved, where each of k distinct keys from the domain $\{0, 1, \ldots, k-1\}$ is inserted r times, such that the total number of accesses $\mathbb{DS}$ makes for serving this sequence is $\Omega(N \log \frac{s}{r})$ with probability at least $1 - \mathsf{negl}(N)$.*

In the example of Theorem 1.6, the total number of items returned is N. Hence, one immediate observation we can draw is that our earlier range query upper bound (Theorem 1.5) is optimal up to $\log\log N$ factors assuming that the number of records matching the query is at least polylogarithmic in size. Moreover, the parameters r and k also reflect the intuition that the difficult case should be when the number k of distinct keys is large; in the extreme case when $r = s = \sqrt{N}$, we only have the trivial lower bound $\Omega(N)$. We defer more detailed technical discussions and proofs to the full version [10].

1.3 Closely Related Work

We are inspired by the recent work of Kellaris et al.[27]. They also consider differential privacy for access patterns for range query databases. In comparison, our work is novel in the following respects:

- Kellaris et al. [27] present a *computational* differential privacy definition for the specific application of *statically* outsourced databases in a client-server setting.

  In comparison, our differential obliviousness is more general and is defined for any (stateless and stateful) algorithms in the RAM model; and for stateful algorithms, we define an adaptive notion of differential obliviousness. Although Kellaris et al. also describe a construction for dynamic databases, they lack formal definitions for this case, and they implicitly assume that the client can store an unbounded amount of data and that metadata operations are for free — in our model where metadata storage and retrieval is no longer for free, their dynamic database scheme would incur on average $\Omega(N)$ cost per query, where N is the database size.

- Second, to support a dynamic range (or point) query database, Kellaris et al. rely on a blackbox ORAM and add noise to the result length. This approach is at least as expensive as generic ORAMs, and thus they do not answer the main question in our paper,

that is, can we achieve differential obliviousness without incurring the cost of generic ORAM or oblivious algorithms.

Another closely related work is by Wagh et al. [42], where they proposed a notion of differentially private ORAM — in their notion, neighboring is defined over the sequence of logical memory requests over time for a generic RAM program. Wagh et al. can rely on composition theorems to support small distances in memory access due to neighboring changes to the input. Their main algorithm changes the way Path ORAM [39] assigns blocks to random paths: they propose to make such assignments using non-uniform distributions to reduce the stash — and thus their approach can only achieve constant-factor savings in comparison with Path ORAM. In comparison, our notion compares the access patterns of a RAM program on neighboring inputs — this notion is more natural but the downside is that the notion makes sense only for databases where entries correspond to individuals, events, or other reasonable privacy units.

Lin, Shi, and Xie [31] recently showed that N balls each tagged with a k-bit key can be *obliviously* sorted in $O(kN \log\log N/\log k)$ time using non-comparison-based techniques — but their algorithm is not *stable*, and as Theorem 1.1 explains, this is inevitable for oblivious sort. Our results for sorting small-length keys differentially obliviously match Lin et al. [31] in asymptotical performance (up to log log factors) but we additionally achieve *stability*, and thus circumventing known barriers pertaining to oblivious sort.

Mazloom and Gordon [33] introduce a notion of secure multi-party computation allowing differentially private leakage (including differentially-private access pattern leakage). They then show how to design an efficient protocol, under this notion, for graph-parallel computations. At the time of our writing, Mazloom and Gordon [33] had results that achieved constant-factor improvement over the prior work GraphSC [36] that achieved full security. Subsequently, they improved their result to obtain asymptotical gains (see their latest version online [36]). Although the two papers investigate related notions, the definitions are technically incomparable since theirs focuses on defining security for multi-party computation allowing differentially private leakage (part of which can be access pattern leakage). Their work also considers parallelism in the computation model whereas our paper focuses on a sequential model of computation[6].

[6] A chronological note: Elaine Shi is grateful to Dov Gordon for bringing to her attention a relaxed notion of access pattern privacy back 3 years ago when she was in UMD. See "Acknowledgments" section for additional thanks to Dov Gordon.

2 Definitions

2.1 Model of Computation Abstractly, we consider a standard Random-Access-Machine (RAM) model of computation that involves a CPU and a memory. We assume that the memory allows the CPU to perform two types of operations: 1) read a value from a specified physical address; and 2) write a value to a specified physical address. In a cloud outsourcing scenario, one can think of the CPU as a *client* and the memory as the *server* (which provides only storage but no computation); therefore, in the remainder of the paper, we often refer to the CPU as the client and the memory as the server.

A (possibly stateful) program in the RAM model makes a sequence of memory accesses during its execution. We define a (possibly stateful) program's *access patterns* to include the *ordered* sequence of physical addresses accessed by the program as well as whether each access is a read or write operation.

2.1.1 Algorithms in the Balls-and-Bins Model In this paper, we consider a set of classical algorithms and data structures in the balls-and-bins model (note that data structures are stateful algorithms.) The inputs to the (possibly stateful) algorithm consist of a sequence of balls each tagged with a key. Throughout the paper, we assume that arbitrary computation can be performed on the keys, but the balls are opaque and can only be moved around. Each ball tagged with its key is often referred to as an *element* or a *record* whenever convenient. For example, a record can represent a patient's medical record or an event collected by a temperature sensor.

Unless otherwise noted, we assume that the RAM's word size is large enough to store its own address as well as a record (including the ball and its key). Sometimes when we present our algorithms, we may assume that the RAM can operate on real numbers and sample from certain distributions at unit cost — but in all cases these assumptions can eventually be removed and we can simulate real number arithmetic on a finite-word-width RAM preserving the same asymptotic performance (and absorbing the loss in precision into the δ term of (ϵ, δ)-differential obliviousness). We defer discussions on simulating real arithmetic on a finite-word-width RAM to the Appendices.

2.1.2 Additional Assumptions We make the following additional assumptions:

- We consider possibly *randomized* RAM programs — we assume that whenever needed, the CPU has

access to private random coins that are unobservable by the adversary. Throughout the paper, unless otherwise noted, for any randomized algorithm we require *perfect correctness*[7].

- Henceforth in this paper, we assume that *the CPU can store $O(1)$ records in its private cache.*

2.2 Differentially Oblivious Algorithms and Oblivious Algorithms We first define differential obliviousness for stateless algorithms. Suppose that $M(\lambda, I)$ is a stateless algorithm expressed as a RAM program. Further, M takes in two inputs, a security parameter λ and an input array (or database) denoted I. We say that two input arrays I and I' are neighboring iff they are of the same length and differ in exactly one entry.

DEFINITION 2.1. (DIFFERENTIALLY OBLIVIOUS (STATELESS) ALGORITHMS) *Let ϵ, δ be functions in a security parameter λ. We say that the stateless algorithm M satisfies (ϵ, δ)-differential obliviousness, iff for any neighboring inputs I and I', for any λ, for any set S of access patterns, it holds that*

$$\Pr[\mathbf{Accesses}^M(\lambda, I) \in S] \leq e^{\epsilon(\lambda)} \cdot \Pr[\mathbf{Accesses}^M(\lambda, I') \in S] + \delta(\lambda),$$

where $\mathbf{Accesses}^M(\lambda, I)$ is a random variable denoting the ordered sequence of memory accesses the algorithm M makes upon receiving the input λ and I.

In the above, the term δ behaves somewhat like a failure probability, i.e., the probability of privacy failure for any individual's record or any event. For our upper bounds subsequently, we typically would like δ to be a negligible function in the security parameter λ, i.e., every individual can rest assured that as long as λ is sufficiently large, its own privacy is unlikely to be harmed. On the other hand, we would like ϵ not to grow w.r.t. λ, and thus a desirable choice for ϵ is $\epsilon(\lambda) = O(1)$ — e.g., we may want that $\epsilon = 1$ or $\epsilon = \frac{1}{\log \lambda}$.

We also present the classical notion of oblivious algorithms since we will later be concerned about showing separations between differential obliviousness and classical obliviousness.

DEFINITION 2.2. (OBLIVIOUS (STATELESS) ALGORITHMS) *We say that the stateless algorithm M satisfies δ-statistical obliviousness, iff for any inputs I and I' of equal length, for any λ, it holds that $\mathbf{Accesses}^M(\lambda, I) \overset{\delta(\lambda)}{\equiv} \mathbf{Accesses}^M(\lambda, I')$, where $\overset{\delta(\lambda)}{\equiv}$ denotes that the two distributions have at most $\delta(\lambda)$ statistical distance. For the $\delta = 0$ special case, we say that M is perfectly oblivious.*

It is not hard to see that if an algorithm M is δ-statistically oblivious, it must also be (ϵ, δ)-differentially oblivious. In other words, (ϵ, δ)-differentially obliviousness is a strict relaxation of δ-statistical obliviousness. Technically speaking, the relaxation comes from two aspects: 1) differential obliviousness requires that the access patterns be close in distribution only for *neighboring* inputs; and the access patterns for inputs that are dissimilar are allowed to be more dissimilar too; and 2) differential obliviousness additionally allows the access pattern distributions induced by neighboring inputs to differ by an e^ϵ multiplicative factor.

Definitions for stateful algorithms. So far, our definitions for differential obliviousness and obliviousness focus on stateless algorithms. Later in our paper, we will also be interested in differentially oblivious data structures. Data structures are stateful algorithms where memory states persist in between multiple invocations. The definition of differential obliviousness is somewhat more subtle for data structures, especially when the adversary can adaptively choose the entries to insert into the data structure, and adaptively choose the queries as well. For readability, we defer defining differentially oblivious data structures (i.e., stateful algorithms) to later technical sections.

3 Differentially Oblivious Sorting: Upper Bounds

We consider sorting in the balls-and-bins model: given an input array containing N opaque balls each tagged with a key from a known domain $[K]$, output an array that is a permutation of the input such that all balls are ordered by their keys. If the sorting algorithm relies only on comparisons of keys, it is said to be *comparison-based*. Otherwise, if the algorithm is allowed to perform arbitrary computations on the keys, it is said to be *non-comparison-based*.

As is well-known, comparison-based sorting must suffer from $\Omega(N \log N)$ runtime (even without privacy requirements) and there are matching $O(N \log N)$ oblivious sorting algorithms [21, 1]. On the other hand, non-private, non-comparison-based sorting algorithms can sort N elements (having keys in a universe of cardinality $O(N)$) in linear time (e.g., counting sort).

In this section, we will show that for certain cases of sorting, the notions of differential obliviousness and

[7] Jumping ahead, given an (ϵ, δ)-differentially oblivious algorithm that incurs δ' correctness error, as long as the algorithm can detect its own error during computation, it can be converted into an algorithm that is perfectly correct and $(\epsilon, \delta + \delta')$-differentially oblivious: specifically, if an error is encountered, the algorithm simply computes and outputs a non-private answer.

obliviousness result in a separation in performance.

3.1 Stably Sorting 1-Bit Keys We start with stably sorting 1-bit keys and later extend to more bits. Stable 1-bit-key sorting is the following problem: given an input array containing N balls each tagged with a key from $\{0, 1\}$, output a *stably* sorted permutation of the input array. Specifically, stability requires that if two balls have the same key, their relative ordering in the output must respect their ordering in the input.

We choose to start with this special case because interestingly, stable 1-bit-key sorting in the balls-and-bins model has a $\Omega(N \log N)$ lower bound due to the recent work by Lin, Shi, and Xie [31] — and the lower bound holds even for non-comparison-based sorting algorithms that can perform arbitrary computation on keys. More specifically, they showed that for any constant $0 < \delta < 1$ any δ-oblivious stable 1-bit-key sorting algorithm must in expectation perform at least $\Omega(N \log N)$ ball movements.

In this section, we will show that by adopting our more relaxed differential obliviousness notion, we can circumvent the lower bound for oblivious 1-bit-key stable (balls-and-bins) sorting. Specifically, for a suitable negligible function δ and for $\epsilon = \Theta(1)$, we can accomplish (ϵ, δ)-differentially oblivious 1-bit-key stable sorting in $O(N \log \log N)$ time. Unsurprisingly, our algorithm is non-comparison-based, since due to the 0-1 principle, any comparison-based sorting algorithm, even for 1-bit keys, must make at least $\Omega(N \log N)$ comparisons.

3.1.1 A Closely Related Abstraction: Tight Stable Compaction Instead of constructing stable 1-bit-key sorting algorithm directly, we first construct a *tight stable compaction* algorithm: given some input array, tight stable compaction outputs an array containing only the 1-balls contained in the input, padded with dummies to the input array's size. Further, we require that the relative order of appearance of the 1-balls in the output respect the order in the input.

Given a tight stable compaction algorithm running in time $t(N)$, we can easily realize a stable 1-bit-key sorting algorithm that completes in time $O(t(N) + N)$ in the following way:

1. Run tight stable compaction to stably move all 0-balls to the front of an output array — let X be the resulting array;
2. Run tight stable compaction to stably move all 1-balls to the end of an output array — let Y be the resulting array (note that this can be done by running tight stable compaction on the reversed input array, and then reversing the result again);
3. In one synchronized scan of X and Y, select an element at each position from either X or Y and write it into an output array.

Moreover, if each instance of tight stable compaction is (ϵ, δ)-differentially oblivious, then the resulting 1-bit-key stable sorting algorithm is $(2\epsilon, 2\delta)$-differentially oblivious.

3.1.2 Intuition Absent privacy requirements, clearly tight stable compaction can be accomplished in linear time, by making one scan of the input array, and writing a ball out whenever a real element (i.e., the 1-balls) is encountered. In this algorithm, there are two pointers pointing to the input array and the output array respectively. Observing how fast these pointers advance allows the adversary to gain sensitive information about the input, specifically, whether each element is real or dummy. Our main idea is to approximately simulate this non-private algorithm, but obfuscate how fast each pointer advances just enough to obtain differential obliviousness. To achieve this we need to combine oblivious algorithms building blocks and differential privacy mechanisms.

First, we rely on batching: we repeatedly read a small batch of s elements into a working buffer (of size $O(s)$), obliviously sort the buffer to move all dummies to the end, and then emit some number of elements into the output. Note that the pointers to the input and output array could still reveal information about the number of non-dummy elements in the batchs read so far. Thus, the challenge is to determine how many elements must be output when the input scan reaches position i. Now, suppose that we have a building block that allows us to differentially privately estimate how many real elements have been encountered till position i in the input for every such i — earlier works on differentially private mechanisms have shown how to achieve this [12, 13, 15]. For example, suppose we know that the number of real elements till position i is in between $[C_i - s, C_i + s]$ with high probability, then our algorithm will know to output exactly $C_i - s$ elements when the input array's pointer reaches position i. Furthermore, at this moment, at most $2s$ real elements will have been scanned but have not been output — and these elements will remain in the working buffer. We can now rely on oblivious sorting again to truncate the working buffer and remove dummies, such that the working buffer's size will never grow too large — note that this is important since otherwise obliviously sorting the working buffer will become too expensive. Below we elaborate on how to make this idea fully work.

3.1.3 Preliminary: Differentially Private Prefix Sum Dwork et al. [15] and Chan et al. [12, 13] proposed a differentially private algorithm for computing all N prefix sums of an input stream containing N elements where each element is from $\{0, 1\}$. In our setting, we will need to group the inputs into bins and then adapt their prefix sum algorithm to work on the granularity of bins.

THEOREM 3.1. (DIFFERENTIALLY PRIVATE PREFIX SUM [12, 13]) *For any $\epsilon, \delta > 0$, there exists an (ϵ, δ)-differentially private algorithm, such that given a stream in $\mathbb{Z}_+^N$ (where neigboring streams differs in at most one position with difference at most 1), the algorithm outputs the vector of all N prefix sums, such that*

- *Any prefix sum that is outputted by the algorithm has only $O(\frac{1}{\epsilon} \cdot (\log N)^{1.5} \cdot \log \frac{1}{\delta})$ additive error (with probability 1).*
- *The algorithm is oblivious and completes in $O(N)$ runtime.*

We remark that the original results in [12, 13] is an $(\epsilon, 0)$-differentially private algorithm such that the outputted prefix sum has at most $O(\frac{1}{\epsilon} \cdot (\log N)^{1.5} \cdot \log \frac{1}{\delta})$ addititive error with probability at least $1 - \delta$, which clearly implies the above theorem (by just outputting the non-private prefix-sum when the error in the output is too large). We choose to state Theorem 3.1 since bounded error is needed for our differentially oblivious algorithms to achieve perfect correctness.

3.1.4 Detailed Algorithm We first describe a tight stable compaction algorithm that stably compacts an input array I given a privacy parameter ϵ and a batch size s.

TightStableCompact(I, ϵ, s):

- Invoke an instance of the differentially private prefix sum algorithm with the privacy budget ϵ to estimate for every $i \in [N]$, the total number of 1-balls in the input stream I up till position i — henceforth we use the notation $\widetilde{Y}_i$ to denote the i-th prefix sum estimated by the differentially private prefix sum algorithm.
- Imagine there is a working buffer initialized to be empty. We now repeat the following until there are no more bins left in the input.
  1. Fetch the next s balls from the input stream into the working buffer.
  2. Obliviously sort the working buffer such that all 1-balls are moved to the front, and all 0-balls moved to the end; we use the ball's index in the input array to break ties for stability.
  3. Suppose that k balls from the input have been operated on so far. If there are fewer than $\widetilde{Y}_k - s$ balls in the output array, pop the head of the working buffer and append to the output array until there are $\widetilde{Y}_k - s$ balls in the output array.
  4. If the working buffer (after popping) is longer than $2s$, truncate from the end such that the working buffer is of size $2s$.
- Finally, at the end, if the output is shorter than N, then obliviously sort the working buffer (using the same relative ordering function as before) and write an appropriate number of balls from the head into the output such that the output buffer is of length N.

Finally, as mentioned, we can construct stable 1-bit-key sorting by running two instances of tight stable compaction and then in $O(N)$ time combining the two output arrays into the final outcome. We state our theorem below but defer the analysis and proofs to the full version [10].

THEOREM 3.2. (STABLE 1-BIT-KEY SORTING) *For any $\epsilon > 0$ and any $0 < \delta < 1$, there exists an (ϵ, δ)-differentially oblivious algorithm such that for any input array with N balls each tagged with a 1-bit key, the algorithm completes in $O(N \log(\frac{1}{\epsilon} \log^{1.5} N \log \frac{1}{\delta}))$ runtime and stably sorts the balls with perfect correctness. As a special case, for $\epsilon = \Theta(1)$, there exists an (ϵ, δ)-differentially oblivious stable 1-bit-key sorting algorithm such that it completes in $O(N \log \log N)$ runtime and has negligible δ.*

Optimality. In light of our lower bound to be presented in the next section (Theorem 4.4), our 1-bit-key stable sorting algorithm is in fact optimal (up to $\log \log$ factors) as long as $\epsilon s \geq 2 \log^2 N$ — note that this includes most parameter ranges one might care about. For the special case of $\epsilon = \Theta(1)$, our upper bound is $\widetilde{O}(N)$ runtime for $\delta = e^{-\operatorname{poly} \log N}$ and $\widetilde{O}(N \log N)$ runtime for $\delta = e^{-N^{0.1}}$ where $\widetilde{O}$ hides a $\log \log$ factor — both cases match our lower bound.

3.2 Sorting More Bits Given an algorithm for stably sorting 1-bit keys, we can easily derive an algorithm for stably sorting k-bit keys simply using the well-known approach of Radix Sort: we sort the input bit by bit starting from the lowest-order bit. Clearly, if the stable 1-bit-key sorting building block satisfies (ϵ, δ)-differentially oblivious, then resulting k-bit-key stable sorting algorithm satisfies $(k\epsilon, k\delta)$-differentially oblivious. This gives rise to the following corollary.

COROLLARY 3.1. (STABLE k-BIT-KEY SORTING) *For any $\epsilon, \delta > 0$, there exists an (ϵ, δ)-differentially oblivious*

algorithm such that for any input array with N balls each tagged with a k-bit key, the algorithm completes in $O(kN \log(\frac{k}{\epsilon} \log^{1.5} N \log \frac{1}{k\delta}))$ runtime and stably sorts the balls with perfect correctness.

As a special case, for $\epsilon = \Theta(1)$, there exists an (ϵ, δ)-differentially oblivious stable k-bit-key sorting algorithm that completes in $O(kN \log\log N)$ runtime and has negligible δ.

We point out that if $k = o(\log N / \log\log N)$, we obtain a stable k-bit-key sorting algorithm that overcomes the $\Omega(N \log N)$ barrier for stable δ-oblivious sort in the balls-and-bins model — recall that Lin, Shi, and Xie [31] show that for even $\delta = O(1)$, any (possibly non-comparison-based) stable 1-bit-key δ-oblivious sorting algorithm in the balls-and-bins model must incur $\Omega(N \log N)$ runtime. We stress that our algorithm is non-comparison-based, since otherwise due to the 0-1 principle, any comparison-based sorting algorithm — even without privacy requirements and even for 1-bit keys — must incur at least $\Omega(N \log N)$ runtime.

4 Limits of Differentially Oblivious Sorting

We showed earlier that for a suitably and negligibly small δ and $\epsilon = \Theta(1)$, by adopting the weaker notion of (ϵ, δ)-differential obliviousness, we can overcome the $\Omega(N \log N)$ barrier for oblivious stable sorting for small keys (in the balls-and-bins model). In this section, we show that if δ must be subexponentially small (including the special case of requiring $\delta = 0$), then (ϵ, δ)-differentially oblivious 1-bit stable sorting would suffer from the same lower bound as the oblivious case. Without loss of generality, we may assume that *the CPU has a single register* and can store a single record (containing a ball and an associated key) and its address — since any $O(1)$ number of registers can be simulated by a trivial ORAM with $O(1)$ blowup.

4.1 Definitions and Preliminaries We begin by presenting some new notions and preliminaries that are necessary for our lower bound.

4.1.1 Plausibility of Access Patterns among Neighboring Inputs In order to derive our lower bounds for differentially oblivious sorting, merging, and data structures, we show that for a differentially oblivious algorithm, with high probability, the access pattern produced for some input I is "plausible" for many inputs that are "close" to I.

Definition 4.1. (r-neighbors) *Two inputs are r-neighboring, if they differ in at most r positions.*

Definition 4.2. (Plausible access pattern) *An access pattern A produced by a mechanism M is plausible for an input I, if $\Pr[\mathbf{Accesses}^M(\lambda, I) = A] > 0$; if $\Pr[\mathbf{Accesses}^M(\lambda, I) = A] = 0$, we say that A is implausible for I.*

Lemma 4.1. *Suppose I_0 is some input for a mechanism M that is (ϵ, δ)-differentially oblivious, and $\mathcal{C}$ is a collection of inputs that are r-neighbors of I_0. Then, the probability that $\mathbf{Accesses}^M(\lambda, I_0)$ is plausible for all inputs in $\mathcal{C}$ is at least $1 - \eta$, where $\eta := |\mathcal{C}| \cdot \frac{e^{\epsilon r} - 1}{e^{\epsilon} - 1} \cdot \delta$.*

4.1.2 Access Pattern Graphs under the Balls-and-Bins Model Recall that we assume a balls-and-bins model and without loss of generality we may assume that the CPU has a single register and can store a single ball and its key.

Access pattern graph. We model consecutive t memory accesses by an access pattern graph defined as follows. Let N index the CPU register together with the memory locations accessed by the CPU in those t accesses. The t memory accesses are represented by $t+1$ layers of nodes, where the layers are indexed from $i = 0$ to t. The nodes and edges of the access pattern graph are defined precisely as follows.

(a) *Nodes.* For each $0 \le i \le t$, layer i consists of nodes of the form (i, u), where $u \in N$ represents either the CPU or a memory location. Intuitively, the node (i, u) represents the opaque ball stored at u after the i-th memory access.

(b) *Edges.* Each edge is directed and points from a node in layer $i-1$ to one in layer i for some $i \ge 1$. For $u \in N$, there is a directed edge from its copy $(i-1, u)$ in layer $i-1$ to (i, u) in layer i. This reflects the observation that if a ball is stored at u before the i-th access, then it is plausible that the same ball is still stored at u after the i-th access.

Suppose the CPU accesses memory location ℓ in the i-th access. Then, we add two directed edges $((i-1, CPU), (i, \ell))$ and $((i-1, \ell), (i, CPU))$. This reflects the balls stored in the CPU and location ℓ can possibly move between those two places.

Compact access pattern graph (compact graph). Observe that in each layer i, any node that corresponds to a location not involved in the i-th access has in-degree and out-degree being 1. Whenever there is such a node x with the in-coming edge (u, x) and the out-going edge (x, v), we remove the node x and add the directed edge (u, v). This is repeated until there is no node with both in-degree and out-degree being 1. We call the resulting graph the *compact access pattern graph*, or simply the *compact graph*. The following lemma relates

the number of memory accesses to the number of edges in the compact graph.

LEMMA 4.2. (NUMBER OF EDGES IN A COMPACT GRAPH) *Suppose N is the set indexing the CPU together with the memory location accessed by the CPU in consecutive t accesses. Then, the compact graph corresponding to these t accesses has $4t + |N| - 2 \leq 5t$ edges.*

4.1.3 Preliminaries on Routing Graph Complexity We consider a routing graph. Let I and O denote a set of n input nodes and $m \geq n$ output nodes respectively. We say that A is an assignment from I to O if A is an injection from nodes in I to nodes O. A routing graph G is a directed graph, and we say that G implements the assignment A if there exist n *vertex-disjoint* paths from I to O respecting the assignment A.

Let $\mathbf{A} := (A_1, A_2, \ldots, A_s)$ denote a set of assignments from I to O. We say $\mathbf{A}$ is non-overlapping if for every input $x \in I$, the assignements map x to distinct outputs, i.e., $A_i(x) \neq A_j(x)$ for every $i \neq j \in [s]$. Pippenger and Valiant proved the following useful result [37].

FACT 4.1. (PIPPENGER AND VALIANT [37]) *Let $\mathbf{A} := (A_1, A_2, \ldots, A_s)$ denote a set of assignments from I to O where $n = |I| \leq |O|$ Let G be a graph that implements every A_i for $i \in [s]$. If $\mathbf{A}$ is non-overlapping, then the number of edges in G must be at least $3n \log_3 s$.*

In our lower bound proofs, we shall make use of Fact 4.1 together with Lemma 4.2 to show that the number of memory location accesses is large in each relevant scenario. A useful set of non-overlapping assignments are shift assuments, defined as follows.

DEFINITION 4.3. (SHIFT ASSIGNMENT) *We say that A is a shift assignment for the input nodes $I = \{x_0, x_1, \ldots, x_{n-1}\}$ and output nodes $O = \{y_0, y_1, \ldots, y_{n-1}\}$ iff there is some s such that for any $i \in \{0, 1, \ldots, n-1\}$, x_i is mapped to y_j where $j = (i + s) \mod n$ — we also refer to s as the shift offset.*

4.2 Lower Bounds for Differentially Oblivious Sorting Warmup and intuition. As a warmup, we consider a simple lower bound proof for the case $\delta = 0$ and for general sorting (where the input can contain arbitrary keys not just 1-bit keys). Suppose there is some ϵ-differentially oblivious balls-and-bins sorting algorithm denoted sort. Now, given a specific input array I, let G be such a compact graph encountered with non-zero probability p. By the requirement of ϵ-differential obliviousness, it must be that for any input array I', the probability of encountering G must be at least $p \cdot e^{-\epsilon N} > 0$. This means G must also be able to explain any other input array I'. In other words, for any input I' there must exist a feasible method for routing the balls contained in the input I' to their correct location in the output locations in G. Recall that in the compact graph G, every node (t, i) can receive a ball from either of its two incoming edges: either from the parent (t', i) for some $t' < t$, from the parent $(t - 1, CPU)$. Let T be the total number of nodes in G, by construction, it holds that the number of edges in $G = \Theta(T)$. Now due to a single counting argument, since the graph must be able to explain all $N!$ possible input permutations, we have $2^T \geq N!$. By taking logarithm on both sides, we conclude that $T \geq \Omega(N \log N)$.

The more interesting question arises for $\delta \neq 0$. We will now prove such a lower bound for $\delta \neq 0$. Instead of directly tackling a general sorting lower bound, we start by considering *stably* sorting balls with 1-bit keys, where stability requires that any two balls with the same key must appear in the output in the same order as in the input. Note that given any general sorting algorithm, we can realize 1-bit-key stable sorting in a blackbox manner: every ball's 1-bit key is appended with its index in the input array to break ties, and then we simply sort this array. Clearly, if the general sorting algorithm attains (ϵ, δ)-differential obliviousness, so does the resulting 1-bit-key stable sorting algorithm. Thus, a lower bound for 1-bit-key stable sorting is stronger than a lower bound for general sorting (parameters being equal).

THEOREM 4.4. (LIMITS OF DIFFERENTIALLY OBLIVIOUS 1-BIT-KEY STABLE SORTING) *Let $0 < s \leq \sqrt{N}$ be an integer. Suppose $\epsilon > 0$ and $0 \leq \delta \leq e^{-(2\epsilon s + \log^2 N)}$. Then, any (randomized) stable 1-bit-key sorting algorithm (in the balls-and-bins model) that is (ϵ, δ)-differentially oblivious must have some input, on which it incurs at least $\Omega(N \log s)$ memory accesses with probability at least $1 - \mathsf{negl}(N)$ for some negligible function $\mathsf{negl}(\cdot)$.*

Proof. We assume that the input is given in N specific memory locations $\mathsf{Input}[0..N-1]$, and the stable sorting algorithm M must write the output in another N specific memory locations $\mathsf{Output}[0..N-1]$.

For each $0 \leq i \leq s$, we define the input scenario I_i as follows, such that in each scenario, there are exactly s elements with key value 0 and $N - s$ elements with key value 1. Specifically, in scenario I_i, the first $s - i$ and the last i elements in $\mathsf{Input}[0..N-1]$ have key value

0, while all other elements have key value 1. It can be checked that any two scenarios are $2s$-neighboring.

Moreover, observe that for $0 \le i \le s$, in scenario I_i, any ball with non-zero key in $\mathsf{Input}[j]$ is supposed to go to $\mathsf{Output}[j+i]$ (where addition $j+i$ is performed modulo N) after the stable sorting algorithm is run.

Observe that a stable sorting algorithm can only guarantee that all the elements with key 0 will appear at the prefix of Output according to their original input order. However, after running the stable sorting algorithm, we can use an extra oblivious sorting network on the first s elements to ensure that in the input scenario I_i, any element with key 0 in $\mathsf{Input}[j]$ originally will end up finally at $\mathsf{Output}[j+i]$. Therefore, the resulting algorithm is still (ϵ, δ)-differentially oblivious.

Therefore, by Lemma 4.1, with probability at least $1-\eta$ (where $\eta := s \cdot \frac{e^{\epsilon \cdot 2s}-1}{e^{\epsilon}-1} \cdot \delta = \neg(N)$), running the algorithm M on input I_0 produces an access pattern A that is plausible for I_i for all $1 \le i \le s$. Let G be the compact graph (defined Section 4.1.2) corresponding to A.

Observe that A is plausible for I_i implies that G contains N vertex-disjoint paths, where for $0 \le j < N$, there is such a path from the node corresponding to the initial memory location $\mathsf{Input}[j]$ to the node corresponding to the final memory location $\mathsf{Output}[j+i]$.

Then, Fact 4.1 implies that G has at least $\Omega(N \log s)$ edges. Hence, Lemma 4.2 implies that the access pattern A makes at least $\Omega(N \log s)$ memory accesses. Since our extra sorting network takes at most $O(s \log s)$ memory accesses, it follows that the original sorting algorithm makes at least $\Omega(N \log s)$ accesses.

Notice that given any general sorting algorithm (not just for 1-bit keys), one can construct 1-bit-key stable sorting easily by using the index as low-order tie-breaking bits. Thus our lower bound for stable 1-bit-key sorting also implies a lower bound for general sorting as stated in the following corollary.

COROLLARY 4.1. *Let $0 < s \le \sqrt{N}$ be an integer. Suppose $\epsilon > 0$ and $0 \le \delta \le e^{-(2\epsilon s + \log^2 N)}$. Then, any (randomized) sorting algorithm that is (ϵ, δ)-differentially oblivious must have some input, on which it incurs at least $\Omega(N \log s)$ memory accesses with probability at least $1 - \mathsf{negl}(N)$ for some negligible function $\mathsf{negl}(\cdot)$.*

Finally, just like our upper bounds, our lower bounds here assume that the algorithm must be perfectly correct. In the full version [10], we show how to generalize the lower bound to work for algorithms that can make mistakes with a small probability.

5 Differentially Oblivious Merging: Upper Bounds

Merging in the balls-and-bins model is the following abstraction: given two input arrays each of which contains at most N balls sorted by their tagged keys, merge them into a single sorted array. Pippenger and Valiant [37] showed that any oblivious merging algorithm in the balls-and-bins model must incur at least $\Omega(N \log N)$ movements of balls.

In this section, we show that when $\epsilon = O(1)$ and δ is negligibly small (but not be subexponentially small), we can accomplish (ϵ, δ)-differentially oblivious merging in $O(N \log \log N)$ time! This is yet another separation between obliviousness and our new notion of differential obliviousness.

Clarifications: definition of neighboring inputs for merging. In merging, both input arrays must be sorted. As a result, to define the notion of neighboring inputs, it does not make sense to take an input array and flip a position to an arbitrarily value — since obviously this would break the sortedness requirement. Instead, we say that two inputs (I_0, I_1) and (I'_0, I'_1) are neighboring iff for each of $b \in \{0,1\}$, the two (multi-)sets $\mathsf{set}(I_b)$ and $\mathsf{set}(I'_b)$ differ in exactly one record. Based on this notion of neighboring, (ϵ, δ)-differentially obliviousness for merging is defined in the same manner as in Section 2.2.

5.1 Intuition

The naïve non-private merging algorithm keeps track of the head pointer of each array, and performs merging in linear time. However, how fast each head pointer advances leaks the relative order of elements in the two input arrays. Oblivious merging hides this information completely but as mentioned, must incur $\Omega(N \log N)$ runtime in the balls-and-bins model. Since our requirement is differential obliviousness, this means that we can reveal some noisy aggregate statistics about the two input arrays. We next highlight our techniques for achieving better runtimes.

Noisy-boundary binning and interior points. Inspired by Bun et al. [8], we divide each sorted input array into $\mathrm{poly}\log\lambda$-sized bins (where λ is the security parameter). To help our merging algorithm decide how fast to advance the head pointer, a differentially private mechanism by Bun et al. [8] is used to return an interior point of each bin, where an interior point is defined to be any value that is (inclusively) between the minimum and the maximum elements of the bin. Technically, the following components are important for our proofs to work.

1. *Random bin loads and localization:* each bin must

contain a random number of real elements padded with dummies to the bin's maximum capacity $Z = \text{poly}\log\lambda$ — this is inspired by Bun et al. [8]. The randomization in bin load allows a "localization" technique in our proofs, since inserting one element into the input array can be obfuscated by local noise and will not significantly affect the distribution of the loads of too many bins.

2. *Secret bin load.* For privacy, it is important that the actual bin loads be kept private from the adversary. This raises a technical challenge: since the adversary can observe the access patterns when the bins are constructed, how can we make sure that the access patterns do not reveal the bins' loads? One naïve approach is to resort to oblivious algorithms — but oblivious sorting in the balls-and-bins model has a well-known $\Omega(N \log N)$ lower bound [31] and thus would be too expensive.

Creating the bins privately. To answer the above question of how to construct the bins securely without disclosing the bins' actual loads, we again rely on a batching and queuing technique similiar in spirit to our tight stable compaction algorithm. At a high level, for every iteration i : 1) we shall read a small, poly-logarithmically sized batch of elements from the input stream into a small, poly-logarithmically sized working buffer; 2) we rely on oblivious algorithms to construct the i-th bin containing the smallest R_i elements in the buffer padded with dummies, where the load R_i has been sampled from an appropriate distribution. These elements will then be removed from the working buffer.

The key to making this algorithm work is to ensure that at any time, the number of elements remaining in the buffer is at most polylogarithmic (in the security parameter). This way, running oblivious algorithms (e.g., oblivious sorting) on this small buffer would incur only $\log\log$ overheads. To this end, we again rely on a differentially private prefix sum mechanism (which must be made oblivious first) to estimate how many real elements will be placed in the first i bins for every choice of i. Suppose that the number of real elements in the first i bins is in the range $[C_i, C_i']$ (except with negligible probability); then when constructing the i-th bin, it suffices to read the input stream upto position C_i'.

It would seem like the above idea still leaks some information about each bin's actual load — but we will prove that this leakage is safe. Concretely, in our Appendices, we will prove a binning composition theorem, showing that with our noisy-boundary binning, it is safe to release any statistic that is differentially private with respect to the binning outcome — the resulting statistic would also be differentially private with respect to the original input.

Putting the above together, we devise an almost linear-time, differentially oblivious procedure for dividing input elements into bins with random bin loads, where each bin is tagged with a differentially private interior point — henceforth we call this list of bins tagged with interior points *thresh-bins*.

Merging lists of thresh-bins. Once we have converted each input array to a list of thresh-bins, the idea is to perform merging by reading bins from the two input arrays, and using each bin's interior point to inform the merging algorithm which head pointer to advance. Since each bin's load is a random variable, it is actually not clear how many elements to emit after reading each bin. Here again, we rely on a differentially private prefix sum mechanism to estimate how many elements to emit, and store all the remaining elements in a poly-logarithmically sized working buffer. In this manner, we can apply oblivious algorithm techniques to the small working buffer incurring only $\log\log$ blowup in performance.

5.2 Preliminaries Oblivious bin placement.

Oblivious bin placement is the following abstraction: given an input array X, and a vector V where $V[i]$ denotes the intended load of bin i, the goal is to place the first $V[1]$ elements of X into bin 1, place the next $V[2]$ elements of X into bin 2, and so on. All output bins are padded with dummies to a maximum capacity Z. Once the input X is fully consumed, all remaining bins will contain solely dummies.

We construct an oblivious algorithm for solving the bin placement problem. Our algorithm invokes building blocks such as oblivious sorting and oblivious propagation constant number of times, and thus it completes in $O(n \log n)$ runtime where $n = \max(|X|, Z \cdot |V|)$. We present the theorem statement for this building block and defer the details to the Appendices.

THEOREM 5.1. (OBLIVIOUS BIN PLACEMENT)
There exists a deterministic, oblivious algorithm that realizes the aforementioned bin placement abstraction and completes in time $O(n \log n)$ where $n = \max(|X|, Z \cdot |V|)$.

Truncated geometric distribution. Let $Z > \mu$ be a positive integer, and $\alpha \geq 1$. The truncated geometric distribution $\mathsf{Geom}^Z(\mu, \alpha)$ has support with the integers in $[0..Z]$ such that its probability mass function at $x \in [0, Z]$ is proportional to $\alpha^{-|\mu - x|}$. We consider the special case $\mu = \frac{Z}{2}$ (where Z is even) and use the shorthand $\mathsf{Geom}^Z(\alpha) := \mathsf{Geom}^Z(\frac{Z}{2}, \alpha)$. In this case, the probability mass function at $x \in [0..Z]$ is

$\frac{\alpha-1}{\alpha+1-2\alpha^{-\frac{Z}{2}}} \cdot \alpha^{-|\frac{Z}{2}-i|}$.

5.3 Subroutine: Differentially Oblivious Interior Point Mechanism Bun et al. [8] propose a differentially private interior point algorithm: given an array I containing sufficient samples, they show how to release an interior point that is between $[\min(I), \max(I)]$ in a differentially private manner. Unfortunately, their algorithm does not offer access pattern privacy if executed in a naïve manner. In the Appendices, we show how to design an oblivious algorithm that efficiently realizes the interior point mechanism — our approach makes use of oblivious algorithm techniques (e.g., oblivious sorting and oblivious aggregation) that were adopted in the design of ORAM and OPRAM schemes [20, 18, 22, 6, 11, 36]. Importantly, since our main algorithm will call this oblivious interior point mechanism on bins containing dummy elements, we also need to make sure that our oblivious algorithm is compatible with the existence of dummy elements and not disclose how many dummy elements there are. We present the following theorem while deferring its detailed proof to the Appendices. In the Appendices, we also discuss how to realize the oblivious interior point mechanism on finite-word-length RAMs without assuming arbitrary-precision real arithmetic.

THEOREM 5.2. (DIFFERENTIALLY PRIVATE INTERIOR POINT) *For any $\epsilon, \delta > 0$, there exists an algorithm such that given any input bin of capacity Z consisting of n real elements, whose real elements have keys from a finite universe $[0..U-1]$ and $n \geq \frac{18500}{\epsilon} \cdot 2^{\log^* U} \cdot \log^* U \cdot \ln \frac{4 \log^* U}{\epsilon\delta}$, the algorithm*

- *completes consuming only $O(Z \log Z)$ time and number of memory accesses.*
- *the algorithm produces an outcome that is (ϵ, δ)-differentially private;*
- *the algorithm has perfect correctness, i.e., the outcome is an interior point of the input bin with probability 1; and*
- *the algorithm's memory access pattern depends only on Z, and in particular, is independent of the number of real elements the bin contains.*

5.4 Subroutine: Creating Thresh-Bins In the ThreshBins subroutine, we aim to place elements in an input array X into bins where each bin contains a random number of real elements (following a truncated geometric distribution), and each bin is padded with dummies to the maximum capacity Z. The ThreshBins will emit exactly B bins. Later when we call ThreshBins we guarantee that B bins will almost surely consume all elements in X. Logically, one may imagine that X is followed by infinitely many ∞ elements such that there are always more elements to draw from the input stream when creating the bins. Note that ∞'s are treated as filler elements with maximum key and not treated as dummies (and this is important for the interior point mechanism to work).

ThreshBins$(\lambda, X, B, \epsilon_0)$:

Assume:

1. $B \leq \text{poly}(\lambda)$ for some fixed polynomial $\text{poly}(\cdot)$.
2. $\epsilon_0 < c$ for some constant c that is independent of λ.
3. The keys of all elements are chosen from a finite universe denoted $[0..U-1]$, where $\log^* U \leq \log\log\lambda$ (note that this is a very weak assumption).
4. Let the bin capacity $Z := \frac{1}{\epsilon_0}\log^8\lambda$, and $s = \frac{1}{\epsilon_0} \cdot \log^3\lambda$

Algorithm:

- Recall that the elements in X are sorted; if the length of the input X is too small, append an appropriate number of elements with key ∞ at the end such that it has length at least $2BZ$.

  This makes sure that the real elements in the input stream do not deplete prematurely in process below.
- For $i = 1$ to B, let $R_i = \mathsf{Geom}^Z(\exp(\epsilon_0))$ be independently sampled truncated geometric random variables. Denote the vector $R := (R_1, R_2, \ldots, R_B)$.
- Call $D := \mathsf{PrefixSum}(\lambda, R, \frac{\epsilon_0}{4}, \delta_0) \in Z_+^B$, which is the $(\frac{\epsilon_0}{4}, \delta_0)$-differentially private subroutine in Theorem 3.1 that privately estimates prefix sums, where δ_0 is set so that the additive error is at most s. We use the convention that $D[0] = 0$.
- Let Buf be a buffer with capacity $Z + s = O(Z)$. Initially, we place the first s elements of X in Buf.
- For $i = 1$ to B:
  - Read the next batch of elements from the input stream X with indices from $D[i-1] + s + 1$ to $D[i] + s$, and add these elements to the buffer Buf. This is done by temporarily increasing the capacity of Buf by appending these elements at the end. Then, oblivious sorting can be used to move any dummy elements to the end, after which we can truncate Buf back to its original capacity.
  - Call $\mathsf{ObliviousBinPlace}(\mathsf{Buf}, (R_i), Z)$ to place the first R_i elements in Buf into the next bin and the bin is padded with dummies to the maximum capacity Z.

– Mark every element in Buf at position R_i or smaller as dummy. (This is done by a linear scan so that the access pattern hides R_i and effectively removes the first R_i elements in Buf in the next oblivious sort.)

- Tag each bin with its estimated prefix sum from vector D. Moreover, we use the $(\frac{\epsilon_0}{4}, \delta)$-differentially oblivious interior point mechanism in Section 5.3 to tag each bin with an interior point, denoted by a vector $P = (P_1, \ldots, P_B)$, where $\delta := \frac{1}{4}\exp(-0.1\log^2 \lambda)$.
- Output the B bins.

5.5 Subroutine: Merging Two Lists of Thresh-Bins We next describe an algorithm to merge two lists of thresh-bins. Recall that the elements in a list of thresh-bins are sorted, where each bin is tagged with an interior point and also an estimate of the prefix sum of the number of real elements up to that bin.

$\mathsf{MergeThreshBins}(\lambda, T_0, T_1, \epsilon_0)$:

Assume:

1. The input is T_0 and T_1, each of which is a list of thresh-bins, where each bin has capacity $Z = \frac{1}{\epsilon_0}\log^8 \lambda$ size. For $b \in \{0,1\}$, let $B_b = |T_b|$ be the number of bins in T_b, and $B := B_0 + B_1$ is the total number of bins. Recall that the bins in T_0 and T_1 are tagged with interior points P_0 and P_1 and estimated prefix sums D_0 and D_1, respectively.
2. The output is an array of sorted elements from T_0 and T_1, where any dummy elements appear at the end of the array. The length of the array is $M := BZ$.

Algorithm:

- Let $s = \frac{1}{\epsilon_0}\log^3 \lambda$.
- Initialize an empty array $\mathsf{Output}[0..M-1]$ of length $M := BZ$.

  Initialize $\mathsf{count} := 0$, the number of elements already delivered to Output.
- Initialize indices $j_0 = j_1 = 0$ and a buffer Buf with capacity $K := 6(Z+s) = O(Z)$. Add elemnts in $T_0[1]$ and $T_1[1]$ to Buf.
- Let $\mathcal{L}$ be the list of sorted bins from T_0 and T_1 according to the tagged interior points. (Observe that we do not need oblivious sort in this step.) We will use this list to decide which bins to add to Buf.
- For $i = 1$ to B:
  – Update the indices j_0, j_1: if the bin $\mathcal{L}[i]$ belongs to T_b, update $j_b \leftarrow j_b + 1$. (This maintains that $\mathcal{L}[i] = T_b[j_b]$.)
  – Add elements in bin $T_b[j_b+1]$ (if exists) to Buf. This is done by appending elements in $T_b[j_b+1]$ at the end of Buf to temporarily increase the size of Buf, and then use oblivious sorting followed by truncation to restore its capacity. (Note that $T_b[j_b+1]$ may not be the next bin in the list $\mathcal{L}$.) Note that the elements in Buf are always sorted.
  – Determine *safe* bins k_0, k_1: For $b \in \{0,1\}$, let k_b be the maximal index k such that the following holds: (i) $T_b[k]$ is inserted in Buf, (ii) there exists some bin $T_{1-b}[u]$ from T_{1-b} that has been inserted into Buf and whose interior point is at least that of $T_b[k+1]$, i.e., $P_{1-b}[u] \geq P_b[k+1]$. (Observe that any element with key smaller than that of an element in a safe bin has already been put into the buffer.) If there is no such index, set $k_b = 0$. Note that the last bin B_b cannot be safe.
  – Remove safe bins from Buf: Set $\mathsf{newcount} := D_0[k_0] + D_1[k_1] - 2s$. Remove the first ($\mathsf{newcount} - \mathsf{count}$) elements from the Buf and copy them into the next available slots in the Output array. Then update $\mathsf{count} \leftarrow \mathsf{newcount}$.
- Output the remaining elements: Let $\mathsf{newcount} = \min\{D_0[B_0] + D_1[B_1] + 2s, BZ\}$. Copy the first ($\mathsf{newcount} - \mathsf{count}$) into the next available slots in the Output array.

5.6 Full Merging Algorithm Finally, the full merging algorithm involves taking the two input arrays, creating thresh-bins out of them using ThreshBins, and then calling Merge to merge the two lists of thresh-bins. We defer concrete parameters of the full scheme and proofs to the Appendices.

$\mathsf{Merge}(\lambda, I_0, I_1, \epsilon)$:

Assume:

1. The input is two sorted arrays I_0 and I_1.
2. We suppose that $\epsilon < c$ for some constant c, $\log^* U \leq \log\log\lambda$, and $|I_0| \leq \mathrm{poly}_0(\lambda)$ and $|I_1| \leq \mathrm{poly}_1(\lambda)$ for some fixed polynomials $\mathrm{poly}_0(\cdot)$ and $\mathrm{poly}_1(\cdot)$.

Algorithm:

1. First, for $b \in \{0,1\}$, let $B_b := \lceil \frac{2|I_b|}{Z}(1 + \frac{2}{\log^2 \lambda}) \rceil$, call $\mathsf{ThreshBins}(\lambda, I_b, B_b, 0.1\epsilon)$ to transform each input array into a list of thresh-bins — let T_0 and T_1 denote the outcomes respectively.

2. Next, call $\mathsf{MergeThreshBins}(\lambda, T_0, T_1, 0.1\epsilon)$ and let T be the sorted output array (truncated to length $|I_0| + |I_1|$).

3. Do a linear scan on T, I_0, I_1 to check if T contains the same number of non-dummy elements as in the input (I_0, I_1). If so, output T. Otherwise (this can happen when the bin load in the thresh-bins are too small so that some elements are dropped), perform a non-private merge to output a correct merged array.

THEOREM 5.3. (DIFFERENTIALLY OBLIVIOUS MERGING) *The* $\mathsf{Merge}(\lambda, I_0, I_1, \epsilon)$ *algorithm is* (ϵ, δ)*-differentially oblivious, where* $\delta = \exp(-\Theta(\log^2 \lambda))$. *Moreover, its running time is* $O((|I_0| + |I_1|)(\log \frac{1}{\epsilon} + \log\log \lambda))$ *and it has perfect correctness.*

We defer the proofs of the above theorem to the Appendices.

5.7 Limits of Differentially Oblivious Merging

In this section, we prove a lower bound regarding the performance of differentially oblivious merging.

THEOREM 5.4. (LIMITS OF (ϵ, δ)-DIFFERENTIALLY OBLIVIOUS MERGING) *Consider the merging problem, in which the input is two sorted lists of elements and the output is the merging of the two input lists into a single sorted list.*

Let $0 < s \leq \sqrt{N}$ *be an integer. Suppose* $\epsilon > 0$ *and* $0 \leq \delta \leq e^{-(\epsilon s + \log^2 N)}$. *Then, any merging algorithm that is* (ϵ, δ)*-differentially oblivious must have some input consisting of two sorted lists each of length* N*, on which it incurs at least* $\Omega(N \log s)$ *memory accesses with probability at least* $1 - \mathsf{negl}(N)$.

Proof. We consider two input lists. The first list $\mathsf{Input}_1[0..N-1]$ is always the same such that $\mathsf{Input}_1[j]$ holds an element with key value $j + 1$.

We consider $s + 1$ scenarios for the second list. For $0 \leq i \leq s$, in scenario I_i, $\mathsf{Input}_2[0..N-1]$ contains i elements with key value 0 and $N - i$ elements with key value $N + 1$. It follows that any two such scenarios are s-neighboring.

By Lemma 4.1, on input scenario I_0, any merging algorithm that is (ϵ, δ)-differentially oblivious produces an access pattern A that is plausible for all I_i's ($1 \leq i \leq s$) with all but probability of $s \cdot \frac{e^{\epsilon s} - 1}{e^{\epsilon} - 1} \cdot \delta = \mathsf{negl}(N)$.

We assume that the merging algorithm writes the merged list into the memory locations $\mathsf{Output}[0..2N-1]$. Hence, for all $0 \leq i \leq s$, in scenario I_i, for all $0 \leq j < N$, the element initially stored at $\mathsf{Input}_1[j]$ will finally appear at $\mathsf{Output}[i + j]$.

Therefore, any access pattern A that is plausible for I_i must correspond to a compact graph G that contains N vertex-disjoint paths, each of which goes from the node representing the initial $\mathsf{Input}_1[j]$ to the node representing the final $\mathsf{Output}[i+j]$, for $0 \leq j < N$.

Hence, Lemma 4.1 implies that if A is plausible for all scenarios I_i's, then the corresponding compact G has $\Omega(N \log s)$ edges, which by Lemma 4.2 implies that the access pattern A must make at least $\Omega(N \log s)$ memory accesses.

6 Differentially Oblivious Range Query Data Structure

6.1 Data Structures A data structure in the RAM model is a possibly randomized stateful algorithm which, upon receiving requests, updates the state in memory and optionally outputs an answer to the request — without loss of generality we may assume that the answer is written down in memory addresses $[0..L-1]$, where L is the length of the answer.

As mentioned, we consider data structures in the balls-and-bins model where every record (e.g., patient or event record) may be considered as an opaque ball tagged with a key. Algorithms are allowed to perform arbitrary computations on the keys but the balls can only be moved around.

We start by considering data structures that support two types of operations, *insertions* and *queries*. Each insertion inserts an additional record into the database and each query comes from some query family $\mathcal{Q}$. We consider two important query families: 1) for our lower bounds, we consider point queries where each query wants to request all records that match a specified key; 2) for our upper bounds, we consider range queries where each query wants to request all records whose keys fall within a specified range $[s, t]$.

Correctness notion under obfuscated lengths. As Kellaris et al. [26] show, leaking the number of records matching each query can, in some settings, cause entire databases to be reconstructed. Our differential obliviousness definitions below will protect such length leakage. As a result, more than the exact number of matching records may be returned with each query. Thus, we require only a relaxed correctness notion: for each query, suppose that L records are returned — we require that all matching records must be found within the L records returned. For example, in a client-server setting, the client can retrieve the answer-set (one by one or altogether), and then prune the non-matching records locally.

Performance metrics: runtime and locality. For our data structure construction, besides the classical *runtime* metric that we have adopted throughout the

paper, we consider an additional *locality* metric which was commonly adopted in recent works on searchable encryption [9, 5] and Oblivious RAM constructions [4]. Real-life storage systems including memory and disks are optimized for programs that exhibit locality in its accesses — in particular, sequential accesses are typically much cheaper than random accesses. We measure a data structure's locality by counting *how many discontiguous memory regions it must access to serve each operation.*

6.2 Defining Differentially Oblivious Data Structures We define two notions of differential obliviousness for data structures, static and adaptive security. Static security assumes that the data structure's operational sequences are chosen statically independent of the answers to previous queries; whereas adaptive security assumes that the data structure's operational sequences are chosen adaptively, possibly dependent on the answers to previous queries. Notice that this implies that both the queries and the database's contents (which are determined by the insertion operations over time) can be chosen adaptively.

As we argue later, adaptive differential obliviousness is strictly stronger than the static notion. We will use the static notion for our lower bounds and the adaptive notion for our upper bounds — this makes both our lower- and upper-bounds stronger.

Due to space limit, the following contents are deferred to the full version [10]:

- Formal definitions of static and adaptive differential obliviousness for data structures.
- Range Query from Thresh-Bins. We use the differentially oblivious algorithmic building blocks thresh-bins introduced in earlier parts of the paper to design an efficient differentially oblivious data structure for range queries.
- Range Query Data Structure Construction. We use a hierarchical data structure that is inspired by hierarchical ORAM constructions [20, 18, 22]. However, we will accomplish rebuilding a level in almost linear time by using the MergeThreshBins procedure described earlier.
- Lower bounds for differentially oblivious data structures.

Acknowledgments

T-H. Hubert Chan was partially supported by the Hong Kong RGC under the grant 17200418.

Elaine Shi is extremely grateful to Dov Gordon for multiple helpful discussions about relaxing the notion of oblivious data accesses over the past several years, including back at Maryland and recently — these discussions partly inspired the present work. She is also grateful to Abhradeep Guha Thakurta for numerous discussions about differential privacy over the past many years including during the preparation of the present paper. We are grateful to Wei-Kai Lin and Tiancheng Xie for numerous inspiring discussions especially regarding the Pippenger-Valiant result [37]. We thank Kobbi Nissim, George Kellaris, and Rafael Pass for helpful discussions and suggestions. We thank Kartik Nayak and Paul Grubbs for helpful feedback that helped improve the paper.

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

Kai-Min Chung was partially supported by the 2016 Academia Sinica Career Development Award under Grant no. 23-17 and Ministry of Science and Technology, Taiwan, under Grant no. MOST 106-2628-E-001-002-MY3.

Bruce Maggs was supported in part by NSF Grant CCF-1535972.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(N \log N)$ sorting network. In *STOC*, 1983.

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, 2013.

[3] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comput. Syst. Sci.*, 57(1):74–93, Aug. 1998.

[4] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi. Oblivious computation with data locality. Cryptology ePrint Archive 2017/772, 2017.

[5] G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In *STOC*, 2016.

[6] E. Boyle, K. Chung, and R. Pass. Oblivious parallel RAM and applications. In *TCC*, 2016.

[7] E. Boyle and M. Naor. Is there an oblivious RAM lower bound? In *ITCS*, 2016.

[8] M. Bun, K. Nissim, U. Stemmer, and S. P. Vadhan. Differentially private release and learning of threshold functions. In *FOCS*, 2015.

[9] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Eurocrypt*, 2014.

[10] T. H. Chan, K. Chung, B. M. Maggs, and E. Shi. Foundations of differentially oblivious algorithms. *IACR Cryptology ePrint Archive*, 2017:1033, 2017.

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

[12] T.-H. H. Chan, E. Shi, and D. Song. Private and continual release of statistics. In *ICALP*, 2010.

[13] T.-H. H. Chan, E. Shi, and D. Song. Private and continual release of statistics. *TISSEC*, 2011.

[14] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)*, 2006.

[15] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *STOC*, 2010.

[16] C. Dwork, G. N. Rothblum, and S. P. Vadhan. Boosting and differential privacy. In *FOCS*, pages 51–60. IEEE Computer Society, 2010.

[17] D. Eppstein, M. T. Goodrich, and R. Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, pages 13–22, 2010.

[18] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[19] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *ACM symposium on Theory of computing (STOC)*, 1987.

[20] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[21] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In *STOC*, 2014.

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

[23] M. T. Goodrich, O. Ohrimenko, and R. Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

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

[25] Y. Han and M. Thorup. Integer sorting in 0(n sqrt (log log n)) expected time and linear space. In *FOCS*, 2002.

[26] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[27] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Accessing data while preserving privacy. *CoRR*, abs/1706.01552, 2017.

[28] M. Keller and P. Scholl. Efficient, oblivious data structures for mpc. In *Asiacrypt*, 2014.

[29] D. G. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. Technical report, 1981.

[30] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. 1998.

[31] W.-K. Lin, E. Shi, and T. Xie. Can we overcome the $n \log n$ barrier for oblivious sort? Manuscript, 2017.

[32] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *S&P*, 2015.

[33] S. Mazloom and S. D. Gordon. Differentially private access patterns in secure computation. Cryptology ePrint Archive, Report 2017/1016, 2017. `https://eprint.iacr.org/2017/1016`.

[34] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 13:10, 2013.

[35] J. C. Mitchell and J. Zimmerman. Data-oblivious data structures. In *STACS*, pages 554–565, 2014.

[36] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.

[37] N. Pippenger and L. G. Valiant. Shifting graphs and their applications. *J. ACM*, 23(3):423–432, July 1976.

[38] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[39] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[40] M. Thorup. Randomized sorting in $o(nloglogn)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Algorithms*, 42(2):205–230, 2002.

[41] S. Vahdan. *The Complexity of Differential Privacy*.

[42] S. Wagh, P. Cuff, and P. Mittal. Root ORAM: A tunable differentially private oblivious RAM. *CoRR*, abs/1601.03378, 2016.

[43] X. S. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*, 2015.

[44] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *CCS*, 2014.

[45] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.