Foundations of Differentially Oblivious Algorithms

T-H. Hubert Chan HKU

Kai-Min Chung Academia Sinica

Bruce Maggs Duke

Elaine Shi Cornell

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 solution of oblivious RAM (ORAM) simulation. Such a notion has stimulated much debate. Some have argued that the notion of ORAM is too strong, and suffers from a logarithmic lower bound on simulation overhead. Despite encouraging progress in designing efficient ORAM algorithms, it would nonetheless be desirable to avoid the oblivious simulation overhead. Others have argued that obliviousness, without protection of length-leakage, is too weak, and have demonstrated examples where entire databases can be reconstructed merely from length-leakage.

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

1 Introduction

Consider a computing paradigm where a *trusted CPU* accesses sensitive data stored in *untrusted memory* while performing computation. This paradigm is of broad relevance in numerous application domains. For instance, in *cloud outsourcing*, the CPU is a client and the memory is a cloud server [23, 45, 52]. In *secure processor design*, the CPU is tamper-proof but the memory and system buses can be easily tapped [36, 38, 42]. In *secure multi-party computation* (MPC) [5, 25, 37], a trusted (abstract) CPU is instantiated with MPC, and memory is possibly shared among multiple parties. Henceforth in this paper, we also refer to a CPU as a *client*, and memory as a *server*. It is well-understood that data access patterns can leak highly sensitive information such as cryptographic keys [28, 29, 53]. Thus understanding how to eliminate or mitigate such leakage to obtain meaningful privacy guarantees is an important challenge.

To tackle this challenge, the majority of earlier works adopt "obliviousness" as the desired privacy guarantee. Informally speaking, obliviousness requires that a simulator, while not knowing the inputs to a program but possibly knowing how many records are accessed, must be able to simulate the access pattern of a program, such that the simulated access pattern is (computationally or statistically) indistinguishable from that generated by the real algorithm. Indeed, an exciting line of research on oblivious RAM (ORAM) initiated by Goldreich and Ostrovsky [19, 20] has culminated in recent improvements [10, 34, 50] showing that any RAM program can be compiled to an oblivious counterpart while incurring only $O(\log^2 N)$ blowup in runtime.

Obliviousness as a privacy notion, however, is somewhat controversial. Some argue that a full simulation notion is *too strong* to be efficiency-friendly: although recent works [10, 46, 50] have tremendously improved the practical performance of ORAM, it would nonetheless be desirable to avoid the general $O(\log^2 N)$ blowup of ORAM [10, 23, 34, 46, 50]. Others argue that in some scenarios, the obliviousness notion, without protecting the lengths of queries, is *too weak* — for example, the recent work by Kellaris et al. [29] suggested that under certain assumptions, even entire databases can be reconstructed by observing merely length leakage. These philosophical debates converge to a few important questions:

What is a good notion of access pattern privacy?

Is achieving obliviousness a necessary stepping stone in attaining any meaningful notion of access pattern privacy?

Can we circumvent impossibility results associated with the standard notion of obliviousness and still attain meaningful notions of access pattern privacy?

Encouragingly, our community has recently begun explorations in these directions. Our work is inspired by the recent work of Kellaris et al. [30], who investigated how to achieve access pattern differential privacy in outsourced database queries. Their work gives a solution for the restricted scenario in which a static database (allowing range and other queries) is outsourced to a remote server after preprocessing. For the case of dynamic data, Kellaris et al. [30] describes a partial solution where they store records in an ORAM and fetch a random number of additional records beyond what is necessary to obfuscate the true number of records matching each query. This solution is partial in that (1) they make the following strong assumptions: the client can store an *unbounded* amount of metadata, and metadata accessess are *unobservable* by the adversary; and (2) their solution nevertheless resorts to using ORAM as a blackbox and thus leaves largely unexplored whether there is a meaningful weaker or incomparable notion of access pattern privacy that does not imply "obliviousness", and whether such a notion is achievable without having to resort to full obliviousness.

1.1 Differential Obliviousness

Inspired by Kellaris et al. [30], we apply the standard notion of differential privacy [16] to access patterns, and call the resulting notion *differential obliviousness*. For generality, we formulate differential obliviousness for random access machines (RAMs) where a trusted CPU with $O(1)$ registers interacts with an untrusted memory to perform computation. Our formulation is applicable to all the application domains mentioned earlier including cloud outsourcing, secure processors, and MPC.

The relationship between differential obliviousness and (simulation-based) obliviousness (see Goldreich and Ostrovsky [19, 20]) is analogous to the relationship between differential privacy [16] and (simulation-based) privacy (see Dalenius [15][1]). While obliviousness requires that the access pattern of an algorithm be simulatable by a simulator that does not know the algorithm's input but possibly knowing certain length leakage, differential obliviousness requires that the algorithm's access pattern be close in distribution for any two neighboring inputs that differ only slightly from each other.

Differential obliviousness can also be intuitively interpreted as differential privacy [16, 48], but now the observables are access patterns. Informally, we would like to guarantee that an adversary

[1] It was pointed out that the Dalenius's fully simulatable privacy notion is unattainable if any meaningful utility is required [16], but fully simulatable obliviousness is of a very different nature and is possible.

such as a cloud server, after having observed access patterns to (encrypted)[2] 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. Since the data can be encrypted, we assume that the server cannot observe the contents of the data, nor does the server learn the outcome of the algorithm (except if the algorithm "voluntarily" discloses intermediate, differentially private statistics[3]).

We make a couple brief remarks at this moment regarding this new notion:

1. As we will explain in more detail later, depending on whether length-leakage is of concern, our differentially obliviousness notion is in some cases weaker than the standard notion of obliviousness (e.g., for sorting), but in other cases incomparable (e.g., for data structures — note that earlier works on oblivious data structures [31, 39, 51] leak the number of records matching each query which would violate our notion of differential obliviousness).

2. Our differential obliviousness notion demands some form of obfuscation of access patterns. Since the definition is close in nature to the existing notion of differential privacy (but now the observables are access patterns), one natural question is whether and how one can leverage differentially private mechanism design techniques to obfuscate access patterns. At first sight, though, this appears difficult since differentially private mechanisms [16, 48] typically protect individual records "in the aggregate"; but for access patterns, each (input-dependent) data movement is observable in a singled-out fashion, and there does not appear to be any immediate way of adding noise to such individual data movements while preserving the algorithm's correctness.

1.2 Our Results

Equipped with this new notion, we investigate the feasibility and infeasibility of efficiently realizing some of the most fundamental algorithmic abstractions, including sorting and data structures, while satisfying differential obliviousness. Through this process, we not only formulate a new theory for differentially oblivious algorithms, but also demonstrate several novel techniques for proving both upper- and lower-bounds. Our lower bounds draw new connections to the complexity of routing graphs that were studied in the classical algorithms literature [41]. On the upper bound front, we are the first to integrate techniques from differentially private mechanisms and oblivious algorithms in a non-trivial and non-blackbox manner. Perhaps somewhat counter-intuitively, we show that indeed, meaningful notions of privacy can be attained, without resorting to full obliviousness — this allows us to not only circumvent several impossibilities pertaining to the classical notion of obliviousness, but also show, in several cases, that we can attain meaningful privacy with only slightly larger overhead than the best-known non-private baselines (i.e., almost for free).

Our work should be viewed as the beginning of a fruitful line of research that will answer many natural open questions, which will be encountered as one thoroughly explores the theoretical landscape of differentially oblivious algorithms. We now elaborate on our results.

1.2.1 Sorting

This is one of the most fundamental and most classical algorithmic abstraction. In this paper, we consider (possibly non-comparison-based) sorting in the *balls-and-bins* model: imagine that there are N balls each tagged with a k-bit key. We would like to sort the balls based on the relative

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

[3] In our security proofs, we show that typically the access patterns depend only on these voluntarily disclosed statistics, and hence are also differentially oblivious.

ordering of their keys. 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.

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 [2, 26, 27, 32, 47]) — and a subset of these techniques apply to the balls-and-bins model. However, a recent manuscript by Lin, Shi, and Xie [35] showed that interesting barriers arise if we require the classical notion of obliviousness for sorting.

Fact 1.1 (Barriers for oblivious sorting [35])**.** *Any oblivious* 1*-bit stable sorting algorithm in the balls-and-bins model, even non-comparison-based ones, must incur at least* $\Omega(N \log N)$ *runtime (even when allowing a constant probability of security or correctness failure). As a direct corollary, any general oblivious sorting algorithm in the balls-and-bins model, even non-comparison-based ones, must incur at least* $\Omega(N \log N)$ *runtime.*

We stress that the above oblivious sorting barrier is applicable only in the balls-and-bins model (otherwise without the balls-and-bins constraint, the feasibility or infeasibility of $o(n \log n)$-overhead, circuit-based sorting remains open [7]). Further, as Lin, Shi, and Xie showed [35], for small-length keys, the barrier also goes away if the stability requirement is removed (see Section 1.4 for more explanations).

Differentially oblivious sorting. A sorting algorithm M is said to be (ϵ, δ)-differentially oblivious, if for any two neighboring input arrays I and I' of equal length that differ only in one position, for any set S of access patterns, it holds that

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

where $\mathbf{Accesses}^M(I)$ denotes the ordered sequence of memory accesses made by the algorithm M upon receiving the input I.

By this definition, (ϵ, δ)-differentially oblivious sorting is a strict relaxation of the classical notion of oblivious sorting (which is equivalent to the case $\epsilon = \delta = 0$). We thus ask: *can we use this relaxation to overcome the aforementioned barriers for oblivious sorting* (in the balls-and-bins model)?

We give a two-sided answer to this question. On the positive side, we show that for reasonable choices of ϵ and δ, indeed, one can sort small-length keys in $o(N \log N)$ time and attain (ϵ, δ)-differential obliviousness. As a typical parameter choice, for $\epsilon = \Theta(1)$ and δ being a suitable negligible function in N, we can *stably* sort N balls tagged with 1-bit keys in $O(N \log \log N)$ time — note that in this case, the best non-private algorithm takes linear time, and thus we show that privacy is attained "almost for free" for 1-bit stable sorting. More generally, for any $k = o(\log N / \log \log N)$, we can stably sort k-bit keys in $o(N \log N)$ time — in other words, for small-length keys we overcome the $\Omega(N \log N)$ barrier of oblivious sorting. We state our result more formally and for generalized parameters:

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

Note that the above upper bound statement allows for general choices of ϵ and δ. We can observe that for $\epsilon = \Theta(1)$, if one demands an exceedingly small δ, say, $\delta = \exp(-N^{0.1})$ or any subexponential function, the upper bound would become $O(N \log N \log\log N)$ for $k = 1$. Interestingly, we show that our upper bound result is *optimal* up to log log factors for a wide parameter range. We present our lower bound statement for general parameters first, and then highlight several particularly interesting parameter choices and discuss their implications. Note that our lower bound below is applicable even to non-comparison-based sorting:

Theorem 1.3 (Lower bound for (ϵ, δ)-differentially oblivious sorting in the balls-and-bins model)**.** *For any* $0 < s \le \sqrt{N}$*, any* $\epsilon > 0$*, and any* $0 \le \delta \le e^{-(2\epsilon s + \log^2 N)}$*, any* (ϵ, δ)*-differentially oblivious stable* 1*-bit sorting algorithm in the balls-and-bins model must incur, on some input, at least* $\Omega(N \log s)$ *memory accesses with high probability.*

As an immediate corollary, under the same parameter assumptions, any (ϵ, δ)*-differentially oblivious general (possibly non-stable) balls-and-bins sorting algorithm must incur, on some input, at least* $\Omega(N \log s)$ *memory accesses with high probability.*

We now highlight several interesting parameter choices and discuss their implications:

- First, note that how the lower bound tightly matches the upper bound (up to log log factors) for $\epsilon = \Theta(1)$ and typical choices of δ, e.g., $\delta = \exp(-\log^2 N)$ or $\delta = \exp(-N^{0.1})$.
- Second, the lower bound allows a tradeoff between ϵ and δ. For example, here is another interesting parametrization: if $\epsilon = \Theta(\frac{1}{\sqrt{N}})$, then we rule out $o(N \log N)$ stable 1-bit sorting for even $\delta = \exp(-\Omega(\log^2 N))$.
- Perhaps even more interestingly, if one requires that $\delta = 0$, then we may conclude that *even when* ϵ *may be arbitrarily large*, any ϵ-differentially oblivious sorting algorithm must suffer from the same lower bounds as oblivious sorting (in the balls-and-bins model)! This is a surprising conclusion because in some sense, very little privacy (or almost no privacy) is attained for large choices of ϵ — and yet if δ must be 0, the same barrier for full obliviousness carries over!

1.2.2 Merging Two Sorted Lists

Merging is also a classical abstraction and has been studied extensively in algorithms [33]. 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 [41] 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[4].

Differentially oblivious merging. To define the notion of (ϵ, δ)-differential obliviousness for merging, we must specify what it means for two inputs to be neighboring. In merging, the input contains two sorted arrays. Due to the sortedness requirement on the input arrays, it does not

[4] Pippenger and Valiant's proof [41] 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, translate their lower bound to the RAM setting.

make sense to flip one coordinate and change it to an arbitrary other value. Instead, we say that two sorted arrays I and I' are neighboring, if the multiset defined by the two input arrays would become identical by removing one element from each array. For merging, we say that two inputs (I_0, I_1) and (I'_0, I'_1) are neighboring if for either $b = 0$ or $b = 1$, I_b and I'_b are neighboring and $I_{1-b} = I'_{1-b}$. Given this new notion of neighboring, differentially oblivious for merging could be defined in exactly the same manner as that for sorting. We stress that such a notion of neighboring is natural and meaningful in practical applications — for example, our data structure construction later makes use of merging as a building block where this natural notion of neighboring is desired.

We show similar results for merging as those for 1-bit stable sorting.

- First, we prove that assuming $\epsilon = \Theta(1)$, if δ must be subexponentially small, then the same lower bound for oblivious merging will be preserved for (ϵ, δ)-differentially oblivious merging.
- Second, we show that for $\epsilon = \Theta(1)$ and δ negligibly small (but not subexponentially small), we can achieve (ϵ, δ)-differentially oblivious merging in $O(N \log \log N)$ time — yet another example of having privacy with only slightly higher overhead.
- 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.

We present the informal theorem statements below and defer the formal versions to the detailed technical sections.

Theorem 1.4 (Lower bound for (ϵ, δ)-differentially oblivious merging in the balls-and-bins model)**.** *For any $0 < s \leq \sqrt{N}$, any $\epsilon > 0$, and any $0 \leq \delta \leq e^{-(2\epsilon s + \log^2 N)}$, any (ϵ, δ)-differentially oblivious merging algorithm in the balls-and-bins model must incur, on some input, at least $\Omega(N \log s)$ memory accesses with high probability.*

Theorem 1.5 ((ϵ, δ)-differentially oblivious merging)**.** *For any $\epsilon > 0$ and any $0 < \delta < 1$, there exists an (ϵ, δ)-differentially oblivious merging algorithm that completes in $O(N(\log \frac{1}{\epsilon} + \log \log N + \log \log \frac{1}{\delta}))$ runtime*[5]*. As a special case, for $\epsilon = \Theta(1)$, there exists an $(\epsilon, \mathsf{negl}(N))$-differentially oblivious merging algorithm that completes in $O(N \log \log N)$ runtime for some suitable negligible function $\mathsf{negl}(\cdot)$.*

Note that the parameter constraints in the lower- and upper-bounds resemble those of the 1-bit stable sorting. Thus, we can interpret these results in a similar fashion as 1-bit stable sorting, e.g., by observing how the lower- and upper-bounds tightly match (up to $\log \log$ factors) for a wide range of parameters, and by observing how, when $\delta = 0$, surprisingly, the oblivious merging $\Omega(N \log N)$ barrier will be preserved no matter how large ϵ is (and how little privacy we get from such a large ϵ).

1.2.3 Data Structures Supporting Insertion and Query

Unlike sorting, data structures are *stateful* (rather than stateless) algorithms, where memory states persist across multiple invocations. Like sorting, data structures are also of fundamental importance to computer science. We thus investigate the feasibilities and infeasibilities of efficient, differentially oblivious data structures.

[5] Here we assume here that arithmetic computations on real numbers and sampling from appropriate distributions take unit time. In the appendices, we show how to remove this assumption for a finite-word-length RAM and for negligibly small δ.

For simplicity, we describe only the *static* notion of differential obliviousness here — we use the static notion for our lower bounds but as we argue later, our upper bounds in fact satisfy *adaptive* differential obliviousness where the adversary can choose the operations adaptively over time.

Let $\mathbb{DS}$ be a data structure supporting insertions and queries from a query family $\mathcal{Q}$. We say that two operational sequences ops_0 and ops_1 are *query-consistent neighboring*, if they differ in exactly one position i such that $\mathsf{ops}_0[i]$ and $\mathsf{ops}_1[i]$ must both be insertion operations.

We say that $\mathbb{DS}$ is (ϵ, δ)-differentially oblivious, if for any operational sequences ops_0 and ops_1 that are query-consistent neighboring, for any set S of access patterns, it holds that

$$\Pr[\mathbf{Accesses}^{\mathbb{DS}}(\mathsf{ops}_0) \in S] \leq e^{\epsilon} \cdot \Pr[\mathbf{Accesses}^{\mathbb{DS}}(\mathsf{ops}_1) \in S] + \delta.$$

Similarly as before, such a privacy notion guarantees that an adversary such as a cloud server, having observed the access patterns to the outsourced data structure over all time, learns approximately the same amount of information about any individual record or any event as if that individual record or event did not get inserted in the database. Note that our privacy definition protects the privacy of individual records that get inserted into the database over time, but assumes that queries are public — thus the notion is applicable in scenarios where the data records are sensitive (e.g., hospital patient records) but the queries are not (e.g., medical researchers running standard statistical tests over the dataset). We also show later that any scheme that additionally protects the queries in a differentially oblivious sense must, on some databases (where records are inserted over time), incur $\Omega(N)$ runtime per query even when the number of matching records is small — such privacy notions are less useful since they rule out efficient constructions.

Relation to oblivious data structures. Here our differentially oblivious notion is *incomparable* to the standard notion of oblivious data structures [31,39,51] and ORAM [19,20,23,46,50] considered in prior works. Specifically, oblivious data structures or ORAMs require that the access patterns be simulatable not knowing either insertions or queries, but knowing length leakage — in other words, oblivious data structures and ORAMs do not hide the number of records matching each query. In comparison, here we weaken the fully simulatable notion but on the other hand, our notion requires obfuscating length leakages, such that reconstruction attacks such as Kellaris et al. [29] are provably defeated.

Upper bound results. We consider data structures that support *insertions* and *range queries*. Every insertion operation inserts a record indexed by a search key, and each range query searches for all records whose keys fall within a specified range $[s, t]$. Absent any privacy requirement, such a 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. Note that here we 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 [3, 4, 9]. We show the following results (stated informally).

Theorem 1.6 (Differentially oblivious data structure upper bound)**.** *Suppose that $\epsilon = \Theta(1)$ and that $\mathsf{negl}(\cdot)$ is a suitable negligible function. There is an $(\epsilon, \mathsf{negl}(N))$-differentially oblivious data structure supporting insertions and range queries, where each of the N insertions incurs amortized $O(\log N \log\log N)$ runtime, and each query costs $O(\mathsf{poly}\log N + L)$ runtime where L denotes the number of matching records, and requires accessing only $O(\log N)$ discontiguous memory regions regardless of L.*

In comparison with the non-private solution (i.e., binary search trees), an immediate observation is that for insertions, we achieve privacy with only a slightly higher overhead; and the same can be said for queries that match sufficiently many records, i.e. $L \geq \mathsf{poly}\log N$. Another compelling feature is that when deployed in a client-server setting, our scheme can additionally achieve *non-interactiveness*, i.e., the server can return all matching results in a single roundtrip. Non-interactivity can be achieved in either designated-client or public-client settings. We use the term *designated-client setting* to refer to the scenario where the data owner who performs insertions is also the query-maker. We use *public-client setting* to refer to the scenario where queries are made by third parties other than the data owner who performs insertions. Specifically, in the designated-client setting, we may assume that data is encrypted under the client's private key; whereas in the public-client setting, we may assume that data is encrypted under attribute-based encryption [24, 43] such that the data owner can issue policy-based keys to different query-makers.

We stress that if the classical obliviousness notion is required, even when allowing length leakage, we know of no solution that can simultaneously achieve statistical security (considering only distributions of access patterns), non-interactiveness, and yet retain non-trivial efficiency — not even in the designated client setting. Perhaps the most relevant point of comparison is oblivious data structures [31, 39, 51] and ORAM-based solutions [19, 20, 23, 34, 46, 50]. Unfortunately all known solutions require logarithmically many roundtrips (unless assuming large block size or large client storage), and incur logarithmic or super-logarithmic blowup in cost in comparison with the non-private baseline (recall that our solution attains privacy with only slight increase in overhead). Finally, except for the recent construction by Asharov et al. [3], all other known constructions would suffer linear in L blowup in locality, where L is the number of matching records for a query.

Lower bounds. In the context of data structures, we also prove lower bounds to demonstrate the price of differential obliviousness. Specifically, consider a data structure supporting insertions and *point queries*, each of which requests all records matching a single specified key. Such data structures are commonly referred to as key-value stores and have broad applications in practice. It is well-known that without privacy guarantees, each insertion takes $O(1)$ time and each query takes $L + O(1)$ time, where L is the number of matching records [22]. On the contrary, we show that with differential obliviousness, some additional costs are necessary depending on the choices of ϵ and δ.

We defer the detailed statements and proofs to the technical sections later.

1.3 Technical Highlights

Our upper bound results in fact establish a new paradigm for designing differentially oblivious algorithms. Most of our algorithms adopt the following design pattern: we first rely on differentially private mechanisms to determine a set of intermediate statistics that are safe to release. Importantly, in some cases we show how to make existing differential privacy mechanisms "oblivious" such that their own access patterns do not leak information about the secret data. We then rely on these differentially private statistics such that our algorithms could somewhat approximate the behavior of the non-private algorithms and yet still be safe. Then, we use oblivious algorithms techniques once again, such that our entire algorithm's access patterns are simulatable after having observed the differentially private intermediate statistics. We believe that the tightly-coupled interactions between oblivious algorithms and differential privacy techniques may be of independent interest.

For our lower bounds, well-known ORAM lower bound techniques [19, 20] do not get us very far: for example, using (certain modifications of) Goldreich and Ostrovsky's ORAM lower bound proof [19, 20], we could (for a few cases) prove some non-trivial lower bounds but only for $\delta = 0$.

To prove tight lower bounds covering a wide range of parameter choices, we unveil new connections to the study of routing graph complexity in the classical algorithms literature [41] (note that the concurrent work by Lin et. al. [35] also adopted similar techniques to prove new lower bounds for certain oblivious algorithms). Specifically, we first use probabilistic reasoning to translate lower bounds for differentially oblivious algorithms into an alternative form where we ask: given a fixed access pattern graph, how many input possibilities must be plausible for this access pattern graph to ensure privacy? In particular, some input is plausible for an access pattern graph *iff* there exist node disjoint paths to route from a subset of the input locations to desired output locations. As a concrete example, in sorting, such routing over the access pattern graph allows the input balls to be routed to the correct output locations in memory. We then rely on the elegant results by Pippenger and Valiant on routing graph complexity [41] to derive our tight lower bounds for differentially oblivious algorithms.

1.4 Related Work

Our differential obliviousness notion is inspired by 1) the elegant notion of differential privacy originally proposed by Dwork, McSherry, Nissim and Smith [16]; and 2) the rich line of research on ORAMs [14,19,20,23,44] first proposed in a ground-breaking work by Goldreich and Ostrovsky [19, 20], and numerous results on efficient oblivious algorithms [21,37,40].

Interestingly, differentially oblivious algorithms can be directly applied to the following popular setting considered in the differential privacy literature. For instance, data analysts would like to query statistics from a sensitive database — but now, imagine that the curator would like to outsource the dataset to an untrusted cloud server. Using differentially oblivious algorithms, we can guarantee that even if the cloud server can observe accesses incurred in the lifetime of the database, it cannot harm any individual's privacy — thus allowing us to provide "end-to-end differential privacy" in such a scenario.

Most closely related works. We are inspired by the recent work of Kellaris et al. [30]. To construct a dynamic database supporting point and range queries, Kellaris et al. [30] relied on a generic ORAM to obfuscate access patterns and break input-output links. Then, noise is added to perturb the number of records matching each query to avoid length-leakage. Kellaris et al. assumed that the client can store an unbounded amount of metadata, 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. Thus, Kellaris et al. does not give a satisfying answer for the main questions phrased in our paper, i.e., whether one could achieve meaningful notions of access pattern privacy *without* leveraging ORAM as an intermediate stepping stone, and whether one could circumvent impossibilities pertaining to oblivious algorithms and still achieve meaningful access pattern privacy. On the other hand, since they did adopt a full ORAM, their scheme can achieve somewhat stronger security, e.g., they would be at least as secure as the ORAM. Interestingly, we also remark that even though our approaches have some similarity with Kellaris et al. [30], the techniques are used in a different way. They first use ORAM to break input-output links entirely, and then perturb the outcome lengths. On the contrary, we first run *oblivious* and differentially private mechanisms to obtain intermediate noisy statistics; and then we rely on oblivious algorithms techniques such that our entire algorithm's access pattern is simulatable after having observed the intermediate noisy statistics.

Another closely related work is by Wagh et al. [49], 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 (and in general it is not clear how neighboring for the logical memory requests would translate to neighboring of the inputs where the latter is typically what we care about). Their main algorithm changes the way Path ORAM [46] 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.

Lin, Shi, and Xie [35] 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. [35] in asymptotical performance (up to $\log\log$ factors) but we additionally achieve *stability*, and thus circumventing known barriers pertaining to oblivious sort.

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.

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.

2.2 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 in 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 asymptotical performance (and absorbing the loss in precision into the δ term of (ϵ, δ)-differential obliviousness). We defer discussions on simulating real arithmetic on a finite-word-width RAM to the appendices.

Assumptions on the CPU's private cache. Henceforth in this paper, we assume that *the CPU can store $O(1)$ number of records in its private cache.*

3 Differentially Oblivious Sorting: Definitions and 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).

3.1 Defining Differentially Oblivious Sorting

Let I denote an input array containing N balls each tagged with a k-bit key. We say that two inputs I and I' are *neighboring*, if they are of the same length and differ in exactly one position.

Definition 3.1 (Differentially oblivious sorting)**.** Let $\epsilon(\cdot), \delta(\cdot)$ be functions of a security parameter λ. We say that an algorithm $M(\lambda, I)$ is an (ϵ, δ)-differential obliviousness sorting algorithm iff

- *Perfect correctness.* On any input I and any λ, the algorithm outputs the correctly sorted outcome with probability 1. Further we say that the sorting algorithm is *stable* iff two balls with the same key always appear in the same order in the output as in the input.

- *Differential obliviousness.* For any neighboring inputs I and I', for any λ, for any set S of access patterns,
$$\Pr[\mathbf{Accesses}^M(\lambda, I) \in S] \leq e^{\epsilon(\lambda)} \cdot \Pr[\mathbf{Accesses}^M(\lambda, I') \in S] + \delta(\lambda)$$
where $\mathbf{Accesses}^M(\lambda, I)$ is a random variable denoting the access patterns the algorithm M makes upon receiving the input λ and I.

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

It would be interesting to contrast our differential oblivious definition with the classical notion of oblivious sort.

Definition 3.2 (Oblivious sort)**.** Let $\delta(\cdot)$ be a function of the security parameter λ. A possibly randomized algorithm denoted $M(\lambda, I)$ is said to be a δ-statistically oblivious sorting algorithm iff 1) M satisfies perfect correctness defined in the same manner as above; and 2) for any two inputs I and I' of equal length, it holds that $\Pr[\mathbf{Accesses}^M(\lambda, I) \in S] \overset{\delta(\lambda)}{\equiv} \Pr[\mathbf{Accesses}^M(\lambda, I') \in S]$ where $\overset{\delta(\lambda)}{\equiv}$ denotes that the two distributions are $\delta(\lambda)$-statistically indistinguishable.

It is easy to observe that any δ-statistically oblivious (where $\delta \neq 0$) sorting algorithm is also a (ϵ, δ)-differentially oblivious sorting algorithm (where $\delta \neq 0$); and any perfectly oblivious sorting algorithm is also an $(\epsilon, \delta = 0)$-differentially oblivious sorting algorithm. In other words, our differential obliviousness definition here is strictly weaker than classical oblivious sort. Thus the interesting question is whether this relaxation can allow us to circumvent impossibilities related to oblivious sorting, and under what parameter regimes we can circumvent such impossibilities.

Remark 3.3. *We note that any (ϵ, δ)-differentially oblivious sorting algorithm that has δ' statistical correctness error can be converted to a $(\epsilon, \delta + \delta')$-differentially oblivious algorithm with perfect correctness: we may simply check the outcome and if wrong retry — this may prolong the running time of the algorithm but only with small probability (assuming that δ' is small). Thus requiring perfect correctness in our definition is without loss of generality.*

3.2 Stably Sorting 1-Bit Keys

We start with stably sorting 1-bit keys and later extend to more bits. Stable 1-bit 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.

We choose to start with this special case because interestingly, stable 1-bit 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 [35] — and the lower bound holds even for non-comparison-based sorting algorithms that can perform arbitrary computation on keys. More specifically, they showed that for any constant $0 < \delta < 1$ any δ-oblivious stable 1-bit sorting algorithm must in expectation perform at least $\Omega(N \log N)$ ball movements.

In this section, we will show that adopting our more relaxed differential obliviousness notion can allow us to circumvent the lower bound for oblivious 1-bit stable sorting (in the balls-and-bins model). In particular, for a suitable negligible function δ, we can accomplish 1-bit stable sorting in $O(N \log \log N)$ time for $\epsilon = \Theta(1)$. 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.

3.2.1 Intuition

Absent privacy requirements, clearly tight stable compaction can be accomplished in linear time, by making one scan of the input array, and writing it out whenever a real element 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 about 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: every time we read a small batch of s elements into a working buffer, obliviously sort the working buffer to move all dummies to the end, and then emit some number of elements into the output. 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, 17]. For example, suppose we know that the number of real elements till position i is in between $[C_i - s, C_i + s]$ with high probability, then our algorithm will know to output exactly $C_i - s$ elements when the input array's pointer reaches position i. Furthermore, at this moment, at most $2s$ real elements will have been scanned but have not been output — and these elements will remain in the working buffer. We can now rely on oblivious sorting again to truncate the working buffer and remove dummies, such that the working buffer's size will never grow too large — note that this is important since otherwise obliviously sorting the working buffer will become too expensive. Below we elaborate on how to make this idea fully work.

3.2.2 Preliminary: Differentially Private Prefix Sum

Dwork et al. [17] 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. However, we first state the result by Chan et al. [11, 12] (whose stated parameters slightly tighter than Dwork et al. [17]).

Theorem 3.4 (Differentially private prefix sum [11, 12])**.** *For any ϵ, there exists an ϵ-differentially private algorithm, such that given a stream in $\mathbb{Z}_+^N$ (where neigboring streams have ℓ_e-norm at most 1), the algorithm outputs the vector of all N prefix sums, such that*

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

3.2.3 Algorithm for Sorting 1-Bit Keys

It suffices to derive a *tight stable compaction* algorithm: tight stable compaction outputs an array containing only the 1-balls 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. Since given a tight stable compaction algorithm running in time $t(N)$, we can easily realize a stable 1-bit 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 the array — let X be the resulting array;
2. Run tight stable compaction to stably move all 1-balls to the end of the 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.

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

We thus focus on describing a tight stable compaction algorithm that stably compacts an input array I given a privacy parameter ϵ and a batch size s.

TightStableCompact(I, ϵ, s):

- Invoke an instance of the differentially private prefix sum algorithm with the privacy budget ϵ to estimate for every $i \in [N]$, the total number of 1-balls in the input stream I up till position i — henceforth we use the notation $\widetilde{Y}_i$ to denote the i-th prefix sum estimated by the differentially private prefix sum algorithm.
- Imagine there is a working buffer initialized to be empty. We now repeat the following until there are no more bins left in the input.
  1. Fetch the next s balls from the input stream into the working buffer.
  2. Obliviously sort the working buffer such that all 1-balls are moved to the front, and all 0-balls moved to the end; we use the ball's index in the input array to break ties for stability.

3. Suppose that k balls from the input have been operated on so far. If there are fewer than $\widetilde{Y}_k - s$ balls in the output array, pop the head of the working buffer and append to the output array until there are $\widetilde{Y}_k - s$ balls in the output array.
4. If the working buffer (after popping) is longer than $2s$, truncate from the end such that the working buffer is of size $2s$.

- Finally, at the end, if the output is shorter than N, then obliviously sort the working buffer (using the same relative ordering function as before) and write an appropriate number of balls from the head into the output such that the output buffer is of length N.

Theorem 3.5 (Tight stable compaction)**.** *For any $\epsilon, \delta > 0$, for any input array I containing N elements, let $s = \frac{1}{\epsilon} \cdot \log^{1.5} N \cdot \log \frac{1}{\delta}$, then the algorithm $\mathsf{TightStableCompact}(I, \epsilon, s)$ satisfies $(\epsilon, 0)$-DO and with $1 - \delta$ probability, produces a correct outcome. 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 $(\epsilon, 0)$-DO, completes in $O(N \log \log N)$ runtime, and with $1 - \mathsf{negl}(N)$ probability, produces a correct outcome where $\mathsf{negl}(\cdot)$ is some negligible function.*

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 ϵ-differential privacy. This follows in a straightforward manner from Theorem 3.4. Correctness of the algorithm is guaranteed as long as no prefix sum has more than s additive error, thus the correctness statement also follows from Theorem 3.4. 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 3.6 (Stable 1-bit sorting)**.** *For any $\epsilon > 0$ and any $0 < \delta < 1$, there exists an $(\epsilon, 0)$-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 correctly except with δ probability. As a special case, for $\epsilon = \Theta(1)$, there exists an $(\epsilon, 0)$-differentially oblivious stable 1-bit sorting algorithm such that completes in $O(N \log \log N)$ runtime and errs only with probability $\mathsf{negl}(N)$ for some negligible function $\mathsf{negl}(\cdot)$.*

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 3.5. □

Optimality. In light of our lower bound to be presented in the next section (Theorem 4.7), our 1-bit stable sorting algorithm is in fact optimal (up to log log factors) as long as $\epsilon s \geq 2 \log^2 N$ — note that this includes most parameter ranges one might care about. For the special case of $\epsilon = \Theta(1)$, our upper bound is $\widetilde{O}(N)$ runtime for $\delta = e^{-\mathsf{poly} \log N}$ 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.

3.3 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 by adopting Radix Sort where we sort the input bit by bit starting from the lowest-order bit. Clearly, if the stable 1-bit sorting building block satisfies (ϵ, δ)-differentially oblivious, then resulting k-bit stable sorting algorithm satisfies $(k\epsilon, k\delta)$-differentially oblivious. This gives rise to the following corollary.

Corollary 3.7 (Stable k-bit sorting). *For any $\epsilon, \delta > 0$, there exists an $(\epsilon, 0)$-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 correctly except with $k\delta$ probability.*

As a special case, for $\epsilon = \Theta(1)$, there exists an $(\epsilon, 0)$-differentially oblivious stable k-bit sorting algorithm that completes in $O(kN \log\log N)$ runtime and errs only with probability $\mathsf{negl}(N)$ for some negligible function $\mathsf{negl}(\cdot)$.

We point out that if $k = o(\log N / \log\log N)$, we obtain a stable k-bit sorting algorithm that overcomes the $\Omega(N \log N)$ barrier for stable δ-oblivious sort in the balls-and-bins model — recall that Lin, Shi, and Xie [35] show that for even $\delta = O(1)$, any (possibly non-comparison-based) stable 1-bit δ-oblivious sorting algorithm in the balls-and-bins model must incur $\Omega(N \log N)$ runtime. We stress that our algorithm is non-comparison-based, since otherwise due to the 0-1 principle, any comparison-based sorting algorithm — even without privacy requirements and even for 1-bit keys — must incur at least $\Omega(N \log N)$ runtime.

4 Limits of Differentially Oblivious Sorting

Earlier, we showed that for a suitable, negligibly small δ and $\epsilon = \Theta(1)$, by adopting the weaker notion of (ϵ, δ)-differential obliviousness. we can overcome the $\Omega(N \log N)$ barrier for oblivious stable sorting for small keys (in the balls-and-bins model). In this section, we show that if δ must be subexponentially small (including the special case of requiring $\delta = 0$), then (ϵ, δ)-differentially oblivious 1-bit stable sorting would suffer from the same lower bound as the oblivious case (despite the relaxation in definition).

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 simply be simulated by a trivial ORAM with $O(1)$ blowup.

4.1 Warmup and Intuition

As a warmup, we consider a simple lower bound proof for the case $\delta = 0$ and for general sorting (where the input can contain arbitrary keys not just 1-bit keys). Suppose there is some ϵ-differentially oblivious balls-and-bins sorting algorithm denoted sort. Now, given a specific input array I, let G be such a compact graph encountered with non-zero probability p. By the requirement of ϵ-differential obliviousness, it must be that for any input array I', the probability of encountering G must be at least $p \cdot e^{-\epsilon N} > 0$. This means G must also be able to explain any other input array I'. In other words, for any input I' there must exist a feasible method for routing the balls contained in the input I' to their correct location in the output locations in G. Recall that in the compact graph G, every node (i, t) can receive a ball from either of its two incoming edges: either from the parent (i, t') for some $t' < t$, from the parent $(CPU, t - 1)$. 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 $\delta \neq 0$. In the remainder of this section, we will prove such a lower bound for $\delta \neq 0$ — our proofs draw an interesting connection to the study of routing graph complexity in the classical algorithms literature [41]. Specifically, we model balls-and-bins sorting as routing multiple commodities to destinations over node-disjoint paths, and we study the complexity of such a routing graph given that a single routing graph must satisfy many different input-output assignments.

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 stable sorting in a blackbox manner: every ball's 1-bit key is appended with its index in the input array to break ties, and then we simply sort this array. Clearly, if the general sorting algorithm attains (ϵ, δ)-differential obliviousness, so does the resulting 1-bit stable sorting algorithm. Thus, a lower bound for 1-bit stable sorting is stronger than a lower bound for general sorting (parameters being equal). To derive our lower bound proofs, we must first introduce several new definitions and preliminaries.

4.2 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. This applies to the various notions of neighboring defined in Section 6.2.1.

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

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

Proof. The proof is deferred to appendices A.1. □

4.3 Access Pattern Graphs under the Balls-and-Bins Model

Recall that we assume a balls-and-bins model and as argued, 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 ℓ in the i-th access. Then, we add two directed edges $((i-1, CPU), (i, \ell))$ and $((i-1, \ell), (i, CPU))$. This reflects the balls stored in the CPU and location ℓ can possibly move between those two places.

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

Lemma 4.4 (Number of Edges in 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 appendices A.1. □

4.4 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.

Pippenger and Valiant proved the following useful result [41].

Fact 4.5 (Pippenger and Valiant [41])**.** *Let $\mathbf{A} := (A_1, A_2, \ldots, A_s)$ denote a set of assignments from I to O where $n = |I| \geq |O|$, such that each input in I is assigned to s different outputs in O by the s assignments in $\mathbf{A}$. Let G be a graph that implements every A_i for $i \in [s]$. It holds that 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.

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.

4.5 Lower Bounds for Differentially Oblivious Sorting

We prove a lower bound for differentially oblivious, 1-bit stable sorting in the balls-and-bins model. Interestingly, our lower bound (almost) tightly matches our earlier upper bound for 1-bit stable sorting for a wide parameter range.

Theorem 4.7 (Limits of differentially oblivious 1-bit stable sorting)**.** *Let $0 < s \leq \sqrt{N}$ be an integer. Suppose $\epsilon > 0$ and $0 \leq \delta \leq e^{-(2\epsilon s + \log^2 N)}$. Then, any (randomized) stable 1-bit sorting algorithm (in the balls-and-bins model) that is (ϵ, δ)-DO must have some input, on which it incurs at least $\Omega(N \log s)$ memory accesses with probability at least $1 - \mathsf{negl}(N)$ for some negligible function $\mathsf{negl}(\cdot)$.*

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

For each $0 \leq i \leq s$, we define the input scenario I_i as follows, such that in each scenario, there are exactly s elements with key value 0 and $N - s$ elements with key value 1. Specifically, in scenario I_i, the first $s - i$ and the last i elements in $\mathsf{Input}[0..N-1]$ have key value 0, while all other elements have key value 1. It can be checked that any two scenarios are $2s$-neighboring.

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

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

Therefore, by Lemma 4.3, with probability at least $1 - \eta$ (where $\eta := s \cdot \frac{e^{\epsilon \cdot 2s} - 1}{e^{\epsilon} - 1} \cdot \delta = \mathsf{negl}(N)$), 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 4.3) 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 $\mathsf{Input}[j]$ to the node corresponding to the final memory location $\mathsf{Output}[j + i]$.

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 stable sorting easily by using the index as low-order tie-breaking bits. Thus our lower bound for stable 1-bit 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 $\epsilon > 0$ and $0 \leq \delta \leq e^{-(2\epsilon s + \log^2 N)}$. Then, any (randomized) sorting algorithm that is (ϵ, δ)-DO must have some input, on which it incurs at least $\Omega(N \log s)$ memory accesses with probability at least $1 - \mathsf{negl}(N)$ for some negligible function $\mathsf{negl}(\cdot)$.*

5 Merging Two Sorted Lists

Merging in the balls-and-bins model is the following abstraction: given two input arrays each of which contains N balls sorted by their tagged keys, merge them into a single sorted array. Pippenger and Valiant [41] showed that any oblivious merging algorithm in the balls-and-bins model must incur at least $\Omega(N \log N)$ movements of balls.

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

5.1 Defining Differentially Oblivious 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 define two arrays I and

I' to be neighboring iff they are of the same length and there is exactly one element in I but not in I' and vice versa. We say that two inputs (I_0, I_1) and (I'_0, I'_1) are neighboring if for either $b = 0$ or $b = 1$, I_b and I'_b are neighboring and $I_{1-b} = I'_{1-b}$. Based on this notion of neighboring, we can define (ϵ, δ)-differentially oblivious merging in the same manner as we defined (ϵ, δ)-differentially sorting (see Section 3.1).

5.2 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. Inspired by Bun et al. [8], we can divide each sorted input array into $\mathsf{poly}\log\lambda$-sized bins (where λ is the security parameter). In order to help our merging algorithm to 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 between the minimum and the maximum (inclusively) elements of the bin.

However, we would like to *localize* the influence of one change in the input, e.g., inserting one one small element at the beginning of an input array should not affect the contents of all bins. We use an idea in Bun et al. [8], where a random number of real elements are put in each bin, which is padded with dummies to its maximum capacity $Z = \mathsf{poly}\log\lambda$. Hence, one change in the input can be masqueraded by the random noise.

Challenges of implementing noisy-boundary binning. For privacy, it is critical that each bin's actual load cannot be revealed to the adversary. In the case of Bun et al. [8], implementing the noisy-boundary binning idea is simple because in their setting the mechanism is run by a trusted curator.

One immediate 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 [35] and thus would be too expensive.

Flawed batching. The next idea is to rely on a buffer. We could, each time, read in a small batch of $\frac{mZ}{2}$ number of elements from the input stream into a working buffer, and write out $m - 2$ bins by executing oblivious bin placement on this small batch. As described in the appendices, such oblivious bin placement for elements in a batch only needs to use oblivious sorting on instances with size $\mathsf{poly}\log\lambda$, thereby reducing the total running time.

However, the issue with this approach is that very soon the working buffer will accumulate too many elements, leading to a *queuing excess* problem. The reason is that to hide the real loads of bins with all but negligible probability, the rate of elements entering the buffer needs to be slightly larger than that of elements leaving.

Defeating queuing excess with differentially private prefix sum. Suppose we somehow know that the number of real elements in the first i bins is in the range $[C_i, C'_i]$. Then, to output the first i bins, it suffices to read the input stream up till position C'_i. After the first i bins have been emitted (through oblivious bin placement), there are at most $C'_i - C_i$ elements left in the working buffer, and thus the working buffer will never be too large.

Fortunately, we have differentially private mechanisms for computing prefix sums [11, 12, 17] that can be used to provide an estimate that is accurate enough. At first sight, it would seem like releasing differentially private prefix sum of bin-loads is defeating the effects of noisy boundary binning, since the prefix sums release some information about the random bin-loads. However, in our appendices, we will formally prove a binning composition theorem, showing that with our noisy-boundary binning, it is safe to release any statistic that is differentially private w.r.t. to the binning outcome — the resulting statistic would actually be differentially private w.r.t. the original input too.

Putting it together: creating thresh-bins. Putting all of 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. Hence, a similar queuing excess problem arises, but we can rely on the differentially private prefix sum trick again to tackle such queuing excess and make sure that at any time, the number of elements remaining in the working buffer is small.

5.3 Preliminaries

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

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

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

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

5.4 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 naïve manners. 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, 19, 20, 23, 40]. 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.

In the following theorem, for convenience, we assume a RAM where computations on arbitrary-precision real numbers and sampling from an exponential distribution have unit cost — in the appendices, we show how to relax this assumption. We also present the proofs in the appendices.

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 $n \geq \frac{18500}{\epsilon} \cdot 2^{\log^* U} \cdot \log^* U \cdot \ln \frac{4 \log^* U}{\beta \epsilon \delta}$, the algorithm*

- *completes consuming only $O(Z \log Z)$ time and number of memory accesses.*
- *the algorithm produces an outcome that is (ϵ, δ)-differentially private;*
- *with probability at least $1 - \beta$, the outcome is an interior point of the input bin; 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.*

5.5 Subroutine: Creating Thresh-Bins

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

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

Assume:

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

Algorithm:

- Recall that the elements in X are sorted; if the length of the input X is too small, append an appropriate number of elements with key ∞ at the end such that it has length at least $\frac{BZ}{2}(1 + \frac{1}{\log^2 \lambda})$.

This makes sure that with overwhelming probability, the real elements in the input stream do not deplete prematurely in the algorithm below.

- For $i = 1$ to B, let $R_i = \mathsf{Geom}^Z(\exp(\epsilon_0))$ be independently sampled truncated geometric random variables. Denote the vector $R := (R_1, R_2, \ldots, R_B)$.

  Call $D := \mathsf{PrefixSum}(\lambda, R, \frac{\epsilon_0}{4}) \in Z_+^B$, which is the $\frac{\epsilon_0}{4}$-differentially private subroutine in Theorem 3.4 that privately estimates prefix sums, each of which has additive error at most s with all but $\exp(-\Theta(\log^2 \lambda))$ probability.

- Let $k := \frac{B}{m}$. Define the vector $C \in \mathbb{Z}_+^k$, where $C[j] := D[jm]$, for $1 \leq j \leq k$. We use the convention $C[0] := 0$.

- Let Buf be a buffer with capacity $\frac{mZ}{2}(1 + \frac{1}{\log^2 \lambda}) + 2s = O(mZ)$. Initially, we place the first s elements of X in Buf.

- For $i = 1$ to k:

  - Read the next batch of elements from the input stream X with indices from $C[i-1] + s + 1$ to $C[i] + s$, and add these elements to the buffer Buf.

    This can be 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.

  - Make a copy of Buf denoted Buf', for every element in Buf after position $R^* := \sum_{j=(i-1)m+1}^{im} R_j$, mark it as dummy in Buf'.

  - Now, call $\mathsf{ObliviousBinPlace}(\mathsf{Buf}', (R_{(i-1)m+1} .. R_{im}), Z)$ to place elements in Buf' into the next m bins where each bin is padded with dummies to the maximum capacity Z.

    Moreover, we use the $(\frac{\epsilon_0}{4}, \delta)$-differentially oblivious interior point mechanism in Section 5.4 to tag each bin with an interior point, denoted by a vector $P = (P_1, \ldots, P_B)$, where $\delta := \frac{1}{4}\exp(-0.1 \log^2 \lambda)$; we also tag each bin with its estimated prefix sum from vector D.

    Append the m bins to the output list T.

  - Mark every element in Buf at position R^* or smaller as dummy.

5.6 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.

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

Assume:

1. The input is T_0 and T_1, each of which is a list of thresh-bins, where each bin has capacity $Z = \frac{1}{\epsilon_0} \log^8 \lambda$ size and $B := |T_0| + |T_1|$ 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 $M := \lceil \frac{Z}{2} \cdot (|T_0| + |T_1|)(1 + \frac{1}{\log^2 \lambda}) \rceil$.

Algorithm:

- Let $m = \log^2 \lambda$, $s = \frac{1}{\epsilon_0} \log^3 \lambda$.
- Initialize an empty array $\mathsf{Output}[0..M-1]$ of length $M := \lceil \frac{Z}{2} \cdot (|T_0| + |T_1|)(1 + \frac{1}{\log^2 \lambda}) \rceil$.

  Initialize $\mathsf{count} := 0$, the number of elements already delivered to Output.
- Initialize an empty buffer Buf with capacity $K := \lceil \frac{(m+10)Z}{2}(1 + \frac{1}{\log^2 \lambda}) + 4s \rceil = O(mZ)$.
- 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.)

  This is the order in which the bins are fetched and processed. Let $B := |T_0| + |T_1|$, the total number of bins.
- For $i = 1$ to $\lceil \frac{B}{m} \rceil$:
  - Let W be the set of the next $m + 2$ bins from the beginning of the list $\mathcal{L}$. Do not remove any bins from the list yet.
  - Denote j_0 as the largest index such that the bin $T_0[j_0]$ is in W; if there is no such index, set $j_0 := 0$. Define j_1 similarly for T_1. (Note that we use the convention that the first bin in T_0 or $T - 1$ has index 1.)

    Let $\widehat{W} := W \cup \{T_0[j_0 + 1], T_1[j_1 + 1]\}$.

    For every bin in $\widehat{W}$ that has not been inserted into **Buf** before, insert it into **Buf**. This can be done by appending the eligible bins in $\widehat{W}$ at the end of **Buf** to temporarily increase the size of **Buf**. Then, oblivious sorting followed by truncation can be used to restore its capacity.
  - Define **safe** bins: For $b \in \{0, 1\}$ and some index k, a bin $T_b[k]$ that has been inserted into **Buf** is called *safe*, if there exists some bin from T_{1-b} that has been inserted into **Buf** and whose interior point is at least that of $T_b[k + 1]$. (Observe that any element with key smaller than that of an element in a safe bin has already been put into the buffer.)

    Let S be the set of safe bins. Remove any bin in S from the list $\mathcal{L}$.
  - Define k_0 to be the largest index such that $T_0[k_0]$ is a safe bin; set $k_0 := 0$ if there is no such index. Define k_1 similarly for T_1.
  - Set $\mathsf{newcount} := D_0[k_0] + D_1[k_1] - 2s$.
  - Remove the first $(\mathsf{newcount} - \mathsf{count})$ elements from the Buf and copy them into the next available slots in the Output array.

    Update $\mathsf{count} \leftarrow \mathsf{newcount}$.
- Oblivious sort **Buf** to copy any remaining elements into any available slots left in Output.

5.7 Full Merging Algorithm

Finally, the full merging algorithm involves taking the two input arrays, creating thresh-bins out of them using $\mathsf{ThreshBins}$, and then calling Merge to merge the two lists of thresh-bins. We defer concrete parameters of the full scheme and proofs to the appendices.

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

Assume:

1. The input is two sorted arrays I_0 and I_1.

2. We suppose that $\epsilon < c$ for some constant c, $\log^* U \leq \log\log\lambda$, and $|I_0| \leq \mathsf{poly}_0(\lambda)$ and $|I_1| \leq \mathsf{poly}_1(\lambda)$ for some fixed polynomials $\mathsf{poly}_0(\cdot)$ and $\mathsf{poly}_1(\cdot)$.

Algorithm:

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

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

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

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

5.8 Lower Bounds for Differentially Oblivious Merging

Theorem 5.4. *Consider the merging problem, in which the input is two sorted lists of elements and the output is the merging of the two input lists into a single sorted list.*

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

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

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

By Lemma 4.3, on input scenario I_0, any merging algorithm that is (ϵ, δ)-DO produces an access pattern A that is plausible for all I_i's $(1 \leq i \leq s)$ with all but probability of $s \cdot \frac{e^{\epsilon s}-1}{e^{\epsilon}-1} \cdot \delta = \mathsf{negl}(N)$.

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

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

Hence, Lemma 4.5 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. □

6 Differentially Oblivious Range Query Data Structure

6.1 Data Structures

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

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

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

Correctness notion under obfuscated lengths. As Kellaris et al. [29] 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 [4, 9] and Oblivious RAM constructions [3]. 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 ops_0 and ops_1 (consisting of insertions and queries) are *query-consistent neighboring*, if the two sequences differ in exactly position i, and moreover both $\mathsf{ops}_0[i]$ and $\mathsf{ops}_1[i]$ 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 λ. We say that a data structure scheme $\mathbb{DS}$ preserves static (ϵ, δ)-differential obliviousness, if for any two query-consistent neighboring operational sequences ops_0 and ops_1, for any λ, for any set S of access patterns,

$$\Pr[\mathbf{Accesses}^{\mathbb{DS}}(\lambda, \mathsf{ops}_0) \in S] \leq e^{\epsilon(\lambda)} \cdot \Pr[\mathbf{Accesses}^{\mathbb{DS}}(\lambda, \mathsf{ops}_1) \in S] + \delta(\lambda) \tag{1}$$

where the random variable $\mathbf{Accesses}^{\mathbb{DS}}(\lambda, \mathsf{ops})$ denotes the access patterns incurred by the data structure upon receiving the security parameter λ and operational sequence 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 $\mathbb{DS}$ satisfies adaptive (ϵ, δ)-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 $\mathsf{Expt}^0_{\mathcal{A}}(\lambda, N)$ and $\mathsf{Expt}^1_{\mathcal{A}}(\lambda, N)$ satisfy the following equation:

$$\Pr[\mathsf{Expt}^0_{\mathcal{A}}(\lambda, N) = 1] \leq e^{\epsilon(\lambda)} \cdot \Pr[\mathsf{Expt}^1_{\mathcal{A}}(\lambda, N) = 1] + \delta(\lambda)$$

```
$\mathsf{Expt}^b_{\mathcal{A}}(\lambda, N)$:
    $\mathsf{addresses}_0 = \bot$
    For $t = 1, 2, 3, \ldots, N$:   $(\mathsf{op}^0_t, \mathsf{op}^1_t) \leftarrow \mathcal{A}(N, \mathsf{addresses}_{t-1})$, $\mathsf{addresses}_t \leftarrow \mathbb{DS}(\lambda, \mathsf{op}^b_t)$
    $b \leftarrow \mathcal{A}$, and output $b$
```

In the above, $\mathsf{addresses}_t$ denotes the ordered sequence of physical memory locations accessed for the t-th operation 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. $\mathbb{DS}$ iff for every λ and every N, for either $b \in \{0,1\}$, with probability 1 in the above experiment $\mathsf{Expt}^b_{\mathcal{A}}(\lambda, N)$, $\mathcal{A}$ outputs $\mathsf{op}^0_t = \mathsf{op}^1_t$ for all but one time step $t \in [N]$; and moreover for this differing time step t, op^0_t and op^1_t 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 $\mathsf{Query}(T, [s,t])$. Let $T := \{\mathsf{Bin}_i\}_{i \in [B]}$ 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 $M_i \le s \le t \le M_j$ — for example, this can be accomplished through binary search. To handle boundary conditions, we may simply assume that there is an imaginery bin before the first bin with the interior point $-\infty$ and there is an imaginery bin at the end with the interior point ∞.

2. Now, read all bins $\mathsf{Bin}_i, \mathsf{Bin}_{i+1}, \ldots \mathsf{Bin}_j$ 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 [19,20,23] — however, in hierarchical ORAMs [19, 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 $\mathsf{MergeThreshBins}$ procedure described earlier.

We now describe our data structure construction supporting insertions and range queries.

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.
- 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, parametrized by ϵ. To insert some record, enter it into Buf and if Buf now contains Z elements, we use $\widetilde{T_0} := \mathsf{ThreshBins}(\lambda, \mathsf{Buf}, 4, \epsilon)$ 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 $T_i := \widetilde{T_i}$ and return (i.e., terminate the procedure).
- Else call $Y := \mathsf{MergeThreshBins}(\lambda, T_i, \widetilde{T_i}, \epsilon)$; and let $\widetilde{T_{i+1}} = \mathsf{ThreshBins}(\lambda, Y, 4 \cdot 2^{i+1}, \epsilon)$, let $i \leftarrow i+1$ and repeat if $i \le L$.

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 $\mathsf{Query}(T_i,[s,t])$. 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 $\epsilon = O(1)$, 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.*

Proof. The proof follows in a straightforward manner by adaptive $\log N$-fold composition of differential privacy [18], by observing that every element is involved in only $\log N$ instances of ThreshBins and MergeThreshBins. For adaptive security, notice that the adaptive composition theorem works for adaptively generated database entries as well as adaptive queries [18]. □

Theorem 6.4 (Performance)**.** *Let $N = \mathsf{poly}(\lambda)$ be the total number of insertion operations over time where $\mathsf{poly}(\cdot)$ 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;*
- *Each range query requires reading only $O(\log N)$ discontiguous memory regions, 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 $\widetilde{T}_i$ contain $Z \cdot 2^i$ real elements and every $2^i/Z$ operations, we must merge T_i and $\widetilde{T}_i$ once incurring $Z2^i \log\log\lambda$ 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 $N = \mathsf{poly}(\lambda)$. The runtime and locality claims for each range query follow in a straighforward 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 $O(\log N)$. □

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) [24, 43], 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 *incomparable* 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-interactiveness, and non-trivial efficiency, even for the designated-client setting. One interesting point of comparison is ORAMs [46, 50] and oblivious data structures [51] 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. [3] 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 δ, then any (ϵ, δ)-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 multiplicty 1 and upon query only 1 record is returned; in the latter, each key has more general multiplicity.

Theorem 6.5 (Limits of (ϵ, δ)-DO: Distinct Keys)**.** *Suppose that* $N = \mathsf{poly}(\lambda)$ *for some fixed polynomial* $\mathsf{poly}(\cdot)$ *and* $0 < s \leq \sqrt{N}$ *are integers. Let* $\epsilon > 0$ *and* $0 \leq \delta \leq e^{-(2\epsilon s + \log^2 N)}$*. Suppose that* $\mathbb{DS}$ *is a perfectly correct,* (ϵ, δ)*-DO 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* $\mathbb{DS}$ *makes for serving this sequence is* $\Omega(N \log s)$ *with probability at least* $1 - \mathsf{negl}(N)$ *for some negligible function* $\mathsf{negl}(\cdot)$*.*

Proof. Define $T := \lfloor \frac{N}{s} \rfloor$. 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 $1 - \mathsf{negl}(N)$; 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 4.3, with probability at least $1 - \eta$ (where $\eta := s \cdot \frac{e^{\epsilon \cdot s} - 1}{e^{\epsilon} - 1} \cdot \delta = \mathsf{negl}(N)$), 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 4.3) 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 $\mathsf{Input}[0..s-1]$ and the s queries correspond to some memory locations $\mathsf{Output}[0..s-1]$, where $\mathsf{Output}[k]$ 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 $\mathsf{Input}[k]$ is supposed to be returned at $\mathsf{Output}[k + j]$ (where addition $j + i$ is performed modulo s) during qeury.

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

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. □

Theorem 6.6 (Limits of (ϵ, δ)-DO: Repeated Keys)**.** *Suppose that $N = \mathsf{poly}(\lambda)$ for some fixed polynomial $\mathsf{poly}(\cdot)$ and fix some integer $r = \mathsf{poly}\log(N)$. Let $r < s \leq \sqrt{N}$ such that r divides s, $\epsilon > 0$ and $0 \leq \delta \leq e^{-(2\epsilon s + \log^2 N)}$. Suppose that $\mathbb{DS}$ is a perfectly correct, (ϵ, δ)-DO data structure supporting point queries.*

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

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 $X_i := \{\frac{s}{r} \cdot (i-1) + j : 0 \leq j < \frac{s}{r}\}$, where each distinct is inserted r times in a batch. The order in which the distinct keys in

X_i are inserted is private. In this lower bound, we consider $\frac{s}{r}$ different cyclic shifts of the $\frac{s}{r}$ batches.

2. s 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, it suffices to show that epoch i requires $\Omega(s \log \frac{s}{r})$ accesses with all but $\mathsf{negl}(N)$ probability. The proof uses the same technique of routing graphs, except that there are only $\frac{s}{r}$ input scenarios, each of which correspond to a bijection from some $\mathsf{Input}[0..s-1]$ memory locations to some $\mathsf{Output}[0..\mathsf{s}-1]$ locations; moreover, each input location is mapped to $\frac{s}{r}$ distinct output locations by the $\frac{s}{r}$ bijections. Hence, it follows that with all but $\mathsf{negl}(N)$ probablity, epoch i takes $\Omega(s \log \frac{s}{r})$ memory accesses, as required. □

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 [41]. We thank Kobbi Nissim, George Kellaris, and Rafael Pass for helpful discussions and suggestions. We thank Kartik Nayak 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

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

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

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

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

[5] E. Boyle, K. Chung, and R. Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *CRYPTO*, 2015.

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

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

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

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

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

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

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

[13] T.-H. H. Chan, E. Shi, and D. Song. Privacy-preserving stream aggregation with fault tolerance. In *FC*, 2012.

[14] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *Asiacrypt*, 2014.

[15] T. Dalenius. Towards a methodology for statistical disclosure control. *Statistik Tidskrift*, 15(429-444):2–1, 1977.

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

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

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

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

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

[21] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14.

[22] M. T. Goodrich, D. S. Hirschberg, M. Mitzenmacher, and J. Thaler. Cache-oblivious dictionaries and multimaps with negligible failure probability. In *Design and Analysis of Algorithms - MedAlg*, 2012.

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

[24] S. Gorbunov, V. Vaikuntanathan, and H. Wee. Attribute-based encryption for circuits. *In submission to STOC 2013.*

[25] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

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

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

[28] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

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

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

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

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

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

[34] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

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

[36] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

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

[38] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

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

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

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

[42] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[43] A. Sahai and B. Waters. Fuzzy identity-based encryption. In *EUROCRYPT*, 2005.

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

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

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

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

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

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

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

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

[52] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[53] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S & P*, 2015.

A Extra Preliminaries

A.1 Technical Lemmas concerning r-Neighbors

Fact A.1 (r-Neighbors Produce Similar Access Patterns [48]). *Suppose a (randomized) algorithm M satisfies (ϵ, δ)-differential obliviousness, where ϵ and δ can depend on some security parameter λ. Then, for any two inputs I and I' that are r-neighboring and any set S of access patterns, we have*

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

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

By Fact A.1, we have

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

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

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

Finally, observe that $\mathbf{Accesses}^M(\lambda, I_0) \in \cup_{I_i \in \mathcal{C}} S_i$ is the complement of the event that $\mathbf{Accesses}^M(\lambda, I_0)$ is plausible for all inputs in $\mathcal{C}$. 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. □

A.2 Stochastic Analysis for Geometric Distribution

The following simple fact follows in a straightforward manner from the definition of Geom.

Fact A.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*

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

Lemma A.3 (Moment Generating Function). *Suppose G is a random variable having truncated geometric distribution $\mathsf{Geom}^Z(\alpha)$ 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*

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

Proof. Let V be a random variable whose distribution the untruncated variant of $\mathsf{Geom}(\alpha)$ 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 $\mathsf{Geom}(\alpha)$ 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, $E[e^{tG}] \leq e^{\frac{Z}{2}\cdot t} \cdot E[e^{tV}]$. Finally, the result follows from applying a technical result from [13, Lemma 1, Appendix B.1] to get an upper bound on $E[e^{tV}]$ the specified range of t. □

Lemma A.4 (Measure concentration for truncated geometric random variables)**.** *Let G_B denote the sum of B independent $\mathsf{Geom}^Z(e^{\epsilon_0})$ random variables (each of which having mean $\frac{Z}{2}$ and support $[0..Z]$). For any B, for sufficiently large λ and $Z \geq \frac{\log^5 \lambda}{\epsilon_0}$, 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 A.3, we have

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

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 ϵ_0.

The first case is when ϵ_0 is some large enough constant. In this case, we set $t > 0$ to be some appropriate constant, and the exponent is $-\Theta(R) = -\Theta(\frac{Z}{2\log^2 \lambda}) \leq -\log^2 \lambda$.

The second case is when ϵ_0 is small. In this case, $\frac{4\alpha}{(\alpha-1)^2} = \Theta(\frac{1}{\epsilon_0})^2$. We set t to $\sqrt{2 \ln \frac{(\alpha+1)^2}{4\alpha}} = \Theta(\epsilon_0)$. Hence, the exponent is $-\Theta(R\epsilon_0) = -\Theta(\frac{Z\epsilon_0}{2\log^2 \lambda}) \leq -\log^2 \lambda$. This completes the proof of the first inequality. □

B Analysis of the Merging Algorithm

This section is devoted to prove Theorem 5.3. We start by introducing the notion of oblivious realization of an ideal functionalities with (differentially private) leakage.

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

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

- *Ideal execution:* choose all random bits ρ needed by $\mathcal{F}$, and let $O_{\text{ideal}} \leftarrow \mathcal{F}(\lambda, I, \rho)$, let $L_{\text{ideal}} \leftarrow \mathcal{L}(\lambda, I, \rho)$ — note that the leakage function $\mathcal{L}$ also obtains the same randomness as $\mathcal{F}$.
- *Real execution:* let $(O_{\text{real}}, L_{\text{real}}, \mathsf{addresses}) \leftarrow \mathsf{Alg}(\lambda, I)$.

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

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

Definition B.2. We say that the leakage function $\mathcal{L}$ is (ϵ, δ)-differentially private w.r.t. the input iff for every λ, for every neighboring I and I' and every set S, it holds that

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

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

Definition B.3. Consider some special leakage function $\mathcal{L}$ that is fully determined by the output of $\mathcal{F}$, i.e., $\mathcal{L}(\lambda, I, \rho) := \mathcal{L}(\lambda, T)$ where $T := \mathcal{F}(\lambda, I, \rho)$. We say that $\mathcal{L}$ is (ϵ, δ)-differentially private w.r.t. the output of $\mathcal{F}$ (or (ϵ, δ)-differentially private w.r.t. T), iff for every λ, 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 in the probability comes from the random coins of $\mathcal{L}$.

The following fact is immediate from the definition.

Fact B.4. *If some algorithm* Alg *obliviously realizes some functionality $\mathcal{F}$ with leakage $\mathcal{L}$ where $\mathcal{L}$ is (ϵ, δ)-differentially private w.r.t. the input, then* Alg *satisfies (ϵ, δ)-differential obliviousness.*

B.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 ∞), a target bin number B, and a parameter ϵ_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 ∞ 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 $R_i \leftarrow_{\$} \mathsf{Geom}^Z(\exp(\epsilon_0))$.
  - 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 $D := \mathsf{PrefixSum}(\lambda, C, \frac{\epsilon_0}{4}) \in Z_+^B$, which is the $\frac{\epsilon_0}{4}$-differentially private subroutine in Theorem 3.4 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 5.4 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 lekage 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 and differ by only one element.

Lemma B.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 (ϵ, δ)-differentially private with respect to the output T, then $\mathcal{L}$ is $(2\epsilon_0 + 4\epsilon, \delta_{\mathsf{bad}} + 4\delta)$-differentially private with respect to X, where $\delta_{\mathsf{bad}} \leq O(\exp(-\Theta(\log^2 \lambda)))$.*

To prove Lemma B.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 $\mathsf{good} \cup \mathsf{bad}$, where $\mathsf{good} = \{R : \sum_{i=1}^{B-1} R_i \geq |X|\}$ and $\mathsf{bad} = [Z]^B \backslash \mathsf{good}$. Let δ_{bad} be the probability that R is in bad when $R \leftarrow_{\$} (\mathsf{Geom}^Z(\exp(\epsilon_0)))^B$. By Lemma A.4, $\delta_{\mathsf{bad}} \leq \exp(-\log^2 \lambda)$.

We need the following technical lemma about the ideal thresh-bins functionality.

Lemma B.6. *Consider two neighboring input arrays X^0, X^1 and parameter B such that $B \geq \lceil \frac{2|X_0|}{Z} \cdot (1 + \frac{2}{\log^2 \lambda}) \rceil$. There exists an* injective *function $f : \mathsf{good} \to [Z]^B$ such that the following holds. For every $R^0 \in \mathsf{good}$, 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 $(\mathsf{Geom}^Z(\exp(\epsilon_0)))^B$, 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 denote $X^0 \backslash \{x^0\}$. 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 ℓ 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 $(\ell+1)$-th coordinate is decrease by 1, i.e., $R'^0_{\ell+1} = R^0_{\ell+1} - 1$ and $R'^0_i = R^0_i$ for all $i \neq \ell + 1$.
- We then define f^1, which takes input R'^0. Let ℓ' denote the bin in $T(X^0, R'^0)$ that x^1 should be inserted in. We define $f^1(R'^0) = R^1$ where R^1 is identical to R'^0 except that with the $(\ell'+1)$-th coordinate is increased by 1, i.e., $R^1_{\ell'+1} = R'^0_{\ell'+1} + 1$ and $R^1_i = R'^0_i$ 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 ℓ 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 coordinate by 1. For property (ii), observe that $T(X^0, R^0)$ and

$T(X'^0, R'^0)$ can only differ in the ℓ-th and $\ell+1$-th 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 $T(X^1, R^1)$ 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 B.5.

Proof. 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 $T^b \leftarrow \mathcal{F}_{\text{threshbins}}(\lambda, X^b, B, \epsilon_0)$, $L^b =\leftarrow \mathcal{L}(\lambda, T^b, \epsilon_0)$, 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

$$\Pr[L^0 \in S] \leq e^{2\epsilon_0 + 4\epsilon} \Pr[L^1 \in S] + \delta_{\mathsf{bad}} + 4\delta$$

This is proved by the following calculation, where the function f is from Lemma B.6.

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

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 $\mathcal{L}(\lambda, T^b, \epsilon_0)$ is (ϵ, δ)-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 $R^0 \in \mathsf{good}$. Both properties are from Lemma B.6. □

B.3 ThreshBins Obliviously Realize $\mathcal{F}_{\text{threshbins}}$

Here we analyze the ThreshBins subroutine and show that it obliviously realize $\mathcal{F}_{\text{threshbins}}$ 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 $\mathcal{L}_{\text{threshbins}}(\lambda, X, B, \epsilon_0)$ simply output $L = (P, D)$.

Lemma B.7. *The algorithm* ThreshBins *δ-obliviously realize $\mathcal{F}_{\text{thresbins}}$ with $\delta = O(\exp(-\Theta(\log^2 \lambda)))$. 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 $\mathcal{F}_{\text{threshbins}}$ 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, the statistical distance between ideal and real execution is equal to the probability that ThreshBins fails to compute the same output as $\mathcal{F}_{\text{threshbins}}$ given (X, R, D, P). In the following, we upper bound this probability.

The only non-trivial part is the k iterations involving the buffer Buf. We prove that except with $\exp(-\Theta(\log^2 \lambda))$ probability, the following invariant holds for all $0 \leq i \leq k$: after iteration i, the first im bins have been output correctly, and the buffer Buf contains elements in $X[1+\sum_{j=1}^{im} R_j..C[i]+s]$.

Observe the base case $i = 0$ holds trivially, and the case $i = k$ implies the correctness of the algorithm.

We assume that for some $i \geq 1$, iteration $i - 1$ completes successfully according to the above invariant, and we consider iteration i.

By the accuracy of the prefix sum estimate C guaranteed by Theorem 3.4, all elements that are supposed to go from bin $(i-1)m + 1$ to im have indices at most $C[i] + s$, which are added to the buffer Buf with all but negligible probability.

Moreover, the accuracy of C also guarantees that the number of elements with bin number higher than im added to Buf is at most $2s$ (with all but negligible probability). Finally, Lemma A.4 ensures that the number of elements from bin $(i-1)m + 1$ to im is at most $\frac{mZ}{2}(1 + \frac{1}{\log^2 \lambda})$ (with all but negligible probability).

Therefore, before Buf is truncated back to its original capacity, the number of real elements it contains is at most $\frac{mZ}{2}(1 + \frac{1}{\log^2 \lambda}) + 2s$. Hence, no real element is lost after truncation (with all but negligible probability).

After this, the correctness of the inductive step is guaranteed by the ObliviousBinPlace subroutine. This completes the inductive argument.

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

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

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

B.4 Proof of Theorem 5.3

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

$\mathcal{F}_{\text{merge}}(\lambda, I_0, I_1, \epsilon)$:

Assume: The same setting as Merge.

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

Lemma B.9. *The algorithm* Merge *δ-obliviously realize* $\mathcal{F}_{\text{merge}}$ *with* $\delta = O(\exp(-\Theta(\log^2 \lambda)))$. *Moreover, the running time is* $O(BZ(\log \frac{1}{\epsilon_0} + \log\log \lambda))$.

Proof. We observe that by construction, the access pattern of Merge is determined by the leakage (P_0, D_0, P_1, D_1). Thus, given the leakage, the access pattern can be readily simulated.

Let us consider a hybrid functionality $\mathcal{F}'_{\text{merge}}$ that on input $(\lambda, I_0, I_1, \epsilon)$, instead of merging I_0 and I_1 directly, $\mathcal{F}'_{\text{merge}}$ first calls $\mathcal{F}_{\text{threshbins}}(\lambda, I_b, B_b, 0.1\epsilon)$ 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 $\mathcal{F}_{\text{merge}}$ and $\mathcal{F}'_{\text{merge}}$ 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 A.4. Thus, up to an $O(\exp(-\Theta(\log^2 \lambda)))$ statistical error, we can switch to consider the hybrid functionality $\mathcal{F}'_{\text{merge}}$.

Now, note that $\mathcal{F}'_{\text{merge}}$ and Merge call $\mathcal{F}_{\text{threshbins}}(\lambda, I_b, B_b, 0.1\epsilon)$ and $\mathsf{ThreshBins}(\lambda, I_b, B_b, 0.1\epsilon)$ for $b \in \{0, 1\}$, respectively. Since $\mathsf{ThreshBins}$ obliviously realized $\mathcal{F}_{\text{threshbins}}$ with error $O(\exp(-\log^2 \lambda))$, we know that the output thresh-bins T_b (which are tagged with P_b, D_b) of $\mathsf{ThreshBins}$ and $\mathcal{F}_{\text{threshbins}}$ are statistically close with statistical distance at most $O(\exp(-\Theta(\log^2 \lambda)))$. From here, the difference between $\mathcal{F}'_{\text{merge}}$ and Merge is that $\mathcal{F}'_{\text{merge}}$ directly merges T_0 and T_1, whereas Merge uses $\mathsf{MergeThreshBins}$. By an union bound, the statistical distance between the real and ideal executions can be upper bounded by $O(\exp(-\log^2 \lambda))$ plus the probability that $\mathsf{MergeThreshBins}$ produces incorrect merged output, given input $(\lambda, T_0, T_1, 0.1\epsilon)$. In the following, we upper bound this probability.

The most non-trivial part of the $\mathsf{MergeThreshBins}$ algorithm is the for loop with $\tau := \lceil \frac{B}{m} \rceil$ iterations. We show the following invariant for $0 \le i \le \tau$: after the i-iteration, the following properties hold.

1. During iteration i, at most $(m + 4)$ new bins are inserted into Buf and at least m bins change to safe during iteration i.
2. Suppose k_0 and k_1 are the largest indices such that the bins $T_0[k_0]$ and $T_1[k_1]$ are safe (at the end of the i-th iteration). Then, the smallest $\max\{D_0[k_0] + D_1[k_1] - 2s, 0\}$ elements in the two input thresh-bins have been correctly copied to the prefix of Output, with all but $\exp(-\Theta(\log^2 \lambda))$ probability.

Then, the invariant trivially holds for $i = 0$. Furthermore, if the invariant holds for $i = \tau$, then this means that all bins have been inserted into the Buf and there are at most $4s$ elements remaining in the Buf. Hence, the last oblivious sort can deliver the remaining elements correctly to Output.

We next prove the inductive step. Assume that for some $i \ge 1$, the invariant holds at the end of the $(i - 1)$-st iteration, and we consider the i-th iteration.

First, by the description of the algorithm, in each iteration, at most $(m + 4)$ new bins ($m + 2$ in W plus possibly two extra bins) are inserted into Buf.

Observe that out of the $m+2$ bins in W (which are not safe at the beginning of iteration i) inserted into Buf, at least m of them will change to safe. The reason is that for each $b \in \{0,1\}$, except the bin from T_b with the largest index in W, all other bins in W from T_b must be safe, because of the extra bin (in addition to the $m+2$ bins in W) from T_{1-b} inserted.

Since the invariant holds at the end of iteration $i-1$, this implies that at the end of iteration $i-1$, the number of elements in Buf is at most equivalent to that from 4 bins plus $4s$ (with all but $\exp(-\Theta(\log^2 \lambda))$ probability).

Therefore, it follows that during iteration i, inserting elements from at most $m+4$ bins into Buf will not cause it to overflow (with all but $\exp(-\Theta(\log^2 \lambda))$ probability, due to Lemma A.4.

Next, we argue that at any moment, if x is the number of elements from all the safe bins, then the smallest x elements from the input have already been inserted into Buf. Consider any safe bin $T_b[k]$. By the definition of safe bin, any element from T_{1-b} in a bin not inserted in Buf must be larger than the interior point of $T_b[k+1]$. This implies that any element smaller than any element in a safe bin must already be inserted in **Buf**. This implies the beginning statement of this paragraph.

Because the invariant holds at the end of iteration $i-1$, it follows that during iteration i, the elements with total global rank from $\mathsf{count}+1$ to $\mathsf{newcount}$ must already be in the Buf, since no real elements have been lost so far (with all but $\exp(-\Theta(\log^2 \lambda))$ probability). Hence, those elements can be delivered to Output after iteration i. This completes the inductive step.

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

C Building Blocks

C.1 Obliviously Realizing the Interior Point Mechanism

We start by recalling the recursive differentially private algorithm $\mathsf{InteriorPoint}$ 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 algorihtm simply ignores the dummy elements. We refer the readers to [8] for the intuition behind the algorithm.

$\mathsf{InteriorPoint}(S, \beta, \epsilon, \delta)$

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

Algorithm:

1. If $|X| \leq 32$, then use the exponential mechanism with privacy parameter ϵ 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 $e^{\epsilon \cdot q(S,x)/2}$ (since the sensitivity of the quality function is 1).

2. Let $k = \lfloor \frac{386}{\epsilon} \ln(\frac{4}{\beta\epsilon\delta}) \rfloor$, 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 $\mathsf{InteriorPoint}$ recursively on $S' = (z_j)_{j=1}^{(n-2k)/2} \in (X')^{(n-2k)/2}$ with parameters β, ϵ, δ. Recall that $|X'| = \log |X|$. Denote the returned value by z.

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 β, ϵ, δ, and the quality function $q : X^* \times \{0, 1\}^{z+1} \to \mathbb{N}$, 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 $e^{\epsilon \cdot q(Y,L)/2}$.[6]

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

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

8. If $\hat{big} \geq \frac{3k}{2}$ then return L_1. Otherwise return L_0.

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.

- 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, respecitvely 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 implmement the exponential and choosing mechanisms in step 1 and 5, which involves sampling from a distribution defined by numbers of the form $e^{(\epsilon/2)\cdot j}$ 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

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

set of such prefixes. Recall that $q(Y, L) = \#\{y_i \in Y : \mathsf{pref}(y_i) = L\}$, where $\mathsf{pref}(y_i)$ denote the $z+1$ bits prefix of y_i. This can be done by invoking oblivious aggregation (see Section C) with key $k_i = \mathsf{pref}(y_i)$ and value $v_i = 1$ (with padded dummy entries to size $n' - 2k$), and the aggregation function Aggr being the summation function. The ouput 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 $e^{(\epsilon/2)\cdot q_j}$. To optimize the efficiency, we do it as follows. Let $q_{\max} = \max_j q_j$. We compute $w_j = e^{(\epsilon/2)\cdot(q_j - q_{\max})} \in (0, 1]$ 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 $v_j = \sum_{\ell \leq j} \lfloor 2^p \cdot w_j \rfloor$. 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 a $o(\delta)$ statistical distance error due to the finite precision. To compute $w_j = \{e^{(\epsilon/2)\cdot(q_j - q_{\max})}\}$ with p bits of precision, note that these are values of the form $e^{-(\epsilon/2)\cdot t}$ for $t \in \mathbb{N}$. Since we only need p bits of precision, the value rounds to 0 when t is too large. Let $t_{\max} = O((1/\epsilon) \cdot \log p)$ be the largest t we need to consider. We precompute the values $\alpha_k = e^{-(\epsilon/2)\cdot 2^k}$ for $k \in \{0, 1, \ldots, \lfloor \log t_{\max} \rfloor\}$, and compute w_j by multiplying a subset of α_k, i.e., by the standard repeated squaring algorithm. (Note that for obliviousness, the access pattern needs to go over all α_k to hide the subset). Finally, to compute α_k, we first compute $\alpha_0 = e^{-\epsilon/2}$ 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.

$\mathsf{Choosing}(Y, z, \epsilon)$

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

Algorithm:

1. Compute the quality function: Invoke oblivious aggregation (see Section C) with key $k_i = \mathsf{pref}(y_i)$ and value $v_i = 1$ (with padded dummy entries to size $n' - 2k$), and the aggregation function Aggr being the summation function. The ouput 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 $\alpha_0 = e^{-\epsilon/2}$ by Taylor expansion, and $\alpha_k = \alpha_{k-1}^2$ for $k \in \{1, \ldots, \lfloor \log t_{\max} \rfloor\}$.

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

4. Compute the accumulated sum $v_j = \sum_{\ell \leq j} \lfloor 2^p \cdot w_j \rfloor$. 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 $\mathsf{time}_{\mathsf{add}}(p)$, $\mathsf{time}_{\mathsf{mult}}(p)$, $\mathsf{time}_{\mathsf{exp}}(p)$, etc., to denote the time to perform addition and multiplication, and compute $e^{-\epsilon/2}$, etc., with p bits of precision. We note that the dominating cost is the computation of the weights w_j, which takes $O(n' \cdot (\log t_{\max}) \cdot \mathsf{time}_{\mathsf{mult}}(p))$ 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 α_k's, which takes time $\mathsf{time}_{\mathsf{exp}} + (\log t_{\max}) \cdot \mathsf{time}_{\mathsf{mult}}(p)$, and the sampling of $u \leftarrow_R [v_{n'-2k}]$. Note that $\mathsf{time}_{\mathsf{exp}}(p)$ is at most $O(p \cdot \mathsf{time}_{\mathsf{mult}}(\mathsf{p})) \leq O(n' \cdot \mathsf{time}_{\mathsf{mult}}(\mathsf{p}))$. All these terms are dominated by the cost of computing w_j's.

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

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

where $\log t_{\max} = O(\log((1/\epsilon) \cdot \log(n/\delta)))$ and $p = O(\log(n/\delta))$. Finally, note that $\mathsf{InteriorPoint}$ 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 $\alpha_k = e^{-(\epsilon/2)\cdot 2^k}$ for $k \in \{0, 1, \ldots, \lfloor \log t_{\max} \rfloor\}$, but all the values $e^{-(\epsilon/2)\cdot t}$ for $t \in \{0, 1, \ldots, t_{\max}\}$ 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 $O(\log t_{\max})$ 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 $\mathsf{time}_{\mathsf{mult}}(p) \leq O((p/w)^2)$. When $\delta = \mathsf{negl}(n')$ 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 C.1 (Differentially private interior point, finite word size version)**.** *Let w be the word size of a RAM machine. Let $\beta, \epsilon, \delta > 0$ 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 $n \geq \frac{18500}{\epsilon} \cdot 2^{\log^* U} \cdot \log^* U \cdot \ln \frac{4 \log^* U}{\beta\epsilon\delta}$, the algorithm*

- *completes consuming only $O((n' \log n') + (n' \cdot \log((1/\epsilon) \cdot \log p) \cdot (p/w)^2))$ time and number of memory accesses, where $p = 2\log(n/\delta)$.*
- *the algorithm produces an (ϵ, δ)-differential private outcome;*
- *with probability $1 - \beta$, the outcome is a correct interior point of the input array S; and*
- *the algorithm's memory access patterns are independent of the input array.*

C.2 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 $\mathsf{Inp} := \{k_i, v_i\}_{i\in[n]}$ 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 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:

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

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

C.3 Oblivious Propagation for a Sorted Array

Oblivious propagation [40] 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. [40] 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.4 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.

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) 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.