Foundations of Differentially Oblivious Algorithms

T-H. Hubert Chan  Kai-Min Chung  Bruce Maggs  Elaine Shi

August 5, 2020

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 “(\(\epsilon, \delta\))-differential obliviousness”. We separate the notion of (\(\epsilon, \delta\))-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 \(\epsilon\) and \(\delta\), 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 \(\epsilon\) and \(\delta\), the same lower bounds for oblivious algorithms would be preserved for (\(\epsilon, \delta\))-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 [2,36], 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) [19,48] 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 [17,24,34] and Oblivious RAM (ORAM) constructions [18,20]. 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 approach is to apply an Oblivious RAM (ORAM) compiler, an elegant algorithmic technique originally proposed by Goldreich and Ostrovsky [18,20]. Although ORAM constructions have significantly improved over the past few years [41,42,46], 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 [18,20].

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 is the first to show 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 [18,20]. 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 $\epsilon$-differential obliviousness and $(\epsilon, \delta)$-differential obliviousness. Specifically, just like $\epsilon$-DP and $(\epsilon, \delta)$-DP, a non-zero $\delta$ 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 $\delta$) 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 $\delta$ term. Perhaps more surprisingly, most of our logarithmic full obliviousness lower bounds would still apply had we required $\epsilon$-differential obliviousness for arbitrarily large $\epsilon$ (even though intuitively, very little privacy is preserved for large values of $\epsilon$)!

Both our lower bounds and upper bounds require novel techniques. Our lower bounds draw connections to the complexity of shifting graphs [39] 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, 44], but now the observables are access patterns. Informally, we would like to guarantee that an adversary, after having observed access patterns to (encrypted) 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 $(\epsilon, \delta)$-differentially oblivious, iff for any two neighboring input databases $I$ and $I'$, for any set $S$ of access patterns, it holds that

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

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

The term $\delta$ 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 $\delta$ be negligibly small in some security parameter $\lambda$. When $\delta = 0$, we also say that $M$ satisfies $\epsilon$-differential obliviousness.

Comparison with full obliviousness. It is interesting to contrast the notion of differential obliviousness with the classical notion of full obliviousness [18, 20]. An algorithm $M$ (expressed as a RAM program) is said to be (statistically) $\delta$-oblivious iff for any input databases $I$ and $I'$ of equal length, it holds that $\text{Accesses}^M(I) \overset{\delta}{=} \text{Accesses}^M(I')$ where $\overset{\delta}{=}\text{denotes that the two distributions have statistical distance at most } \delta$. 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 $(\epsilon, \delta)$-differential obliviousness is a relaxation of $\delta$-obliviousness; and likewise $\epsilon$-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 $e^\epsilon$ difference in the distribution of the access patterns incurred by neighboring inputs (besides the $\delta$ failure probability); whereas full obliviousness does not permit this $e^\epsilon$ relaxation.

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

1Our 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.
carry over to $\epsilon$-differential obliviousness for several algorithmic abstractions we are concerned with. However, by additionally permitting a non-zero (but negligibly small) failure probability $\delta$, 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) for any two neighboring databases (inserted dynamically)

$^2$ Our notion of adaptive differential obliviousness is akin to the standard adaptive DP notion for dynamic databases $^3$, 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 $\epsilon$ and $\delta$ 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 $^3$) — and a subset of these techniques apply to the balls-and-bins model. A recent manuscript by Lin, Shi, and Xie $^3$ showed that interesting barriers arise if we require full obliviousness for sorting:

**Fact 1.1 (Barriers for oblivious sorting $^3$).** 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

---

$^2$ Roughly, we say two dynamic databases are neighboring if the sequence of insert and query operations differ in at most one position; see Section $^3$ 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 rule out \( \Omega(1) \) oblivious stable sorting algorithms 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 \cite{7}). Further, as Lin, Shi, and Xie showed \cite{33}, 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 \( \epsilon \) and \( \delta \) that give reasonable privacy, one can indeed sort small-length keys in \( o(N \log N) \) time and attain \((\epsilon, \delta)\)-differential obliviousness. As a typical parameter choice, for \( \epsilon = \Theta(1) \) and \( \delta \) 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 we overcome the \( \Omega(N \log N) \) barrier of oblivious sorting.

We state our result more formally and for generalized parameters:

**Theorem 1.2** \((\epsilon, \delta)\)-differentially oblivious stable \( k \)-bit sorting. For any \( \epsilon > 0 \) and any \( 0 < \delta < 1 \), there exists an \((\epsilon, \delta)\)-differentially oblivious \( k \)-bit stable sorting algorithm that completes in \( O(kN(\log^2 \frac{\delta}{\epsilon} + \log \log N + \log \log \frac{1}{\delta})) \) runtime. As a special case, for \( \epsilon = \Theta(1) \), there exists an \((\epsilon, \negl(N))\)-differentially oblivious stable 1-bit sorting algorithm that completes in \( O(N \log \log N) \) runtime for some suitable negligible function \( \negl(\cdot) \), say, \( \negl(N) := \exp(-\log^2 N) \).

Note that the above upper bound statement allows for general choices of \( \epsilon \) and \( \delta \). Interestingly, for the typical parameters \( \epsilon = \Theta(1) \) and \( \delta < 1/N \), our 1-bit stable sorting algorithm is optimal in light of the following lower bound. 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 1.3** (Limits of \((\epsilon, \delta)\)-differentially oblivious stable sorting in the balls-and-bins model). For any \( 0 < s \leq \sqrt{N} \), any \( \epsilon > 0 \), 0 < \( \beta < 1 \) and \( 0 \leq \delta \leq \beta \cdot \frac{s}{\epsilon} \cdot e^{-2s} \), any \((\epsilon, \delta)\)-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 probability at least \( 1 - \beta^4 \).

As a corollary, under the same parameters, any \((\epsilon, \delta)\)-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.

Note that the lower bound allows a tradeoff between \( \epsilon \) and \( \delta \). 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 \( \delta \) is required to be 0, then even when \( \epsilon \) may be arbitrarily large, any \( \epsilon \)-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 \( \epsilon \) — and yet if \( \delta \) must be 0, the same barrier for full obliviousness carries over!

\footnote{All lower bounds in this paper can be extended to handle imperfect correctness as we show in the Appendices (Section 15).}
1.2.2 Merging Two Sorted Lists

Merging is also a classical abstraction and has been studied extensively in the algorithms literature (e.g., [32]). 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 [39] 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 \( b \in \{0, 1\} \), \( \text{set}(I_b) \) and \( \text{set}(I'_b) \) 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 1.4** (Limits of \((\epsilon, \delta)\)-differentially oblivious merging in the balls-and-bins model). For any \( 0 < s \leq \sqrt{N} \), any \( \epsilon > 0 \), \( 0 < \beta < 1 \) and \( 0 \leq \delta \leq \beta \cdot \frac{\epsilon}{s} \cdot e^{-\epsilon s} \), any \((\epsilon, \delta)\)-differentially oblivious merging algorithm in the balls-and-bins model must incur, on some input, at least \( \Omega(N \log s) \) memory accesses with probability at least \( 1 - \beta \).

**Theorem 1.5** \((\epsilon, \delta)\)-differentially oblivious merging). For any \( \epsilon > 0 \) and any \( 0 < \delta < 1 \), there exists an \((\epsilon, \delta)\)-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 \((\epsilon, \text{negl}(N))\)-differentially oblivious merging algorithm that completes in \( O(N \log \log N) \) runtime for some suitable negligible function \( \text{negl}(\cdot) \).

The above theorems are stated for general choices of \( \epsilon \) and \( \delta \), below we point out several notable special cases:

1. First, assuming \( \epsilon = \Theta(1) \), if \( \delta \) must be subexponentially small, then the same lower bound for oblivious merging will be preserved for \((\epsilon, \delta)\)-differentially oblivious merging.

2. Second, for \( \epsilon = \Theta(1) \) and \( \delta \) negligibly small (but not subexponentially small), we can achieve \((\epsilon, \delta)\)-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 \( \epsilon \) is (and how little privacy we get from such a large \( \epsilon \)).

\(^5\)Pippenger and Valiant’s proof [39] 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.
1.2.3 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 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).

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 and ORAM, 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 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.
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 1.6** ($((\epsilon, \delta)$-differentially oblivious data structures). Suppose that $\epsilon = \Theta(1)$ and that $\text{negl}(\cdot)$ is a suitable negligible function. There is an $(\epsilon, \text{negl}(N))$-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 $O(\text{polylog} N + L)$ 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 [30, 37, 47] and the non-private baseline:

1. We asymptotically outperform known oblivious data structures that have logarithmic (multiplicative) overheads [30, 37, 47] (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 $L \geq \text{polylog} N$.

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 1.7** (Limits of $(\epsilon, \delta)$-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 $\epsilon > 0$, $0 < \beta < 1$ and $0 \leq \delta \leq \beta \cdot \frac{s}{r} \cdot e^{-\epsilon s}$. Suppose that $\mathcal{DS}$ is a perfectly correct and $(\epsilon, \delta)$-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 $\mathcal{DS}$ makes for serving this sequence is $\Omega(\log \frac{s}{r})$ with probability at least $1 - \beta$. 

9
Hence, one immediate observation we can draw is that our earlier range query upper bound (Theorem 1.6) is optimal assuming that the number of records matching the query is at least polylogarithmic in size and assuming the typical parameters $\epsilon = \Theta(1)$ and $\delta < 1/N$. 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 subsequent formal sections.

1.3 Closely Related Work

We are inspired by the recent work of Kellaris et al. [29]. 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. [29] 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. [45], 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 [42] 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 [33] 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. [33] 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 [35] 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 [35] had results that achieved constant-factor improvement over the prior work GraphSC [38] that achieved full security. Subsequently, they improved their
result to obtain asymptotical gains (see their latest version online [38]). 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.

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 $\delta$ term of $(\epsilon, \delta)$-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

---

*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.*
paper, unless otherwise noted, for any randomized algorithm we require perfect correctness\footnote{Jumping ahead, given an \((\epsilon, \delta)\)-differentially oblivious algorithm that incurs \(\delta'\) 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.}

- 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 \(\lambda\) 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 \(\epsilon, \delta\) be functions in a security parameter \(\lambda\). We say that the stateless algorithm \(M\) satisfies \((\epsilon, \delta)\)-differential obliviousness, iff for any neighboring inputs \(I\) and \(I'\), for any \(\lambda\), for any set \(S\) of access patterns, it holds that

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

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

In the above, the term \(\delta\) 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 \(\delta\) to be a negligible function in the security parameter \(\lambda\), i.e., every individual can rest assured that as long as \(\lambda\) is sufficiently large, its own privacy is unlikely to be harmed. On the other hand, we would like \(\epsilon\) not to grow w.r.t. \(\lambda\), and thus a desirable choice for \(\epsilon\) 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 \(\delta\)-statistical obliviousness, iff for any inputs \(I\) and \(I'\) of equal length, for any \(\lambda\), it holds that

\[
\text{Accesses}^M(\lambda, I) \overset{\delta(\lambda)}{=} \text{Accesses}^M(\lambda, I')
\]

where \(\delta(\lambda)\) 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 \(\delta\)-statistically oblivious, it must also be \((\epsilon, \delta)\)-differentially oblivious. In other words, \((\epsilon, \delta)\)-differentially obliviousness is a strict relaxation of \(\delta\)-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 \(e^\epsilon\) 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 [1, 21]. 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 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 [33] — 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 $\delta$-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 $\delta$ and for $\epsilon = \Theta(1)$, we can accomplish $(\epsilon, \delta)$-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 $(\epsilon, \delta)$-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 batches 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 [11,12,15]. 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 oblivious 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. [11,12] 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 [11,12]). For any $\epsilon, \delta > 0$, there exists an $(\epsilon, \delta)$-differentially private algorithm, such that given a stream in $\mathbb{Z}_+^N$ (where neighboring streams differ 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 $O\left(\epsilon \cdot (\log N)^{1.5} \cdot \log \frac{1}{\delta}\right)$ additive error (with probability 1).

- The algorithm is oblivious and completes in $O(N)$ runtime.

We remark that the original results in [11, 12] is an $(\epsilon, 0)$-differentially private algorithm such that the outputted prefix sum has at most $O\left(\epsilon \cdot (\log N)^{1.5} \cdot \log \frac{1}{\delta}\right)$ additive 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 $\epsilon$ and a batch size $s$.

**TightStableCompact**: $(I, \epsilon, s)$:

- Invoke an instance of the differentially private prefix sum algorithm with the privacy budget $\epsilon$ 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 $\tilde{Y}_i$ 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 $\tilde{Y}_k - s$ balls in the output array, pop the head of the working buffer and append to the output array until there are $\tilde{Y}_k - s$ 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.

**Theorem 3.2** (Tight stable compaction). For any $\epsilon, \delta > 0$, for any input array $I$ containing $N$ elements, let $s = \Theta\left(\frac{1}{\epsilon} \cdot \log^{1.5} N \cdot \log \frac{1}{\delta}\right)$, then the algorithm **TightStableCompact**(I, $\epsilon$, $s$) satisfies $(\epsilon, \delta)$-differential obliviousness and perfect correctness. Further, the algorithm completes in $O(N \log s)$ runtime. As a special case, for any $\epsilon = \Theta(1)$ and $s = \log^3 N$, the algorithm satisfies $(\epsilon, \delta)$-differential obliviousness with perfect correctness and negligible $\delta$. 
Proof. Notice that the access patterns of the algorithm is uniquely determined by the set of prefix sums computed. Thus it suffices to prove that the set of prefix sums resulting from the prefix sum algorithm satisfies \((\epsilon, \delta)-differential privacy. This follows in a straightforward manner from Theorem 3.1. Perfect correctness of the algorithm is guaranteed since the prefix sum has at most \(s\) additive error, thus perfect correctness also follows from Theorem 3.1. The runtime of the algorithm is dominated by \(O(N/s)\) number of oblivious sortings of the working buffer whose size, by construction, is at most \(O(s)\). Thus the runtime claims follows naturally.

Corollary 3.3 (Stable 1-bit sorting). For any \(\epsilon > 0\) and any \(0 < \delta < 1\), there exists an \((\epsilon, \delta)\)-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 \((\epsilon, \delta)\)-differentially oblivious stable 1-bit sorting algorithm such that it completes in \(O(N \log \log N)\) runtime and has negligible \(\delta\).

Proof. As mentioned, we can construct stable 1-bit sorting by running two instances of tight stable compaction and then in \(O(N)\) time combining the two output arrays into the final outcome. Thus the corollary follows in a straightforward fashion from Theorem 3.2.

Optimality. In light of our lower bound to be presented in the next section (Theorem 4.7), our 1-bit-key stable sorting algorithm is in fact optimal for the typical parameters \(\epsilon = \Theta(1)\) and \(\delta < 1/N\) — 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 \(\delta = e^{-\text{poly} \log N}\) 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 \((\epsilon, \delta)\)-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 3.4 (Stable \(k\)-bit-key sorting). For any \(\epsilon, \delta > 0\), there exists an \((\epsilon, \delta)\)-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 \((\epsilon, \delta)\)-differentially oblivious stable \(k\)-bit-key sorting algorithm that completes in \(O(kN \log \log N)\) runtime and has negligible \(\delta\).

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 \(\delta\)-oblivious sort in the balls-and-bins model — recall that Lin, Shi, and Xie [33] show that for even \(\delta = O(1)\), any (possibly non-comparison-based) stable 1-bit-key \(\delta\)-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 $\delta$ and $\epsilon = \Theta(1)$, by adopting the weaker notion of $(\epsilon, \delta)$-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 $\delta$ must be subexponentially small (including the special case of requiring $\delta = 0$), then $(\epsilon, \delta)$-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 $\Pr[\text{Accesses}_M(\lambda, I) = A] > 0$; if $\Pr[\text{Accesses}_M(\lambda, I) = A] = 0$, we say that $A$ is implausible for $I$.

**Lemma 4.3.** Suppose $I_0$ is some input for a mechanism $M$ that is $(\epsilon, \delta)$-differentially oblivious, and $C$ is a collection of inputs that are $r$-neighbors of $I_0$. Then, the probability that $\text{Accesses}_M(\lambda, I_0)$ is plausible for all inputs in $C$ is at least $1 - \eta$, where $\eta := |C| \cdot e^{\frac{\epsilon^2 r}{2} - 1} \cdot \delta$.

**Proof.** The proof is deferred to the Appendices (Section C.1).

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 $0 \leq i \leq t$, 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 $i \geq 1$. 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 \( \ell \) in the \( i \)-th access. Then, we add two directed edges \( ((i - 1, \text{CPU}), (i, \ell)) \) and \( ((i - 1, \ell), (i, \text{CPU})) \). This reflects the balls stored in the CPU and location \( \ell \) 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 4.4** (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.

**Proof.** The proof is deferred to the Appendices (Section C.1).

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 \( A := (A_1, A_2, \ldots, A_s) \) denote a set of assignments from \( I \) to \( O \). We say \( A \) is non-overlapping if for every input \( x \in I \), the assignments 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 [39].

**Fact 4.5** (Pippenger and Valiant [39]). Let \( A := (A_1, A_2, \ldots, A_s) \) 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 \( A \) 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 4.5 together with Lemma 4.4 to show that the number of memory location accesses is large in each relevant scenario. A useful set of non-overlapping assignments are shift assignments, defined as follows.

**Definition 4.6** (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 \( \epsilon \)-differentially oblivious balls-and-bins sorting algorithm denoted \( \text{sort} \). Now, given a specific input array \( I \), let \( G \) be such a compact graph encountered with non-zero probability \( p \). By the requirement of \( \epsilon \)-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 \((\epsilon, \delta)\)-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 4.7** (Limits of differentially oblivious 1-bit-key stable sorting). Let \( 0 < s \leq \sqrt{N} \) be an integer. Suppose \( \epsilon > 0 \), \( 0 < \beta < 1 \) and \( 0 \leq \delta \leq \beta \cdot \frac{\epsilon^2}{s} \cdot e^{-2s} \). Then, any (randomized) stable 1-bit-key sorting algorithm (in the balls-and-bins model) that is \((\epsilon, \delta)\)-differentially oblivious must have some input, on which it incurs at least \( \Omega(N \log s) \) memory accesses with probability at least \( 1 - \beta \).

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

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 \( \text{Input}[0..N-1] \) 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 \( 0 \leq i \leq s \), in scenario \( I_i \), any ball with non-zero key in \( \text{Input}[j] \) is supposed to go to \( \text{Output}[j+i] \) (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 \( \text{Output} \) 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 \( \text{Input}[j] \) originally will end up finally at \( \text{Output}[j+i] \). Therefore, the resulting algorithm is still \((\epsilon, \delta)\)-differentially oblivious.

Therefore, by Lemma 4.3, with probability at least \( 1 - \eta \) (where \( \eta := s \cdot \frac{\epsilon^2}{s} \cdot e^{-2s} \cdot \delta \leq \beta \)), running the algorithm \( M \) on input \( I_0 \) produces an access pattern \( A \) that is plausible for \( I_i \) for all \( 1 \leq i \leq s \). 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 \( 0 \leq j < N \), there is such a path from the node corresponding to the initial memory location \( \text{Input}[j] \) to the node corresponding to the final memory location \( \text{Output}[j+i] \).

Then, Fact 4.5 implies that \( G \) has at least \( \Omega(N \log s) \) edges. Hence, Lemma 4.4 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 4.8.** Let \(0 < s \leq \sqrt{N}\) be an integer. Suppose \(\epsilon > 0\), \(0 < \beta < 1\) and \(0 \leq \delta \leq \beta \cdot \frac{s}{s} \cdot e^{-2\epsilon s}\). Then, any (randomized) sorting algorithm that is \((\epsilon, \delta)\)-differentially oblivious must have some input, on which it incurs at least \(\Omega(N \log s)\) memory accesses with probability at least \(1 - \beta\).

Finally, just like our upper bounds, our lower bounds here assume that the algorithm must be perfectly correct. In the Appendices (Section B), 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 [39] 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 \(\delta\) is negligibly small (but not be subexponentially small), we can accomplish \((\epsilon, \delta)\)-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 \(\text{set}(I_b)\) and \(\text{set}(I'_b)\) differ in exactly one record. Based on this notion of neighboring, \((\epsilon, \delta)\)-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 \(\text{polylog}\lambda\)-sized bins (where \(\lambda\) 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 \(Z = \text{polylog}\lambda\) — 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 [33] 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 similar 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 up to 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 \( \text{Geom}^Z(\mu, \alpha) \) 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 \( \text{Geom}^Z(\alpha) := \text{Geom}^Z\left(\frac{Z}{2}, \alpha\right) \). In this case, the probability mass function at \( x \in [0..Z] \) is
\[
\frac{\alpha^{-1}}{\alpha^{1-2\alpha^{-\frac{1}{2}}}} \cdot \alpha^{-\frac{Z}{2} - i}.
\]

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 [6, 10, 18, 20, 23, 38]. 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 \( n \geq \frac{18500}{\epsilon} \cdot 2^{\log^* U} \cdot \log^* U \cdot \ln \frac{4 \log^* U}{\delta} \), the algorithm

- completes consuming only \( O(Z \log Z) \) time and number of memory accesses.
- the algorithm produces an outcome that is \((\epsilon, \delta)\)-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 $\infty$ elements such that there are always more elements to draw from the input stream when creating the bins. Note that $\infty$’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 $\lambda$.
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 $\infty$ 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 $R_i = \text{Geom}^Z(\exp(\epsilon_0))$ be independently sampled truncated geometric random variables. Denote the vector $R := (R_1, R_2, \ldots, R_B)$.
- Call $D := \text{PrefixSum}(\lambda, R, \frac{s}{2}, \delta_0) \in \mathbb{Z}_+^B$, which is the $(\frac{s}{2}, \delta_0)$-differentially private subroutine in Theorem 3.1 that privately estimates prefix sums, where $\delta_0$ 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 ObliviousBinPlace(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.

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

**Assume:**
1. The input is $T_0$ and $T_1$, each of which is a list of thresh-bins, where each bin has capacity $Z = \frac{1}{\epsilon_0} \log^3 \lambda$ 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 $\text{Output}[0..M - 1]$ of length $M := BZ$. Initialize $\text{count} := 0$, the number of elements already delivered to $\text{Output}$.
- Initialize indices $j_0 = j_1 = 0$ and a buffer $\text{Buf}$ with capacity $K := 6(Z + s) = O(Z)$. Add elements in $T_0[1]$ and $T_1[1]$ to $\text{Buf}$.
- 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 $\text{Buf}$.
- 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 $\text{Buf}$. This is done by appending elements in $T_b[j_b + 1]$ at the end of $\text{Buf}$ to temporarily increase the size of $\text{Buf}$, 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 $\text{Buf}$ 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 $\text{Buf}$, (ii) there exists some bin $T_{1-b}[u]$ from $T_{1-b}$ that has been inserted into $\text{Buf}$ 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 newcount := $D_0[k_0] + D_1[k_1] - 2s$. Remove the first (newcount−count) elements from the Buf and copy them into the next available slots in the Output array. Then update count ← newcount.

• Output the remaining elements: Let newcount = $\min\{D_0[B_0] + D_1[B_1] + 2s, BZ\}$. Copy the first (newcount − 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 $\text{ThreshBins}$, and then calling $\text{Merge}$ to merge the two lists of thresh-bins. We defer concrete parameters of the full scheme and proofs to the Appendices.

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

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 $|I_0| \leq \text{poly}_0(\lambda)$ and $|I_1| \leq \text{poly}_1(\lambda)$ for some fixed polynomials $\text{poly}_0(\cdot)$ and $\text{poly}_1(\cdot)$.

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 $\text{ThreshBins}(\lambda, I_b, B_b, 0.1\epsilon)$ to transform each input array into a list of thresh-bins — let $T_0$ and $T_1$ denote the outcomes respectively.
2. Next, call $\text{MergeThreshBins}(\lambda, T_0, T_1, 0.1\epsilon)$ 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 $\text{Merge}(\lambda, I_0, I_1, \epsilon)$ algorithm is $(\epsilon, \delta)$-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 $(\epsilon, \delta)$-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 $\epsilon > 0$, $0 < \beta < 1$ and $0 \leq \delta \leq \beta \cdot \frac{\epsilon}{2} \cdot e^{-c_8}$. Then, any merging algorithm that is $(\epsilon, \delta)$-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 $1 - \beta$. 
Proof. We consider two input lists. The first list \( \text{Input}_1[0..N−1] \) is always the same such that \( \text{Input}_1[j] \) 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 \( I_i \), \( \text{Input}_2[0..N−1] \) 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 4.3, on input scenario \( I_0 \), any merging algorithm that is \((\epsilon,\delta)\)-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 \( s \cdot e^{s−1} \cdot \delta \leq \beta \).

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

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 \( \text{Input}_1[j] \) to the node representing the final \( \text{Output}[i+j] \), for \( 0 \leq j < N \).

Hence, Lemma 4.5 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 4.4 implies that the access pattern \( A \) must make at least \( \Omega(N \log s) \) memory accesses.  

\[ \square \]

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 \( Q \). 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. \[28\] 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 [5, 9] 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.

6.2.1 Static Differential Obliviousness for Data Structures

We now define differential obliviousness for data structures. Our privacy notion captures the following intuition: for any two neighboring databases that differ only in one record (where the database is determined by the insertion operations over time), the access patterns incurred for insertions or queries must be close in distribution. Such a notion protects the privacy of individual records in the database (or of individual events), but does not protect the privacy of the queries. Thus our notion is suitable for a scenario where the data is of a sensitive nature (e.g., hospital records) and the queries are non-sensitive (e.g., queries by a clinical researcher). In fact we will later show that if one must additionally protect the privacy of the queries, then it would be inevitable to incur $\Omega(N)$ blowup in cost on at least some operational sequences. This observation also partly motivates our definition, which requires meaningful and non-trivial privacy guarantees, and importantly, does not rule out efficient solutions.

We say that two operational sequences $\text{ops}_0$ and $\text{ops}_1$ (consisting of insertions and queries) are query-consistent neighboring, if the two sequences differ in exactly position $i$, and moreover both $\text{ops}_0[i]$ and $\text{ops}_1[i]$ must be insertion operations.

**Definition 6.1** (Static differential obliviousness for data structures). Let $\epsilon(\cdot)$ and $\delta(\cdot)$ be functions of a security parameter $\lambda$. We say that a data structure scheme $\mathcal{D}$ preserves static $(\epsilon, \delta)$-differential obliviousness, if for any two query-consistent neighboring operational sequences $\text{ops}_0$ and $\text{ops}_1$, for any $\lambda$, for any set $S$ of access patterns,

$$\Pr[\text{Accesses}_{\mathcal{DS}}(\lambda, \text{ops}_0) \in S] \leq e^{\epsilon(\lambda)} \cdot \Pr[\text{Accesses}_{\mathcal{DS}}(\lambda, \text{ops}_1) \in S] + \delta(\lambda)$$

where the random variable $\text{Accesses}_{\mathcal{DS}}(\lambda, \text{ops})$ denotes the access patterns incurred by the data structure upon receiving the security parameter $\lambda$ and operational sequence $\text{ops}$.

Discussions on alternative notions. It is interesting to consider a stronger notion where the queries must be protected too. We consider one natural strengthening where we want to protect the queries as well as insertions, but the fact whether each operation is an insertion or query is considered non-sensitive. To formalize such a notion, one may simply redefine the notion of
“neighboring” in the above definition, such that any two operational sequences that are type-consistent (i.e., they agree in the type of every operation) and differ in exactly one position are considered neighboring — and this differing position can either be query or insertion. It would not be too difficult to show that such a strong notion would rule out efficient solutions: for example, consider a sequence of operations such that some keys match $\Omega(N)$ records and others match only one record. In this case, to hide each single query, it becomes inevitable that each query must access $\Omega(N)$ elements even when the query is requesting the key with only one occurrence.

6.2.2 Adaptive Differential Obliviousness for Data Structures

We will prove our lower bounds using the above, static notion of differential obliviousness. However, our data structure upper bounds in fact satisfies a stronger, adaptive and composable notion of security as we formally specify below. Here we allow the adversary to adaptively choose the database (i.e., insertions) as well as the queries.

**Definition 6.2** (Adaptive differential obliviousness for data structures). We say that a data structure $\mathcal{DS}$ satisfies adaptive $(\epsilon, \delta)$-differential obliviousness iff for any (possibly unbounded) stateful algorithm $\mathcal{A}$ that is query-consistent neighbor-respecting (to be defined below), for any $N$, $\mathcal{A}$’s view in the following two experiments $\text{Expt}_A^0(\lambda, N)$ and $\text{Expt}_A^1(\lambda, N)$ satisfy the following equation:

$$\Pr[\text{Expt}_A^0(\lambda, N) = 1] \leq e^{\epsilon(\lambda)} \cdot \Pr[\text{Expt}_A^1(\lambda, N) = 1] + \delta(\lambda)$$

$	ext{Expt}_A^b(\lambda, N)$:

1. $\text{addresses}_0 = \perp$
2. For $t = 1, 2, \ldots, N$:
   - $(\text{op}_t^0, \text{op}_t^1) \leftarrow \mathcal{A}(N, \text{addresses}_{t-1})$, $\text{addresses}_t \leftarrow \mathcal{DS}(\lambda, \text{op}_t^b)$
   - $\text{op}_t^b \leftarrow \mathcal{A}$, and output $b'$

In the above, $\text{addresses}_t$ denotes the ordered sequence of physical memory locations accessed for the $t$-th operation $\text{op}_t$ (including whether each access is read or write).

**Neighbor-respecting.** We say that $\mathcal{A}$ is query-consistent neighbor-respecting w.r.t. $\mathcal{DS}$ iff for every $\lambda$ and every $N$, for either $b \in \{0, 1\}$, with probability 1 in the above experiment $\text{Expt}_A^b(\lambda, N)$, $\mathcal{A}$ outputs $\text{op}_t^0 = \text{op}_t^1$ for all but one time step $t \in [N]$; and moreover for this differing time step $t$, $\text{op}_t^0$ and $\text{op}_t^1$ must both be insertion operations.

6.3 Warmup: Range Query from Thresh-Bins

We show that using the differentially oblivious algorithmic building blocks introduced in earlier parts of the paper, we can design an efficient differentially oblivious data structure for range queries. We first explain how the thresh-bins structure introduced for our merging algorithm can also be leveraged for range queries. Recall that a thresh-bins structure contains a list of bins in which all the real elements are sorted in increasing order, and each bin is tagged with an interior point. Given a list of thresh-bins, one can answer a range query simply by returning all bins whose interior point fall in the queried range, as well as the two bins immediately before and after (if they exist).
Range queries Query\((T, [s,t])\). Let \(T := \{\text{Bin}_i\}_{i \in [R]}\) be a list of thresh-bins where Bin\(_i\)'s interior point is \(M_i\). To query a range \([s,t]\), we can proceed in the following steps:

1. Find a smallest set of consecutive bins \(i, i+1, i+2, \ldots, j\) such that \(M_i \leq s \leq t \leq M_j\) — for example, this can be accomplished through binary search. To handle boundary conditions, we may simply assume that there is an imaginary bin before the first bin with the interior point \(-\infty\) and there is an imaginary bin at the end with the interior point \(\infty\).

2. Now, read all bins Bin\(_i\), Bin\(_i+1\), \ldots Bin\(_j\) and output the concatenation of these bins.

6.4 Range Query Data Structure Construction

When records are inserted over time one by one, we may maintain a hierarchy of thresh-bins, where level \(i\) of the hierarchy is a list of thresh-bins containing in total \(2^i \cdot Z\) elements. Interestingly, our use of a hierarchical data structure is in fact inspired by hierarchical ORAM constructions \([18,20,23]\) — however, in hierarchical ORAMs \([18, 20, 23]\), rebuilding a level of capacity \(n\) in the hierarchical structure requires \(O(n \log n)\) time, but we will accomplish such rebuilding in almost linear time by using the \text{MergeThreshBins} procedure described earlier.

We now describe our data structure construction supporting insertions and range queries. The algorithm is parametrized by a privacy parameter \(\epsilon\).

In-memory data structure. Let \(N\) denote the total number of insertions so far. The in-memory data structure consists of the following:

- A recent buffer denoted Buf of capacity \(Z\) to store the most recently inserted items, where \(Z := \frac{1}{\epsilon} \log 8 \lambda\).

- A total of \(\log N\) search structures henceforth denoted \(T_0, T_1, \ldots, T_L\) for \(L := \lceil \log N \rceil\) where \(T_i\) contains \(2^i \cdot Z\) real records and \(N\) denotes the total number of insertions over all time.

Algorithm for insertion. To insert some record, enter it into Buf and if Buf now contains \(Z\) elements, we use \(\tilde{T}_0 := \text{ThreshBins}(\lambda, \text{Buf}, 4, \epsilon)\) to put the elements of Buf into 4 bins, and empty Buf. Now repeat the following starting at \(i = 0\):

- If \(T_i\) is empty, let \(T_i := \tilde{T}_i\) and return (i.e., terminate the procedure).

- Else call \(Y := \text{MergeThreshBins}(\lambda, T_i, \tilde{T}_i, \epsilon)\); and let \(\tilde{T}_{i+1} = \text{ThreshBins}(\lambda, Y, 4 \cdot 2^{i+1}, \epsilon)\), let \(i \leftarrow i + 1\) and repeat if \(i \leq L\).

- For each call of \text{ThreshBins}, check if \text{ThreshBins} produces correct answers by counting the non-dummy elements using a linear scan (to make sure no non-dummy elements are dropped due to too small bin loads \(R\)). If \text{ThreshBins} produce incorrect answers, compute a non-private answer with bin load size \(R_i = Z/2\) for all \(i\).

Algorithm for range query. To query for some range \([s,t]\), let \(T_0, T_1, \ldots, T_L\) be the search structures in memory. For \(i \in [0, 1, \ldots, L]\), call \(\text{Query}(T_i, [s,t])\). Now, concatenate all these outcomes as well as Buf, and copy the concatenated result to a designated location in memory. To further speed up the query, we can maintain the interior points of all active levels in the hierarchical data structure in a single binary search tree.
Theorem 6.3 (Differential obliviousness). Let \( N \) be the total number of insertion operations over time, let \( \epsilon = \Theta(1) \), and suppose that the universe of key satisfies \( \log^* U \leq \log\log\lambda \). Then, there exists a negligible function \( \delta(\cdot) \) such that the above scheme satisfies adaptive \((4\epsilon\log N, \delta)\)-differential obliviousness. Furthermore, the above scheme achieves perfect correctness.

Proof. We first prove perfect correctness. In Lemma A.7 in the appendices, we show that ThreshBins 0-obliviously realize \( F_{\text{threshbins}} \), which implies that ThreshBins always enter input elements in the bins according to its bin load \( R \) correctly. Also in the proof of Lemma A.7 we show that MergeThreshBins always produce correct merged results. Therefore, the only error is when the bin load \( R \) is too small to store all input elements, which happens with probability at most \( O(\exp(-\Theta(\log^2 \lambda))) \) by Lemma C.4. When this happens, the scheme detects the error and outputs a correct non-private answer. Therefore, the scheme achieves perfect correctness at the cost of increasing the privacy error by \( O(\exp(-\Theta(\log^2 \lambda))) \).

Now, differential obliviousness follows in a straightforward manner by adaptive \( \log N \)-fold composition of differential privacy [16], by observing that every element is involved in only \( \log N \) instances of ThreshBins and MergeThreshBins (and also account for the above \( O(\exp(-\Theta(\log^2 \lambda))) \) error). For adaptive security, notice that the adaptive composition theorem works for adaptively generated database entries as well as adaptive queries [16]. \( \square \)

Theorem 6.4 (Performance). Let \( N = \poly(\lambda) \) be the total number of insertion operations over time where \( \poly(\cdot) \) is some fixed polynomial. The above range query data structure achieves the following performance:

- Each insertion operation consumes amortized \( O(\log N \log \log N) \) runtime;
- Each range query whose result set contains \( L \) records consumes \( O(Z \log N + L) \) runtime (and number of accesses) and accesses only \( O(\log N) \) discontiguous regions in memory no matter how large \( L \) is, i.e., the locality is independent of the number of matching records \( L \).

Proof. The insertion cost is dominated by the cost for merging the search structures. In our construction \( T_i \) and \( \widehat{T}_i \) contain \( Z \cdot 2^i \) real elements and every \( 2^i/Z \) operations, we must merge \( T_i \) and \( \widehat{T}_i \) once incurring \( 2Z^2 \log \log \lambda \) time. It is easy to see that the total amortized cost is \( O(\log N \log \log \lambda) \) — note that \( \log N = O(\lambda) \) assuming \( N = \poly(\lambda) \). The runtime and locality claims for each range query follow in a straightforward manner by observing that we can build a single, standard binary search tree data structure (called the index tree) to store all the interior points of all currently active search structures, where leaves are stored from small to large in a consecutive memory region. During insertion, a level containing \( n = 2^i Z \) elements has only \( O(2^i) \) interior points, and thus inserting or deleting all of them from the index tree takes \( o(n) \) time. For query, it takes at most \( O(\log N + L/Z) \) accesses into the index tree to identify all the bins that match the query; and the number of discontiguous regions accessed when searching this index tree is upper bounded by \( O(\log N) \). \( \square \)

We stress that even absent privacy requirements, one of the best known approaches to build a range query data structure is through a binary search tree where each insertion costs \( O(\log N) \) and each query matching \( L \) records costs \( O(\log N + L) \) and requires accessing \( O(\log N) \) discontiguous memory regions. In comparison, our solution achieves differential obliviousness almost for free in many cases: each insertion incurs only a \( O(\log \log N) \) blowup, and each query incurs no asymptotical blowup if there are at least polylogarithmically many matching records and the locality loss is also \( O(1) \).
6.5 Applications in the Designated-Client and Public-Client Settings

In a designated client setting, the data owner who performs insertions is simultaneously the querier. In this case, all records can be encrypted by the data owner’s private key. In a public client setting, the data owner performs insertions of records, whereas queries are performed by other third parties. In this case, the data owner can encrypt the data records using Attribute-Based Encryption (ABE) \([25, 40]\), and then it can issue policy-binding decryption keys to third parties to permit them to query and decrypt records that satisfy the policy predicates. In either case, we stress that our scheme can support queries non-interactively. In particular, the differentially private interior points can be released in the clear to the server, and the server can simply find the matching bins on behalf of the client and return all relevant bins to the client in a single round-trip.

We stress that if the notion of obliviousness were required (say, we would like that any two operational sequences that are query-consistent and length-consistent be indistinguishable in access patterns), then we are not aware of any existing solution that simultaneously achieves statistical security, non-interactivity, and non-trivial efficiency, even for the designated-client setting. One interesting point of comparison is ORAMs \([42, 46]\) and oblivious data structures \([47]\) which can achieve statistical security, but 1) they work only for the designated-client setting but not the public-client setting; 2) in general they incur logarithmically many rounds and \(O(L \log^2 N)\) cost per query (absent large block-size assumptions); and 3) except for the recent work of Asharov et al. \([4]\) which incurs polylogarithmic locality blowup regardless of \(L\), all other known solutions would suffer from (super-)linear in \(L\) locality blowup.

6.6 Lower Bounds for Differentially Oblivious Data Structures

For lower bounds, we first focus on point queries — a special case of the range queries considered in our upper bounds.

Non-private baseline. To put our results in perspective and clearly illustrate the cost of privacy, we first point out that absent any privacy requirements, we can build a data structure that support point queries (in the balls-and-bins model) such that except with negligible probability, each insertion completes in \(O(1)\) time; each point query completes in \(O(L)\) time and accessing only \(O(1)\) discontiguous memory regions where \(L\) is the number of matching records \([22]\).

Limits of differential oblivious data structures. We now prove lower bounds showing that assuming \(\epsilon = O(1)\), if one desires sub-exponentially small \(\delta\), then any \((\epsilon, \delta)\)-differentially oblivious data structure must on some sequences of length \(N\), incur at least \(\Omega(N \log N)\) ball movements. We prove lower bounds for the case of distinct keys and repeated keys separately: in the former case, each key has multiplicity 1 and upon query only 1 record is returned; in the latter, each key has more general multiplicity.

**Theorem 6.5** (Limits of \((\epsilon, \delta)\)-differentially oblivious data structures: distinct keys). Suppose that \(N = \text{poly}(\lambda)\) for some fixed polynomial \(\text{poly}(\cdot)\) and \(0 < s \leq \sqrt{N}\) are integers. Let \(\epsilon > 0\), \(0 < \beta < 1\) and \(0 < \delta \leq \beta \cdot \frac{s}{N} \cdot e^{-\epsilon s}\). Suppose that \(\mathbb{D}S\) is a perfectly correct and \((\epsilon, \delta)\)-differentially oblivious data structure supporting point queries.

Then, there exists an operational sequence with \(N\) insertion and \(N\) query operations interleaved together, where the \(N\) keys inserted are distinct and are from the domain \(\{0, 1, \ldots, N\}\) such that the total number of accesses \(\mathbb{D}S\) makes for serving this sequence is \(\Omega(N \log s)\) with probability at least \(1 - \beta\).
Proof. Define $T := \lfloor \frac{N}{r} \rfloor$. For $1 \leq i \leq T$, define the sub-domain $X_i := \{(i-1)s + j : 0 \leq j < s\}$ of keys. Each of the operational sequences we consider in the lower bound can be partitioned into $T$ epochs. For $1 \leq i \leq T$, the $i$-th epoch consists of the following operations:

1. $s$ insertion operations: the $s$ keys in $X_i$ are inserted one by one. The order in which the keys in $X_i$ are inserted is private. In this lower bound, it suffices to consider $s$ cyclic shifts of the keys in $X_i$.

2. $s$ query operations: this is done in the (publicly-known) increasing order of keys in $X_i$.

Observe that the keys involved between different epochs are disjoint. It suffices to prove that the number of memory accesses made in each epoch is at least $\Omega(s \log s)$ with probability at least $1 - \frac{\beta}{T}$; by the union bound, this immediately implies the result.

Fix some epoch $i$, and consider the $s$ different cyclic shift orders of $X_i$ in which the keys are inserted. For $0 \leq j < s$, let $I_j$ be the input scenario where ordering of the keys in $X_i$ is shifted with offset $j$.

Observe that if we only change the insertion operations in epoch $i$ and keep all operations in other epochs unchanged, we have input scenarios that are $s$-neighbors. Therefore, by Lemma 1.3 with probability at least $1 - \eta$ (where $\eta := s \cdot \epsilon s^{-1} \cdot \delta \leq \beta \cdot \frac{s}{N}$), the input scenario $I_0$ in epoch $i$ produces an access pattern $A$ that is plausible for $I_j$ for all $1 \leq j < s$. Let $G$ be the compact graph (defined Section 4.1.2) corresponding to $A$.

Since we know that in every input scenario $I_j$, each key in $X_i$ is inserted exactly once, we can assume that the $s$ insertions in epoch $i$ correspond to some memory locations $\text{Input}[0..s-1]$. Even though the result of each query can contain dummy elements, because we know that exactly one of the returned elements must be a real element, in this case, by a final linear scan on the returned elements, we can assume that the $s$ queries correspond to some memory locations $\text{Output}[0..s-1]$, where $\text{Output}[k]$ is supposed to return the element with key $(i-1)s + k$.

Moreover, observe that for $0 \leq j < s$, in scenario $I_j$, the element inserted at $\text{Input}[k]$ is supposed to be returned at $\text{Output}[k+j]$ (where addition $j+i$ is performed modulo $s$) during query.

Observe that an access pattern $A$ is plausible for $I_j$ implies that $G$ contains $s$ vertex-disjoint paths, where for $0 \leq k < s$, there is such a path from the node corresponding to the initial memory location $\text{Input}[k]$ to the node corresponding to the final memory location $\text{Output}[k+j]$.

Then, Fact 4.5 implies that if $G$ is the compact graph of an access pattern $A$ that is plausible for all $I_j$’s, then $G$ has at least $\Omega(s \log s)$ edges. Hence, Lemma 4.4 implies that the access pattern $A$ makes at least $\Omega(s \log s)$ memory accesses. This completes the lower bound proof for the number of memory accesses in one epoch, which, as mentioned above, implies the required result.

The following theorem is a generalization of the earlier Theorem 6.5 where each key is allowed to have multiplicity.

**Theorem 6.6 (Limits of ($\epsilon, \delta$)-differentially oblivious data structures: repeated keys).** 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 $\epsilon > 0$, $0 < \beta < 1$ and $0 \leq \delta \leq \beta \cdot \frac{s}{N} e^{-c s}$. Suppose that $\mathcal{DS}$ is a perfectly correct and ($\epsilon, \delta$)-differentially oblivious data structure supporting point queries.

Then, there exists an operational sequence with $N$ insertion and $\frac{N}{r}$ query operations interleaved together, where each of $\frac{N}{r}$ distinct keys from the domain $\{0, 1, \ldots, \frac{N}{r} - 1\}$ is inserted $r$ times, such that the total number of accesses $\mathcal{DS}$ makes for serving this sequence is $\Omega(N \log \frac{s}{r})$ with probability at least $1 - \beta$.  

32
Proof. The proof structure follows that of Theorem 6.5 in which there are \( T := \lfloor \frac{N}{s} \rfloor \) epochs. For \( 1 \leq i \leq T \), the \( i \)-th epoch is defined as follows:

1. \( s \) insertion operations: the \( s \) keys are from the sub-domain \( X_i := \{ \frac{s}{r}, (i-1) + j : 0 \leq j < \frac{s}{r} \} \), where each distinct key is inserted \( r \) times in a batch. The order in which the distinct keys in \( X_i \) are batch-inserted is private. In this lower bound, we consider \( \frac{s}{r} \) different cyclic shifts of the \( \frac{s}{r} \) batches.

2. \( \frac{s}{r} \) query operations: this is done in the (publicly-known) increasing order of keys in \( X_i \), where each query should return \( r \) repeated keys.

As in Theorem 6.5 since the keys involved between different epochs are disjoint, it suffices to prove that the number of memory accesses made in each epoch is at least \( \Omega(s \log \frac{s}{r}) \) with probability at least \( 1 - \frac{\beta}{T} \); this immediately implies the result by the union bound over all epochs.

Fix some epoch \( i \) and observe that if we only change the insertion operations in epoch \( i \) and keep all operations in other epochs unchanged, we have input scenarios that are \( s \)-neighbors.

We consider the \( \frac{s}{r} \) different cyclic shift orders of \( X_i \) in which the keys are inserted. For \( 0 \leq j < \frac{s}{r} \), let \( I_j \) be the input scenario where ordering of the keys in \( X_i \) is shifted with offset \( j \). Therefore, by Lemma 4.3 with probability at least \( 1 - \eta \) (where \( \eta := s \cdot \frac{\delta \cdot \frac{s}{r} + 1}{s - 1} \cdot \delta \leq \beta \cdot \frac{s}{r} \)) the input scenario \( I_0 \) in epoch \( i \) produces an access pattern \( A \) that is plausible for \( I_j \) for all \( 1 \leq j < \frac{s}{r} \). Let \( G \) be the compact graph (defined Section 4.1.2) corresponding to \( A \).

Since we know that in every input scenario \( I_j \), each of the \( \frac{s}{r} \) keys in \( X_i \) is inserted exactly \( r \) times, we can assume that the \( s \) insertions in epoch \( i \) correspond to some memory locations \( \text{Input}[0..s-1] \).

Moreover, each of the \( \frac{s}{r} \) queries returns exactly \( r \) records with the same key, maybe together with some dummy elements. Hence, for \( 0 \leq k < \frac{s}{r} \), we can assume that the result of the query for the key \( (i-1)\frac{s}{r} + k \) is returned in some array \( \text{Output}_k \), whose length is at least \( r \) (and can contain dummy elements).

Moreover, observe that for \( 0 \leq j < \frac{s}{r} \), in scenario \( I_j \), the element inserted at \( \text{Input}[\ell] \) is supposed to be returned in the array \( \text{Output}_k \), where \( k = \lfloor \frac{s}{r} \rfloor + j \mod \frac{s}{r} \). The important point is that the element in \( \text{Input}[\ell] \) will be returned at different locations in different scenarios \( I_j \)'s.

Observe that an access pattern \( A \) is plausible for \( I_j \) implies that \( G \) contains \( s \) vertex-disjoint paths, where for \( 0 \leq \ell < s \), there is such a path from the node corresponding to the initial memory location \( \text{Input}[\ell] \) to the node corresponding to some final memory location inside the array \( \text{Output}_k \), where \( k = \lfloor \frac{s}{r} \rfloor + j \mod \frac{s}{r} \).

Then, Fact 4.5 implies that if \( G \) is the compact graph of an access pattern \( A \) that is plausible for all \( I_j \)'s, then \( G \) has at least \( \Omega(s \log \frac{s}{r}) \) edges. Hence, Lemma 4.4 implies that the access pattern \( A \) makes at least \( \Omega(s \log \frac{s}{r}) \) memory accesses. Hence, it follows that with all but \( \frac{\beta}{T} \) probability, epoch \( i \) takes \( \Omega(s \log \frac{s}{r}) \) memory accesses, as required.

This completes the lower bound proof for the number of memory accesses in one epoch, which, as mentioned above, implies the required result. \( \square \)

Acknowledgments

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 [39]. 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.

References


Y. Han and M. Thorup. Integer sorting in O(n sqrt (log log n)) expected time and linear space. In FOCS, 2002.


Appendices

A Additional Details and Analysis of the Merging Algorithm

This section is devoted to 1) proving Theorem 5.3. We start by introducing the notion of oblivious realization of an ideal functionality with (differentially private) leakage; and 2) supplying additional details of our differentially oblivious merging algorithm, particularly, details of the oblivious interior point mechanism.

A.1 Oblivious Realization of Ideal Functionalities with Differentially Private Leakage

Definition A.1. Given a (possibly randomized) functionality $F$, we say that some (possibly randomized) algorithm $\text{Alg} \ δ$-obliviously realizes $F$ with leakage $L$, if there exists a simulator $\text{Sim}$ (that produces simulated access pattern) such that for any $\lambda$, for any input $I$, define the following executions:

- Ideal execution: choose all random bits $\rho$ needed by $F$, and let $O_{\text{ideal}} \leftarrow F(\lambda, I, \rho)$, let $L_{\text{ideal}} \leftarrow L(\lambda, I, \rho)$. Note that the leakage function $L$ also obtains the same randomness as $F$, and may use additional internal randomness.
- Real execution: let $(O_{\text{real}}, L_{\text{real}}, \text{addresses}) \leftarrow \text{Alg}(\lambda, I)$.

Then, it must hold that the following distributions are $\delta(\lambda)$-statistically close, i.e., their statistical distance is at most $\delta(\lambda)$:

$$(O_{\text{ideal}}, L_{\text{ideal}}, \text{Sim}(\lambda, L_{\text{ideal}})) \equiv (O_{\text{real}}, L_{\text{real}}, \text{addresses}).$$

Definition A.2. We say that the leakage function $L$ is $(\epsilon, \delta)$-differentially private (with respect to the input) iff for every $\lambda$, for every neighboring inputs $I$ and $I'$ and every set $S$, it holds that

$$\Pr_{\rho, \mathcal{L}}[L(\lambda, I, \rho) \in S] \leq e^\epsilon \Pr_{\rho, \mathcal{L}}[L(\lambda, I', \rho) \in S] + \delta$$

In the above, the notation $\Pr_{\rho, \mathcal{L}}$ means that the randomness comes from the random choice of $\rho$ as well as the internal coins of $\mathcal{L}$.

Definition A.3. Consider some special leakage function $\mathcal{L}$ that is fully determined by the output of $F$, i.e., $\mathcal{L}(\lambda, I, \rho) := \mathcal{L}(T)$ where $T := F(\lambda, I, \rho)$. We say that $\mathcal{L}$ is $(\epsilon, \delta)$-differentially private with respect to the output of $F$ (or $(\epsilon, \delta)$-differentially private with respect to $T$), iff for every $\lambda$, for every neighboring $T$ and $T'$ and every set $S$, it holds that $\Pr[\mathcal{L}(T) \in S] \leq e^\epsilon \Pr[\mathcal{L}(T') \in S] + \delta$, where the randomness in the probability comes from the random coins of $\mathcal{L}$.

The following fact is immediate from the definition.

Fact A.4. If some algorithm $\text{Alg}$ obliviously realizes some functionality $F$ with leakage $\mathcal{L}$, where $\mathcal{L}$ is $(\epsilon, \delta)$-differentially private with respect to the input, then $\text{Alg}$ satisfies $(\epsilon, \delta)$-differential obliviousness.
A.2 Ideal $\mathcal{F}_{\text{threshbins}}$ Functionality

We describe a logical thresh-bin functionality $\mathcal{F}_{\text{threshbins}}$ that the ThreshBins subroutine obliviously realizes, and prove a lemma that formalize the main property that ThreshBins achieves.

Given an input sorted array $X$ containing real elements (which can take a special key value $\infty$), a target bin number $B$, and a parameter $\epsilon_0$, the ideal thresh-bin functionality $\mathcal{F}_{\text{threshbins}}$ outputs an ordered list of $B$ bins where each bin contains a random number of real elements padded with dummies to the bin’s capacity $Z = \frac{1}{\epsilon_0} \log^8 \lambda$; all real elements occur in sorted order. Moreover, each bin is tagged with an interior point. Furthermore, each bin is also tagged with an estimate of the cumulative sum, i.e., the number of real elements in the prefix up to and including this bin.

If the input $X$ contains too many real elements, only a prefix of them may appear in the output bins; if the input $X$ contains too few elements, the functionality automatically appends elements with key $\infty$ at the end such that there are enough elements to draw from the input. More concretely, the functionality $\mathcal{F}_{\text{threshbins}}$ is specified below:

$$\mathcal{F}_{\text{threshbins}}(\lambda, X, B, \epsilon_0):$$

**Assume:** The same setting as ThreshBins.

**Functionality:**

- For $i = 1$ to $B$:
  - Sample $R_i \sim \text{Geom}^Z(\exp(\epsilon_0))$.
  - Draw the next $R_i$ elements (denoted $S_i$) from $X$.
  - Place these $R_i$ elements in order in a new bin and append with an appropriate number of dummies to reach the bin’s capacity $Z$.

Let $T$ denote the list of $B$ bins in order, and $R = (R_1, \ldots, R_B)$ be the bin load vector.

- Call $D := \text{PrefixSum}(\lambda, C, \frac{\epsilon_0}{4}) \in \mathbb{Z}^B_+$, which is the $\frac{\epsilon_0}{4}$-differentially private subroutine in Theorem 3.1 that privately estimates prefix sums, each of which has additive error at most $s$ with all but $\exp(-\Theta(\log^2 \lambda))$ probability. We 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 thresh-bins $T$ (which is tagged with the interior points $P$ and the estimated prefix sums $D$).

The following lemma states that for $\mathcal{F}_{\text{threshbins}}$, if a leakage function $\mathcal{L}$ is differentially private with respect to the output $T$, then $\mathcal{L}$ is also differentially private with respect to the input $X$. Here, two thresh-bins $T^0$ and $T^1$ are neighboring if they have the same number of bins, which, except for at most one pair of corresponding bins (from $T^0$ and $T^1$), contain exactly the same elements; for the pair of bins that may differ, their symmetric difference contains only one element.

**Lemma A.5.** Consider the ideal thresh-bins functionality $\mathcal{F}_{\text{threshbins}}$ and a leakage function $\mathcal{L}(\lambda, T, \epsilon_0)$. If the input satisfies $B \geq \lceil \frac{2|X|}{Z} \cdot (1 + \frac{2}{\log^2 \lambda}) \rceil$ and the leakage function $\mathcal{L}$ is $\epsilon, \delta$-differentially private with respect to the output $T$, then $\mathcal{L}$ is $(2\epsilon_0 + 4\epsilon, \delta_{\text{bad}} + 4\delta)$-differentially private with respect to $X$, where $\delta_{\text{bad}} \leq O(\exp(-\Theta(\log^2 \lambda)))$.
To prove Lemma A.5, we start with some notations. Given an input array $X$ and a bin load vector $R = (R_1, \ldots, R_B) \in [Z]^B$, we let $T(X, R)$ denote the resulting thresh-bins. We say two thresh-bins $T, T'$ are $k$-neighboring if there exists $T_1 = T, T_2, \ldots, T_k, T_{k+1} = T'$ such that $T_i, T_{i+1}$ are neighboring. We partition the domain $[Z]^B$ of the bin load vectors into good $\cup$ bad, where $\text{good} = \{ R : \sum_{i=1}^{B-1} R_i \geq |X| \land \forall i : 0 < R_i < Z \}$ and $\text{bad} = [Z]^B\setminus \text{good}$. Let $\delta_{\text{bad}}$ be the probability that $R$ is in bad when $R \leftarrow (\text{Geom}^Z(\exp(\epsilon_0)))^B$. By Lemma C.4 $\delta_{\text{bad}} \leq \exp(- \log^2 \lambda)$. We need the following technical lemma about the ideal thresh-bins functionality.

Lemma A.6. Consider two neighboring input arrays $X^0, X^1$ and parameter $B$ such that $B \geq \lceil 2|X_0| / Z \rceil \cdot (1 + \frac{2}{\log^2 \lambda})$. There exists an injective function $f : \text{good} \to [Z]^B$ such that the following holds. For every $R^0 \in \text{good}$, let $R^1 = f(R^0)$, $T^0 = T(X^0, R^0)$, and $T^1 = T(X^1, R^1)$. We have (i) $\Pr[R^0] \leq e^{2\epsilon_0} \Pr[R^1]$ where the probability is drawn from $(\text{Geom}^Z(\exp(\epsilon_0)))^B$, and (ii) $T^0$ and $T^1$ are 4-neighboring.

Proof. Recall that $X^0, X^1$ are neighboring means they have equal length and differ by one element. Thus, we can view $X^1$ as obtained by removing some $x^0$ from $X^0$ and then inserting some $x^1$ to it. Let $X^0 \setminus \{x^0\}$. Let $i$ denote the location of $x^0$ in $X^0$, and $i'$ denote the location of $x^1$ in $X^0$. We define $f$ in two corresponding steps.

- We first define $f^0$. On input $R^0$, let $\ell$ denote the bin in $T(X^0, R^0)$ that contains $x^0$. We define $f^0(R^0) = R^0$ where $R^0$ is identical to $R^0$ except that with the $(\ell + 1)$-st coordinate is decreased by 1, i.e., $R_{\ell+1}^0 = R_{\ell+1}^0 - 1$ and $R_i^0 = R_i^0$ for all $i \neq \ell + 1$.

- We then define $f^1$, which takes input $R^0$. Let $\ell'$ denote the bin in $T(X^0, R^0)$ that $x^1$ should be inserted in; if it is possible to insert $x^1$ into one of two neighboring bins, take $\ell'$ to be the larger index of the two.

We define $f^1(R^0) = R^1$ where $R^1$ is identical to $R^0$ except that with the $(\ell' + 1)$-st coordinate is increased by 1, i.e., $R_{\ell'+1}^1 = R_{\ell'+1}^0 + 1$ and $R_i^1 = R_i^0$ for all $i \neq \ell' + 1$.

We define $f = f^1 \circ f^0$. Note that by the definition of the good set, $\ell, \ell' < B$ so $f$ is well-defined.

We now verify the properties of $f$. For the injective property, let’s first argue that $f^0$ is injective by showing that it is invertible. The key observation is that give $X^0, X^1$ and the output $R^0$, the bin $\ell$ that $x_0$ belongs to is uniquely defined. Thus, we can compute $(f^0)^{-1}(R^0)$ by increasing the $\ell + 1$-th coordinate by 1. The same argument shows that $f^1$ is injective, and hence $f$ is injective.

The property that $\Pr[R^0] \leq e^{2\epsilon_0} \Pr[R^1]$ follows by the definition of truncated geometric and the fact that $R^0$ and $R^1$ only differ in two coordinates by 1. For property (ii), observe that $T(X^0, R^0)$ and $T(X^0, R^0)$ can only differ in the $\ell$-th and $\ell + 1$-st bins by at most one element for each bin, which means that $T(X^0, R^0)$ and $T(X^0, R^0)$ are 2-neighboring. Similarly, $T(X^0, R^0)$ and $T(X^0, R^0)$ are 2-neighboring by the same observation. Hence, $T^0$ and $T^1$ are 4-neighboring.

With the above lemma, we are ready to prove Lemma A.5.

Proof of Lemma A.5. Consider two neighboring input arrays $X^0, X^1$ and parameter $B$ such that $B \geq (4|X^0| / Z) + 1$. For $b \in \{0, 1\}$, let $T^b \leftarrow \mathcal{F}_\text{threshbins}(\lambda, X^b, B, \epsilon_0)$, $L^b = \leftarrow L(\lambda, T^b, \epsilon_0)$, and $R^b$ be the bin load vector used in $\mathcal{F}_\text{threshbins}$. Let $S$ be an arbitrary subset in the support of the leakage. We need to show that

$$
\Pr[L^0 \in S] \leq e^{2\epsilon_0 + 4\epsilon} \Pr[L^1 \in S] + \delta_{\text{bad}} + 4\delta
$$
that the invariant holds for \( i \) in the help of the following invariant: at the beginning of iteration each \( R \) first properties are from Lemma A.6. Also the fourth inequality uses the property that \( \Pr[X_1, f(R^0)] \). Thus, it remains to show that \( \Pr[L^0] \leq e^{2\epsilon_0} \cdot \Pr[f(R^1)] \) for \( R^0 \in \text{good} \). Both properties are from Lemma A.6.

\[
\Pr[L^0 \in S] \leq \left( \sum_{R^0 \in \text{good}} \Pr[R^0] \Pr[L^0 \in S|X^0, R^0] \right) + \delta_{\text{bad}}
\]

\[
\leq \left( \sum_{R^0 \in \text{good}} \Pr[R^0] \left( e^{4\epsilon} \cdot \Pr[L^1 \in S|X^1, f(R^0)] + 4\delta \right) \right) + \delta_{\text{bad}}
\]

\[
\leq \left( \sum_{R^0 \in \text{good}} \Pr[R^0] \left( e^{4\epsilon} \cdot \Pr[L^1 \in S|X^1, f(R^0)] \right) + 4\delta + \delta_{\text{bad}}
\]

\[
\leq \left( e^{2\epsilon_0 + 4\epsilon} \cdot \sum_{R^0 \in \text{good}} \Pr[f(R^0)] \cdot \Pr[L^1 \in S|X^1, f(R^0)] \right) + 4\delta + \delta_{\text{bad}}
\]

\[
\leq e^{2\epsilon_0 + 4\epsilon} \cdot \Pr[L^1 \in S] + 4\delta + \delta_{\text{bad}}
\]

In the above calculation, we make \( X^b \) explicit in the conditioning even though it is not random. The key step is the second inequality, where we use the property that \( T^0 = T(X^0, R^0) \) and \( T^1 = T(X^1, f(R^0)) \) are 4-neighboring, and \( L(\lambda, T^0, \epsilon_0) \) is \((\epsilon, \delta)\)-differentially private with respect to \( T \). Also the fourth inequality uses the property that \( \Pr[R^0] \leq e^{2\epsilon_0} \cdot \Pr[f(R^1)] \) for \( R^0 \in \text{good} \). Both properties are from Lemma A.6.

**A.3 ThreshBins Obliviously Realize \( F_{\text{threshbins}} \)**

Here we analyze the ThreshBins subroutine and show that it obliviously realize \( F_{\text{threshbins}} \) with differentially private leakages. Specifically, the leakage is the interior points \( P \) and the estimated prefix sums \( D \) associated with the output thresh-bin \( T \). Namely, the leakage function \( L_{\text{threshbins}}(\lambda, X, B, \epsilon_0) \) simply output \( L = (P, D) \).

**Lemma A.7.** The algorithm ThreshBins 0-obliviously realize \( F_{\text{threshbins}} \) with leakage function \( L_{\text{threshbins}} \). Moreover, its running time is \( O(BZ(\log \frac{1}{\epsilon_0} + \log \log \lambda)) \).

**Proof.** We first observe that by construction, the access pattern of ThreshBins is determined by the leakage \((P, D)\). Thus, given the leakage, the access pattern can be readily simulated. Now, note that the output of \( F_{\text{threshbins}} \) is determined by the input \( X \), the bin load vector \( R \), the estimated prefix sums \( D \) and the interior points \( P \), and that these values are computed in an identical way in ThreshBins. Thus, it remains to show that ThreshBins computes exactly the same function as \( F_{\text{threshbins}} \) correctly for every \((X, R, D, P)\).

By definition, \( F_{\text{threshbins}} \) simply puts the first \( \sum_{i=1}^{B} R_i \) elements of \( X \) in \( B \) bins in order with bin load specified by the vector \( R \). On the other hand, at each iteration \( i \), ThreshBins places the first \( R_i \) elements in \( \text{Buf} \) in the \( i \)-th bin. We show that ThreshBins places the correct elements with the help of the following invariant: at the beginning of iteration each \( i \), the non-dummy elements in \( \text{Buf} \) consists of the \((\sum_{j=1}^{i-1} R_j + 1)\)-th to \((D[i-1] + s)\)-th elements in \( X \).

Clearly, the invariant holds for \( i = 1 \) (with the convention that \((\sum_{j=1}^{i-1} R_j) = 0\)). Assume that the invariant holds for \( i \), we observe that after the first step in the iteration, \( \text{Buf} \) consists
of the $((\sum_{j=1}^{i-1} R_j) + 1)$-th to $(D[i] + s)$-th elements in $X$ in sorted order. Since the output of \texttt{PrefixSum} has at most $s$ additive error, we have $(D[i] + s) \geq \sum_{j=1}^{i} R_j$. Also, there are at most $(D[i] + s) - (\sum_{j=1}^{i} R_j) \leq Z + s$ non-dummy elements in $\text{Buf}$, so no elements are lost after truncation. Hence, \texttt{ObliviousBinPlace}($\text{Buf}$, $(R_i, Z)$) can place the $((\sum_{j=1}^{i-1} R_j) + 1)$-th to $(\sum_{j=1}^{i} R_j)$-th elements in $X$ in the $i$-th bin as $F_{\text{threshbins}}$. Then after the first $R_i$ elements in $\text{Buf}$ are marked as dummy, the non-dummy elements in $\text{Buf}$ consists of the $((\sum_{j=1}^{i} R_j) + 1)$-th to $(D[i] + s)$-th elements in $X$, so the invariant holds for $i + 1$.

Observe that the running time of the algorithm is also dominated by the $B$ iterations, each of which takes time $O(Z \log Z) = O(Z(\log \frac{1}{\epsilon_0} + \log \log \lambda))$ due to oblivious sorting, which implies the desired running time.

Noting that the leakage $L_{\text{threshbins}}$ is the outputs of differentially private mechanisms with input determined by the thresh-bins $T$ (since $T$ implicitly determines the bin load $R$), $L_{\text{threshbins}}$ is differentially private with respect to $T$. By Lemma A.5, $L_{\text{threshbins}}$ is differentially private with respect to $X$. We state this in the following lemma.

\textbf{Lemma A.8.} The leakage $L_{\text{threshbins}}$ is $(O(\epsilon_0), \delta)$-differentially private with respect to the input $X$ for $\delta = O(\exp(-\Theta(\log^2 \lambda)))$

\textbf{A.4 Proof of Theorem 5.3}

We are ready to prove Theorem 5.3. We will show that \texttt{Merge} obliviously realize an ideal merge functionality $F_{\text{merge}}$ defined below with differentially private leakage $L_{\text{merge}}$, which implies that $\texttt{Merge}$ is differentially oblivious by Fact A.4

$F_{\text{merge}}(\lambda, I_0, I_1, \epsilon)$:

\textbf{Assume:} The same setting as $\texttt{Merge}$.

\textbf{Functionality:}

- Output a sorted array $T$ that merges elements from $I_0$ and $I_1$, where the dummy elements appear at the end of the array.

The leakage function $L_{\text{merge}}$ is defined to be the concatenation of the leakage $L_{\text{threshbins}}$ on $I_0$ and $I_1$. Namely, $L_{\text{merge}}(\lambda, I_0, I_1, \epsilon) = (L_{\text{threshbins}}(\lambda, I_0, B_0, 0.1\epsilon), L_{\text{threshbins}}(\lambda, I_1, B_1, 0.1\epsilon))$ Clearly, the leakage is differentially private with respect to the input $(I_0, I_1)$.

\textbf{Lemma A.9.} The algorithm $\texttt{Merge}$ $\delta$-obliviously realize $F_{\text{merge}}$ with leakage function $L_{\text{merge}}$ and $\delta = O(\exp(-\Theta(\log^2 \lambda)))$. Moreover, $\texttt{Merge}$ has perfect correctness and its running time is $O(BZ(\log \frac{1}{\epsilon_0} + \log \log \lambda))$.

\textbf{Proof.} We observe that by construction, the access pattern of $\texttt{Merge}$ is determined by the leakage $(P_0, D_0, P_1, D_1)$, unless the check in Step 3 fails. Thus, given the leakage, the access pattern can be readily simulated if the check in Step 3 does not fail. We will show that that check fails with exponentially small probability later and focus on the case when the check does not fail.

Let us consider a hybrid functionality $F'_{\text{merge}}$ that on input $(\lambda, I_0, I_1, \epsilon)$, instead of merging $I_0$ and $I_1$ directly, $F'_{\text{merge}}$ first calls $F_{\text{threshbins}}(\lambda, I_b, B_b, 0.1\epsilon)$ to obtain $T_b$ for $b \in \{0, 1\}$, and then outputs $T$ that merges elements from $T_0$ and $T_1$. Note that the output of $F_{\text{merge}}$ and $F'_{\text{merge}}$ are the same, except for the case that the bin load $R_b$ is not enough to accommodate $I_b$ for some
$b \in \{0, 1\}$, which happens with probability at most $O(\exp(-\Theta(\log^2 \lambda)))$ by Lemma C.4. Thus, up to an $O(\exp(-\Theta(\log^2 \lambda)))$ statistical error, we can switch to consider the hybrid functionality $F'_{\text{merge}}$.

Now, note that $F'_{\text{merge}}$ and $\text{Merge}$ call $F_{\text{threshbins}}(\lambda, I_b, B_b, 0.1\epsilon)$ and $\text{ThreshBins}(\lambda, I_b, B_b, 0.1\epsilon)$ for $b \in \{0, 1\}$, respectively. Since $\text{ThreshBins}$ obliviously realized $F_{\text{threshbins}}$ (with no error), we know that the output thresh-bins $T_b$ (which are tagged with $P_b, D_b$) of $\text{ThreshBins}$ and $F_{\text{threshbins}}$ are identical. From here, the difference between $F'_{\text{merge}}$ and $\text{Merge}$ is that $F'_{\text{merge}}$ directly merges $T_0$ and $T_1$, whereas $\text{Merge}$ uses $\text{MergeThreshBins}$. We now argue that in fact, for any thresh-bins $T_0, T_1$ (with tagged $P_b, D_b$), $F'_{\text{merge}}$ and $\text{MergeThreshBins}$ produce exactly the same answers.

Observe that $\text{MergeThreshBins}$ merges $T_0$ and $T_1$ by inserting the bins into a buffer $\text{Buf}$ in a certain order, and along the way outputting certain numbers of smallest elements in $\text{Buf}$. To argue that $\text{MergeThreshBins}$ computes the same merged result as $F'_{\text{merge}}$, it suffices to show that (i) the buffer $\text{Buf}$ never overflows (i.e., we never truncate non-dummy elements), and (ii) the elements outputted from $\text{Buf}$ are indeed the smallest elements among the remaining elements, since the two conditions imply that $\text{MergeThreshBins}$ correctly output smallest elements in $T_0$ and $T_1$ step by step.

For (i), we claim that at any iteration $i$, both $T_0[j_b - 1]$ and $T_1[j_1 - 1]$ are safe bins (i.e., $k_b \geq j_b - 1$), and hence the number of bins that are inserted into $\text{Buf}$ but not safe is at most 4. Note that by construction, both bins $T_0[j_b + 1]$ and $T_1[j_1 + 1]$ are inserted in $\text{Buf}$ and that $P_b[j_b] \leq P_{1-b}[j_{1-b} + 1]$ for $b \in \{0, 1\}$. This implies that $P_{1-b}[j_{1-b} + 1] \geq P_b[(j_b - 1) + 1]$; namely, $k_b \geq j_b - 1$ for $b \in \{0, 1\}$. Now, note that at each iteration, the first $T_b[j_b + 1]$ bins are inserted in $\text{Buf}$ and the first $D_0[k_0] + D_1[k_1] - 2s$ elements are outputted. Since the output of $\text{PrefixSum}$ has at most $s$ additive error and the bins contains at most $Z$ elements, the number of non-dummy elements in $\text{Buf}$ is at most $5Z + 2s$ at any point of each iteration. Thus, the buffer $\text{Buf}$ never overflows.

For (ii), we argue that as remarked in the construction, when a bin is marked safe, any bins with elements smaller that the elements in the safe bin are already inserted, or equivalently, all bins that are not inserted contains only elements larger than the elements in the safe bins. Thus, the elements outputted from $\text{Buf}$ are indeed the smallest elements among the remaining elements. To see this, consider a safe bin $T_b[k]$. Clearly, any un-inserted bin $T_b[v]$ from $T_b$ has elements greater than elements in $T_b[k]$. For bins in $T_{1-b}$, by definition, there exists some inserted bin $T_{1-b}[u]$ with $P_{1-b}[u] \geq P_b[k + 1]$. Hence, for any un-inserted bin $T_{1-b}[v]$ in $T_{1-b}$, the elements in $T_{1-b}[v]$ has key value $\geq P_{1-b}[v] \geq P_b[k + 1]$, and hence greater than elements in $T_b[k]$.

Now, we argued that the first two steps of $\text{Merge}$ produces identical answers to $F'_{\text{merge}}$ with probability 1, and that $F'_{\text{merge}}$ and $F_{\text{merge}}$ output identical answers, except when the bin load $R_b$ is not enough to accommodate $I_b$ for some $b \in \{0, 1\}$, which happens with probability at most $O(\exp(-\Theta(\log^2 \lambda)))$ by Lemma C.4. Furthermore, note that when this happens, Step 3 of $\text{Merge}$ will detect the error and output a correct merged output by a non-private merge algorithm. Therefore, we can conclude that $\text{Merge}$ $\delta$-obliviously realize $F_{\text{merge}}$ with $\delta = O(\exp(-\Theta(\log^2 \lambda)))$. Furthermore, $\text{Merge}$ outputs correct merged output with probability 1.

Finally, the running time is dominated by the operations on $\text{Buf}$. Since each iteration takes $O(K \log K) = O(Z(\log \frac{1}{\epsilon_0} + \log \log \lambda))$ time due to oblivious sorting, it follows that the total time is $O(BZ(\log \frac{1}{\epsilon_0} + \log \log \lambda))$, as required.

A.5 Obliviously Realizing the Interior Point Mechanism

We start by recalling the recursive differentially private algorithm $\text{InteriorPoint}$ of Bun et al. [8] for the interior point problem below and then discuss how to make the algorithm oblivious and implement it with a RAM machine with finite word size, as well as analyze its complexity. For
the algorithm to be a useful building block for differentially oblivious algorithms, we assume that the input database may contain certain dummy elements but with sufficiently many non-dummy elements. The following algorithm is taken almost verbatim from Bun et al. [8], where the algorithm simply ignores the dummy elements. We refer the readers to [8] for the intuition behind the algorithm.

**InteriorPoint**($S, \beta, \epsilon, \delta$)

**Assume:** Database $S = (x_j)_{j=1}^{n'} \in (X \cup \{\text{dummy}\})^{n'}$ with $n$ non-dummy elements.

**Algorithm:**

1. If $|X| \leq 32$, then use the exponential mechanism with privacy parameter $\epsilon$ and quality function $q(S, x) = \min\{\#\{j : x_j \geq x\}, \#\{j : x_j \leq x\}\}$ to choose and return a point $x \in X$. Specifically, each $x \in X$ is chosen with probability proportion to $e^{\epsilon q(S, x)/2}$ (since the sensitivity of the quality function is 1).

2. Let $k = \lceil \frac{386}{\epsilon} \ln \left(\frac{4}{3\delta}\right) \rceil$, and let $Y = (y_1, y_2, \ldots, y_{n-2k})$ be a random permutation of the smallest $(n-2k)$ elements in $S$.

3. For $j = 1$ to $\frac{n-2k}{2}$, define $z_j$ as the length of the longest prefix for which $y_{2j-1}$ agrees with $y_{2j}$ (in base 2 notation).

4. Execute **InteriorPoint** recursively on $S' = (z_j)_{j=1}^{(n-2k)/2} \in (X')^{(n-2k)/2}$ with parameters $\beta, \epsilon, \delta$. Recall that $|X'| = \log |X|$. Denote the returned value by $z$. \[8\]

5. Use the choosing mechanism to choose a prefix $L$ of length $(z + 1)$ with a large number of agreements among elements in $Y$. Use parameters $\beta, \epsilon, \delta$, and the quality function $q : X^* \times \{0,1\}^{z+1} \rightarrow \mathbb{N}$, where $q(Y, L)$ is the number of agreement on prefix $L$ among $y_1, \ldots, y_{n-2k}$. Specifically, the choosing mechanism simply chooses one of prefixes with non-zero quality using exponential mechanism. Namely, each prefix $L$ with $q(Y, L) \geq 1$ is chosen with probability proportion to $e^{-q(Y, L)/2}$. \[8\]

6. For $\sigma \in \{0, 1\}$, define $L_\sigma \in X$ to be the prefix $L$ followed by $(\log |X| - z - 1)$ appearances of $\sigma$.

7. Compute $\text{big} = \text{Lap}\left(\frac{1}{\epsilon}\right) + \#\{j : x_j \geq L_1\}$.

8. If $\text{big} \geq \frac{3k}{2}$ then let $\text{ret} = L_1$. Otherwise let $\text{ret} = L_0$.

9. If $\text{ret}$ is not an interior point, then return any $x \in X$. Otherwise, return $\text{ret}$. \[9\]

Our goal is to implement the algorithm obliviously in a RAM machine with a finite word size efficiently. Note that for obliviousness, we cannot reveal the number $n$ of non-dummy elements. This is simple for step 3, 6, 7, and 8 by adding dummy access. For the recursion step 4, we can invoke the recursion with $(n' - 2k)/2$ size database with $(n - 2k)/2$ non-dummy elements. For step 2, we need to use random permutation to pair up $n - 2k$ smallest non-dummy elements in an oblivious way. This can be done with the help of oblivious sorting as follows.

---

8 See [8] for definition and properties of the choosing mechanism. Here we only describe the behavior of the choosing mechanism is this specific case.

9 We remark that the algorithm in [8] directly returns $\text{ref}$ and may fail to output an interior point with probability $O(\beta)$. We modify it so that it always output an interior point at the price of increasing the privacy parameter by $O(\beta)$ and set $\beta = O(\delta)$. 

43
- We first use oblivious sorting to identify the \( n - 2k \) smallest non-dummy elements in \( Y \). Namely, we sort the elements according to its value and mark the \( n - 2k \) smallest non-dummy elements.

- We apply a random permutation to the \( n' \) elements. Note that this permutes the \( n - 2k \) marked elements uniformly. Also note that we do not need to hide the permutation since it is data independent.

- We use a stable oblivious sorting again to move the \( n - 2k \) marked elements together. Specifically, we view the marked and unmarked elements as having values 0 and 1, respectively and we sort according to this value. The output \( Y = \{ y_1, y_2, \ldots, y_{n-2k} \} \) are the first \( n - 2k \) elements. Since we use stable sorting, the order among the \( n - 2k \) elements remains randomly permuted.

The time complexity of this step is \( O(n' \cdot \log n') \)

We now discuss how to implement the exponential and choosing mechanisms in step 1 and 5, which involves sampling from a distribution defined by numbers of the form \( e^{(r/2) \cdot j} \) for integer \( j \in \mathbb{Z} \). To implement the sampling in a RAM machine with a finite word size, we need to compute the values with enough precision to maintain privacy, which may cause efficiency overhead. We discuss how to do it efficiently. We focus on step 5 since it is the more involved step. Step 1 can be done in an analogous way.

To implement the choosing mechanisms in step 5, the first step is to compute the quality function for all prefixes \( L \in \{0, 1\}^{z+1} \) with \( q(Y, L) \geq 1 \) obliviously. Let \( P = \{ L : q(Y, L) \geq 1 \} \) denote the set of such prefixes. Recall that \( q(Y, L) = \# \{ y_i \in Y : \text{pref}(y_i) = L \} \), where \( \text{pref}(y_i) \) denote the \( z + 1 \) bits prefix of \( y_i \). This can be done by invoking oblivious aggregation (see Section C.3) with key \( k = \text{pref}(y_i) \) and value \( v_i = 1 \) (with padded dummy entries to size \( n' - 2k \)), and the aggregation function \( \text{Aggr} \) being the summation function. The output is an array \( \{ (L_j, q_j) \} \) of the same size \( n' - 2k \) where the first half contains all prefixes \( L_j \in P \) with \( q_j = q(Y, L_j) \geq 1 \) and the remaining are dummy entries.

Now we need to sample a prefix \( L_j \) with probability proportion to \( e^{(r/2) \cdot q_j} \). To optimize the efficiency, we do it as follows. Let \( q_{\text{max}} = \max_j q_j \). We compute \( w_j = e^{(r/2) \cdot (q_j - q_{\text{max}})} \in [0, 1] \) with \( p = 2 \cdot \log(n/\delta) \) bits of precision. We set \( w_j = 0 \) for the dummy entries. Then we compute the accumulated sum \( v_j = \sum_{\ell < j} [2^\ell \cdot w_j] \). Finally, we sample a uniform \( u \leftarrow_R [v_{n' - 2k}] \) and output the \( L_j \) such that \( v_{j-1} < u \leq v_j \) (we set \( v_0 = 0 \)). It is not hard to see that this samples \( L_j \) with the correct distribution up to an \( o(\delta) \) statistical distance error due to the finite precision. To compute \( w_j = \{ e^{(r/2) \cdot q_j - q_{\text{max}}} \} \) with \( p \) bits of precision, note that these are values of the form \( e^{(r/2) \cdot t} \) for \( t \in \mathbb{N} \). Since we only need \( p \) bits of precision, the value rounds to 0 when \( t \) is too large. Let \( t_{\text{max}} = O((1/r) \cdot \log p) \) be the largest \( t \) we need to consider. We precompute the values \( \alpha_k = e^{-(r/2) \cdot 2^k} \) for \( k \in \{0, 1, \ldots, \lfloor \log t_{\text{max}} \rfloor \} \), and compute \( w_j \) by multiplying a subset of \( \alpha_k \), i.e., by the standard repeated squaring algorithm. (Note that for obliviousness, the access pattern needs to go over all \( \alpha_k \) to hide the subset). Finally, to compute \( \alpha_k \), we first compute \( \alpha_0 = e^{-r/2} \) using Taylor expansion, and then compute \( \alpha_k = \alpha_{k-1}^2 \) by multiplication.

We summarize below the implementation of the choosing mechanism in step 5 discussed above. Note that it has a deterministic access pattern.

**Choosing(\( Y, z, \epsilon \))**

Assume: \( Y = (y_j)_{j=1}^{n-2k} \in (X \cup \{\text{dummy}\})^{n' - 2k} \) with \( n - 2k \) non-dummy elements.

**Algorithm:**
1. Compute the quality function: Invoke oblivious aggregation (see Section C.3) with key $k_i = \text{pref}(y_i)$ and value $v_i = 1$ (with padded dummy entries to size $n' - 2k$), and the aggregation function $\text{Aggr}$ being the summation function. The output is an array $\{(L_j, q_j)\}$ of the same size $n' - 2k$ where the first half contains all prefixes $L_j \in P$ with $q_j = q(Y, L_j) \geq 1$ and the remaining are dummy entries.

2. Compute $\alpha_0 = e^{-\varepsilon/2}$ by Taylor expansion, and $\alpha_k = \alpha_{k-1}^2$ for $k \in \{1, \ldots, \lceil \log t_{\text{max}} \rceil \}$.

3. Compute the weights $w_j = e^{(\varepsilon/2) \cdot (q_j - q_{\text{max}})}$ by multiplying a proper subset of $\alpha_k$. Set $w_j = 0$ for the dummy entries. (Note that for oblivious security, we need to do dummy computation to go over all $\alpha_k$ for all $j$.)

4. Compute the accumulated sum $v_j = \sum_{t \leq j} 2^p \cdot w_j$. Set $v_0 = 0$.

5. Sample a uniform $u \leftarrow_R [v_{n' - 2k}]$. Use a linear scan to find $j$ such that $v_{j-1} < u \leq v_j$ and output $L_j$. (Note that we cannot do binary search here for oblivious security.)

We now analyze the complexity of the above choosing mechanism. The first step takes $O(n' \log n')$ time for oblivious aggregation. For the sampling steps, for clarity, let us use $\text{time}_\text{add}(p)$, $\text{time}_\text{mult}(p)$, $\text{time}_\exp(p)$, etc., to denote the time to perform addition and multiplication, and compute $e^{-\varepsilon/2}$, etc., with $p$ bits of precision. We note that the dominating cost is the computation of the weights $w_j$, which takes $O(n' \cdot (\log t_{\text{max}}) \cdot \text{time}_\text{mult}(p))$ time. Other costs that are linear in $n'$ are the computation of accumulated sum and the search of index $j$ such that $v_{j-1} < u \leq v_{j}$, which takes linear number of addition and comparison, respectively. The remaining costs are the computation of $\alpha_k$’s, which takes time $\text{time}_\text{exp} + (\log t_{\text{max}}) \cdot \text{time}_\text{mult}(p)$, and the sampling of $u \leftarrow_R [v_{n' - 2k}]$. Note that $\text{time}_\text{exp}(p)$ is at most $O(p \cdot \text{time}_\text{mult}(p)) \leq O(n' \cdot \text{time}_\text{mult}(p))$. All these terms are dominated by the cost of computing $w_j$’s.

Therefore, the total cost of $\text{InteriorPoint}$ without considering the recursion is

$$O(n' \log n') + O(n' \cdot (\log t_{\text{max}}) \cdot \text{time}_\text{mult}(p)),$$

where $\log t_{\text{max}} = O(\log((1/\varepsilon) \cdot \log(n/\delta)))$ and $p = O(\log(n/\delta))$. Finally, note that $\text{InteriorPoint}$ only invokes the recursion once with size shrinking by a factor of 2, so the overall complexity remains the same.

We remark that one may consider to precompute not only $\alpha_k = e^{-(\varepsilon/2) \cdot 2^t}$ for $k \in \{0, 1, \ldots, \lceil \log t_{\text{max}} \rceil \}$, but all the values $e^{-(\varepsilon/2) \cdot t}$ for $t \in \{0, 1, \ldots, t_{\text{max}}\}$ so that we do not need to compute the weight $w_j$’s by repeated squaring. However, note that we cannot directly fetch the precomputed value for $w_j$ since it breaks the oblivious security, and it seems that to maintain oblivious security, an $O(\log t_{\text{max}})$ overhead is needed. This can yield a small asymptotic improvement over the above solution, but we choose to present the above solution for simplicity.

Let $w$ denote the word size of a RAM machine and suppose word operations have unit cost. We know that $\text{time}_\text{mult}(p) \leq O((p/w)^2)$. When $\delta = \text{neg}(n')$ and $w = O(\log n')$, we have $p/w = \omega(1)$ and the overall complexity is $O(n' \log n')$. We summarize the above discussion in the following theorem.

**Theorem A.10** (Differentially private interior point, finite word size version). Let $w$ be the word size of a RAM machine. Let $\beta, \varepsilon, \delta > 0$ be parameters. There exists an algorithm such that given any array $S$ containing $n'$ elements from a finite universe $X = [0, U - 1]$ with some dummy elements but at least $n$ non-dummy, where $n \geq \frac{18500}{\varepsilon} \cdot 2^{\log^* U} \cdot \log^* U \cdot \ln \frac{4 \log^* U}{\varepsilon \delta}$, the algorithm

- completes consuming only $O\left((n' \log n') + (n' \cdot (\log((1/\varepsilon) \cdot \log p) \cdot (p/w)^2))\right)$ time and number of memory accesses, where $p = 2 \log(n/\delta)$. 

45
the algorithm produces an \((\epsilon,\delta)\)-differential private outcome;

- the algorithm is perfect correct, i.e., the outcome is a correct interior point of the input array \(S\) with probability 1; and

- the algorithm’s memory access patterns are independent of the input array.

## B Limits of Differentially Oblivious Algorithms with Imperfect Correctness

In this section, we discuss how to extend our lower bounds for differentially oblivious algorithms to handle imperfect correctness (with small correctness error). The main observation is that it is sufficient to generalize Lemma 4.3 to handle imperfect correctness. Then, the lower bound follows by replacing Lemma 4.3 to the generalized version in the original proof. We first refine the definition of plausible access pattern.

### Definition B.1 (Good access pattern)

An access pattern \(A\) produced by a mechanism \(M\) is good for an input \(I\), if

\[
\Pr[(\text{Accesses}^M(\lambda, I) = A) \land (M \text{ outputs correct answers})] > 0;
\]

otherwise, we say that \(A\) is bad for \(I\).

### Lemma B.2

Suppose \(I_0\) is some input for a mechanism \(M\) that is \((\epsilon,\delta)\)-differentially oblivious with correctness error \(\gamma\), and \(C\) is a collection of inputs that are \(r\)-neighbors of \(I_0\). Then, the probability that \(\text{Accesses}^M(\lambda, I_0)\) is good for all inputs in \(C\) is at least \(1 - \eta\), where \(\eta := |C| \cdot (e^{\epsilon r} \cdot \gamma + \frac{e^{\epsilon r} - 1}{e^\epsilon - 1} \cdot \delta) \leq |C| \cdot e^{r \epsilon} \cdot (\gamma + \delta)\).

**Proof.** Let \(S\) be the set of access patterns that are good for input \(I_0\), which happens with probability at least \(1 - \gamma\). For each \(I_i \in C\), define \(S_i \subset S\) to be the subset of access patterns in \(S\) that are bad for \(I_i\).

By Fact C.1, we have

\[
\Pr[\text{Accesses}^M(\lambda, I_0) \in S_i] \leq e^{\epsilon r} \cdot \Pr[\text{Accesses}^M(\lambda, I_i) \in S_i] + \frac{e^{\epsilon r} - 1}{e^\epsilon - 1} \cdot \delta \leq e^{\epsilon r} \cdot \gamma + \frac{e^{\epsilon r} - 1}{e^\epsilon - 1} \cdot \delta,
\]

where the last equality follows because the access patterns in \(S_i\) are bad for \(I_i\), which happens with probability at most \(\gamma\). Therefore, by the union bound, we have

\[
\Pr[\text{Accesses}^M(\lambda, I_0) \in \cup_{I_i \in C} S_i] \leq |C| \cdot (e^{\epsilon r} \cdot \gamma + \frac{e^{\epsilon r} - 1}{e^\epsilon - 1} \cdot \delta).
\]

Finally, observe that \(\text{Accesses}^M(\lambda, I_0) \in \cup_{I_i \in C} S_i\) is the complement of the event that \(\text{Accesses}^M(\lambda, I_0)\) is plausible for all inputs in \(C\). Hence, the result follows.

### Extending our lower bounds for imperfect correctness.

One interpretation of Lemma B.2 is that an \((\epsilon,\delta)\)-differentially oblivious mechanism \(M\) with correctness error \(\gamma\) can be compared to an \((\epsilon,\delta + \gamma)\)-differentially oblivious mechanism \(\hat{M}\) that is perfectly correct. To get an informal intuition, suppose for the special case, it can be checked (efficiently in an oblivious manner) whether the output of mechanism \(M\) is correct. Then, in the case that the output is incorrect (which happens with probability at most \(\gamma\)), a non-oblivious (but always correct) algorithm can be run. Hence, the
resulting mechanism \( \hat{M} \) is \((\epsilon, \delta + \gamma)\)-differentially oblivious and perfectly correct. However, since in general there might be no efficient and oblivious manner to check the correctness of a mechanism, Lemma B.2 is needed to extend our lower bounds for imperfect correctness. Given Lemma B.2, our oblivious sorting lower bound can be extended to the following:

**Corollary B.3.** Suppose \( 0 < s \leq \sqrt{N} \), \( \epsilon > 0 \), \( 0 < \beta < 1 \) and \( 0 \leq \delta + \gamma \leq \beta \cdot \frac{s}{e} \cdot e^{-2\epsilon s} \). Then, any (randomized) stable 1-bit-key sorting algorithm in the balls-and-bins model that is \((\epsilon, \delta)\)-differentially oblivious must, on some input, incur at least \( \Omega(N \log s) \) memory accesses with probability at least \( 1 - \beta \).

**Proof.** Same as the proof for Theorem 4.7 but replace the usage of “plausible access pattern” there with “good access pattern” instead.

Similarly, our lower bounds for merging and data structures can be extended in the same way using Lemma B.2 to account for imperfect correctness, and we omit the detailed statements.

**C Extra Preliminaries**

**C.1 Technical Lemmas Concerning \( r \)-Neighbors**

**Fact C.1 (r-Neighbors Produce Similar Access Patterns [44]).** Suppose a (randomized) algorithm \( M \) satisfies \((\epsilon, \delta)\)-differential obliviousness, where \( \epsilon \) and \( \delta \) can depend on some security parameter \( \lambda \). Then, for any two inputs \( I \) and \( I' \) that are \( r \)-neighboring and any set \( S \) of access patterns, we have

\[
\Pr[\text{Accesses}^M(\lambda, I) \in S] \leq e^{\epsilon r} \cdot \Pr[\text{Accesses}^M(\lambda, I') \in S] + \frac{e^{\epsilon r} - 1}{e^\epsilon - 1} \cdot \delta.
\]

**Proof of Lemma 4.3.** Let \( S \) be the set of access patterns that are plausible for input \( I_0 \). For each \( I_i \in C \), define \( S_i \subset S \) to be the subset of access patterns in \( S \) that are implausible for \( I_i \).

By Fact C.1, we have

\[
\Pr[\text{Accesses}^M(\lambda, I_0) \in S_i] \leq e^{\epsilon r} \cdot \Pr[\text{Accesses}^M(\lambda, I_i) \in S_i] + \frac{e^{\epsilon r} - 1}{e^\epsilon - 1} \cdot \delta = \frac{e^{\epsilon r} - 1}{e^\epsilon - 1} \cdot \delta,
\]

where the last equality follows because the access patterns in \( S_i \) are implausible for \( I_i \). Therefore, by the union bound, we have

\[
\Pr[\text{Accesses}^M(\lambda, I_0) \in \bigcup_{I_i \in C} S_i] \leq |C| \cdot \frac{e^{\epsilon r} - 1}{e^\epsilon - 1} \cdot \delta.
\]

Finally, observe that \( \text{Accesses}^M(\lambda, I_0) \in \bigcup_{I_i \in C} S_i \) is the complement of the event that \( \text{Accesses}^M(\lambda, I_0) \) is plausible for all inputs in \( C \). Hence, the result follows.

**Proof of Lemma 4.4** Before contraction, there are exactly \( t \cdot |N| + 2t \) edges in the access pattern graph. For each layer \( 1 \leq i \leq t - 1 \), there are exactly \( |N| - 2 \) nodes with in-degree and out-degree being 1. Therefore, the number of edges decreases by \( (t - 1) \cdot (|N| - 2) \) to form the compact graph.

Finally, we observe that \( |N| \leq t \), because at most \( t \) memory location can be accessed in \( t \) accesses.
C.2 Stochastic Analysis for Geometric Distribution

The following simple fact follows in a straightforward manner from the definition of $\text{Geom}$.

**Fact C.2.** For any even $Z$ and any $\epsilon > 0$, for any integers $a,a' \in [0,Z]$ such that $|a-a'| = 1$, we have that

$$\Pr[\text{Geom}^Z(e^\epsilon) = a] \leq e^\epsilon \cdot \Pr[\text{Geom}^Z(e^\epsilon) = a'].$$  

**Lemma C.3** (Moment Generating Function). Suppose $G$ is a random variable having truncated geometric distribution $\text{Geom}^Z(\alpha)$ with support $[0..Z]$ and mean $\frac{Z}{2}$, for some $\alpha > 1$. Then, for $|t| \leq \min\{\frac{1}{2}, \sqrt{2 \ln \frac{(\alpha+1)^2}{4\alpha}}\}$, we have

$$E[e^{tG}] \leq \exp\left(\frac{Z}{2} \cdot t + \frac{4\alpha}{(\alpha-1)^2} \cdot t^2\right).$$

**Proof.** Let $V$ be a random variable whose distribution the untruncated variant of $\text{Geom}(\alpha)$ that is symmetric around 0, i.e., for all integers $x$, its probability mass function is proportional to $\alpha^{-|x|}$.

Let $W$ be the truncated variant of $\text{Geom}(\alpha)$ that is symmetric around 0 and has support in $[-\frac{Z}{2}, \frac{Z}{2}]$. Hence, $G$ has the same distribution as $\frac{Z}{2} + W$.

It can be shown that for any real $t$, $E[e^{tW}] \leq E[e^{tV}]$. This follows from the fact that the function $i \mapsto e^{ti} + e^{-ti}$ is increasing for positive integers $i$.

Therefore, $E[e^{tG}] \leq e^{\frac{Z}{2}t} \cdot E[e^{tV}]$. Finally, the result follows from applying a technical result from \cite{13} Lemma 1, Appendix B.1 to get an upper bound on $E[e^{tV}]$ the specified range of $t$. \hfill $\square$

**Lemma C.4** (Measure concentration for truncated geometric random variables). Let $G_B$ denote the sum of $B$ independent $\text{Geom}^Z(e^{\alpha})$ random variables (each of which having mean $\frac{Z}{2}$ and support $[0..Z]$). For any $B$, for sufficiently large $\lambda$ and $Z \geq \frac{\log^2 \lambda}{\epsilon_0}$, it holds that

$$\Pr\left[ G_B \geq \frac{BZ}{2} \cdot \left(1 + \frac{1}{\log^2 \lambda}\right) \right] \leq \exp(-\log^2 \lambda)$$

and

$$\Pr\left[ G_B \leq \frac{BZ}{2} \cdot \left(1 - \frac{1}{\log^2 \lambda}\right) \right] \leq \exp(-\log^2 \lambda).$$

**Proof.** We prove the first inequality. The second inequality can be proved using the same approach. Denote $R := \frac{Z}{2 \log^2 \lambda}$ and $\alpha = e^{\epsilon_0}$. Using the standard argument as in the proof of the Chernoff Bound, for positive $t \leq \min\{\frac{1}{2}, \sqrt{2 \ln \frac{(\alpha+1)^2}{4\alpha}}\}$ in the range specified in Lemma C.3, we have

$$\Pr\left[ G_B \geq \frac{BZ}{2} \cdot \left(1 + \frac{1}{\log^2 \lambda}\right) \right] \leq \exp\left(\frac{4\alpha}{(\alpha-1)^2} \cdot t^2 - Rt\right) \leq \exp\left\{\frac{4\alpha}{(\alpha-1)^2} \cdot t^2 - R\right\},$$

where the last inequality holds if we pick $t > 0$ such that the exponent in the last term is negative.

Hence, it suffices to analyze the exponent for two cases of $\epsilon_0$.

The first case is when $\epsilon_0$ is some large enough constant. In this case, we set $t > 0$ to be some appropriate constant, and the exponent is $-\Theta(R) = -\Theta\left(\frac{Z}{2 \log^2 \lambda}\right) \leq -\log^2 \lambda$.

The second case is when $\epsilon_0$ is small. In this case, $\frac{4\alpha}{(\alpha-1)^2} = \Theta\left(\epsilon_0\right)^2$. We set $t$ to $\sqrt{2 \ln \frac{(\alpha+1)^2}{4\alpha}} = \Theta(\epsilon_0)$. Hence, the exponent is $-\Theta(R\epsilon_0) = -\Theta\left(\frac{Z\epsilon_0}{2 \log^2 \lambda}\right) \leq -\log^2 \lambda$. This completes the proof of the first inequality. \hfill $\square$
C.3 Existing Building Blocks

C.3.1 Oblivious Aggregation

Oblivious aggregation is the following primitive where given an array of (key, value) pairs, each representative element for a key will learn some aggregation function computed over all pairs with the same key.

• Input: An array $\text{Inp} := \{(k_i, v_i)\}_{i \in [n]}$ of possibly dummy (key, value) pairs. Henceforth we refer to all elements with the same key as the same group. We say that index $i \in [n]$ is a representative for its group if $i$ is the leftmost element of the group.

• Output: Let $\text{Aggr}$ be a publicly known, commutative and associative aggregation function and we assume that its output range can be described by $O(1)$ number of blocks. The goal of oblivious aggregation is to output the following array:

$$\text{Outp}_i := \begin{cases} \text{Aggr} \left( \{(k, v) | (k, v) \in \text{Inp} \text{ and } k = k_i\} \right) & \text{if } i \text{ is a representative} \\ \bot & \text{o.w.} \end{cases}$$

Oblivious aggregation can be implemented in time $O(n \log n)$ with a deterministic access pattern.

C.3.2 Oblivious Propagation for a Sorted Array

Oblivious propagation [38] is the opposite of aggregation. Given an array of possibly dummy (key, value) pairs where all elements with the same key appear consecutively, we say that an element is the representative if it is the leftmost element with its key. Oblivious propagation aims to achieve the following: for each key, propagate the representative’s value to all other elements with the same key. Nayak et al. [38] show that such oblivious propagation can be achieved in $O(\log n)$ steps consuming $n$ CPUs where $n$ is the size of the input array.

C.3.3 Oblivious Bin Placement

Oblivious bin placement is the following task: 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.

$$\text{ObliviousBinPlace}(X, V, Z):$$

• Let $W$ be the accumulated sum of $V$, i.e., $W[i] = \sum_{j \leq i} V[j]$. Obliviously sort all elements of $X$ and $W$ together, where each element carries the information whether it comes from $X$ or $Z$. The sorting is governed by the following key assignments: an element in $X$ is assigned the key that equals to its position in $X$, and the $i$-th element in $W$ is assigned the key that is equal to $W[i]$. If two elements have the same key, then the one from $W$ appears later.

• In this sorted array, every element from $X$ wants to hear from the first element from $W$ that appears after itself. We accomplish this by calling an oblivious propagation algorithm (see Section C.3.2) such that at the end, each element from $X$ learns which bin it is destined for.
• In one scan of the resulting array, mark every element from $W$ as dummy. For every $i \in [B]$ where $B = |W|$, append $Z$ filler elements destined for bin $i$ to the array (note that fillers are not dummies).

• Obliviously sort the outcome array by destined bin number, leaving all dummies at the end. If two elements have the same bin number, filler appear after real elements.

• In one linear scan, mark all but the first $Z$ elements of every bin as dummy.

• Obliviously sort by bin number, leaving all dummies at the end. If two elements have the same bin number, break ties by arranging smaller elements first, and having fillers appear after real elements.

• Truncate the result and preserve only the first $BZ$ elements. In one scan, rewrite every filler as dummy.

• Return the outcome.