

---

# An Algorithmic Framework For Differentially Private Data Analysis on Trusted Processors

---

Joshua Allen  
Harsha Nori

Bolin Ding\*  
Olga Ohrimenko

Janardhan Kulkarni  
Sergey Yekhanin

Microsoft

## Abstract

Differential privacy has emerged as the main definition for private data analysis and machine learning. The *global* model of differential privacy, which assumes that users trust the data collector, provides strong privacy guarantees and introduces small errors in the output. In contrast, applications of differential privacy in commercial systems by Apple, Google, and Microsoft, use the *local* model. Here, users do not trust the data collector, and hence randomize their data before sending it to the data collector. Unfortunately, local model is too strong for several important applications and hence is limited in its applicability. In this work, we propose a framework based on trusted processors and a new definition of differential privacy called *Oblivious Differential Privacy*, which combines the best of both local and global models. The algorithms we design in this framework show interesting interplay of ideas from the streaming algorithms, oblivious algorithms, and differential privacy.

## 1 Introduction

Most large IT companies rely on access to raw data from their users to train machine learning models. However, it is well known that models trained on a dataset can release private information about the users that participate in the dataset [10, 48]. With new GDPR regulations and also ever increasing awareness about privacy issues in the general public, doing private and secure machine learning has become a major challenge to IT companies. To make matters worse, while it is easy to spot a violation of privacy when it occurs, it is much more tricky to give a rigorous definition of it.

*Differential privacy (DP)*, introduced in the seminal work of Dwork *et al.* [16], is arguably the only mathematically rigorous definition of privacy in the context of machine learning and big data analysis. Over the past decade, DP has established itself as the defacto standard of privacy with a vast body of research and growing acceptance in industry. Among its many strengths, the promise of DP is intuitive to explain: No matter what the adversary knows about the data, the privacy of a single user is protected from output of the data-analysis. A differentially private algorithm guarantees that the output does not change significantly, as quantified by a parameter  $\epsilon$ , if the data of any single user is omitted from the computation, which is formalized as follows.

**Definition 1.1.** A randomized algorithm  $\mathcal{A}$  is  $(\epsilon, \delta)$ -differentially private if for all any two neighboring databases  $\mathcal{D}_1, \mathcal{D}_2$  any subset of possible outputs  $S \subseteq \mathcal{Z}$ , we have:

$$\Pr[\mathcal{A}(\mathcal{D}_1) \in S] \leq e^\epsilon \cdot \Pr[\mathcal{A}(\mathcal{D}_2) \in S] + \delta.$$

This definition of DP is often called the *global* definition (GDP), as it assumes that users are willing to trust the data collector. By now, there is a large body of work on *global differential privacy*,

---

\*Current affiliation: Alibaba Group. Work done while at Microsoft.

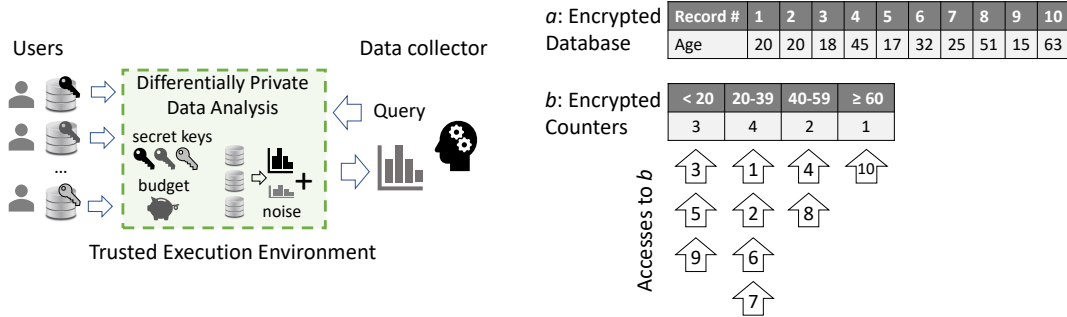


Figure 1: *Left*: Secure differentially-private data-analysis. *Right*: Visualization of the access pattern of a naive histogram computation over a database and four age ranges counters ( $k = 4$ ) stored in arrays  $a$  and  $b$ , respectively. The code reads a record from  $a$ , decrypts it, accesses the corresponding age bucket in  $b$ , increments its counter, encrypts it and writes it back. The arrows indicate increment accesses to the histogram counters ( $b$ ) and the numbers correspond to records of  $a$  that were accessed prior to these accesses. An adversary of the accesses to  $a$  and  $b$  learns the histogram and which database records belong to the same age range.

and many non-trivial machine learning problems can be solved in this model very efficiently. See authoritative book by Dwork and Roth [18] for more details. However in the context of IT companies, adoption of GDP is not possible as there is no trusted data collector – users want privacy of their data from the data collector! Because of this, all industrial deployments of DP by Apple, Google, and Microsoft, with the exception of Uber [27], have been set in the so called *local model of differential privacy (LDP)* [19, 15, 14]. In the LDP model, users randomize their data *before* sending it to the data collector.

**Definition 1.2.** A randomized algorithm  $\mathcal{A} : \mathcal{V} \rightarrow \mathcal{Z}$  is  $\epsilon$ -locally differentially private ( $\epsilon$ -LDP) if for any pair of values  $v, v' \in \mathcal{V}$  held by a user and any subset of output  $S \subseteq \mathcal{Z}$ , we have:

$$\Pr[\mathcal{A}(v) \in S] \leq e^\epsilon \cdot \Pr[\mathcal{A}(v') \in S].$$

Despite its very strong privacy guarantees, the local model has several drawbacks compared to the global model: many important problems cannot be solved in the LDP setting within a desired level of accuracy. Consider the simple task of understanding the number of distinct websites visited by users, or words in text data. This problem admits no good algorithms in LDP setting, whereas in the global model the problem becomes trivial. Even for problems that can be solved in LDP setting [19, 4, 3, 15], errors and  $\epsilon$  are significantly larger compared to the global model. For example, if one is interested in understanding the histogram of websites visited by users, in the LDP setting an optimal algorithm achieves an error of  $\Omega(\sqrt{n})$ , whereas in the global model error is  $O(\frac{1}{\epsilon})$ ! See experiments and details in [8] for scenarios where the errors introduced by (optimal) LDP algorithms are unacceptable in practice. Finally, most IT companies collect data repeatedly, hence the database keeps evolving. The global model of differential privacy allows for very clean solutions for this problem based on *sparse vector techniques* [17, 45] such that privacy budget grows only when the actual outcome changes substantially. The local model of differential privacy admits no such elegant solutions. These drawbacks of LPD naturally lead to the following question:

*Is there a way to bridge local and global differential privacy models such that users enjoy the privacy guarantees of the local model whereas the data collector enjoys the accuracy of the global model?*

We propose a framework (see Figure 1 (left)) based on trusted processors (for example, Intel SGX [26]) and a new definition called Oblivious Differential Privacy (ODP) that combines the best of both local and global models.

1. Data is collected, stored, and used in an encrypted form and is protected from the data collector.
2. The data collector obtains information about the data only through the results of a DP-algorithm.

The DP-algorithms themselves run within a Trusted Execution Environments (TEE) that guarantee that the data is decrypted only by the processor during the computation and is always encrypted in

memory. Hence, raw data is inaccessible to anyone, including the data collector. To this end, our framework is similar to other systems for private data analysis based on trusted processors, including machine learning algorithms [37] and data analytical platforms such as PROCHLO [8, 46]. Recently, systems for supporting TEE have been announced by Microsoft<sup>2</sup> and Google<sup>3</sup>, and we anticipate a wide adoption of this model for doing private data analysis and machine learning.

The private data analysis within trusted processors has to be done carefully since the rest of the computational environment is untrusted and is assumed to be under adversary’s control. Though the data is encrypted, the adversary can learn private information based on memory access patterns through caches and page faults. In particular, *memory access patterns* are often dependent on private information and have been shown to be sufficient to leak information [38, 7, 53]. Since differentially private algorithms in the global model have been designed with a trusted data collector in mind, they are also susceptible to information leakage through their access pattern.

The main goal of our paper is to formalize the design of differentially private algorithms in the trusted processor environments. Our contributions are summarized below:

- Building on the recent works of [37, 8], we propose a framework that enables collection and analysis of data in the global model of differential privacy without relying on a trusted curator. Our framework uses encryption and secure processors to protect data and computation such that only the final differentially private output of the computation is revealed.
- Trusted execution environments impose certain restrictions on the design of algorithms. We formalize the mathematical model for designing differentially private algorithms in TEEs.
- We define a new class of differentially-private algorithms (Definition 3.1) called *obliviously differentially private algorithms* (ODP), which ensure that privacy leakage that occurs through algorithm’s memory access patterns and the output together satisfy the differential privacy guarantees.
- We design ODP algorithms with provable performance guarantees for some commonly used statistical routines such as computing the number of distinct elements, histogram, and heavy hitters. We prove that the privacy and error guarantees of our algorithms (Theorems (4.1, 4.4, 4.5)) are significantly better than in the local model.

A technically novel aspect of our paper is that it draws ideas from various different fields: streaming algorithms, oblivious algorithms, and differentially private algorithms. This fact becomes clear in §4 where we design ODP algorithms.

**Related work** There are several systems that propose confidential data analysis using TEEs [37, 54, 46, 8]. PROCHLO [8], in particular, provides support for differential private data analysis. While PROCHLO emphasizes more on the system aspects (without formal proofs), our work gives a formal framework for analyzing and designing algorithms for private data analysis in TEEs, with full proofs for many statistical problems.

## 2 Preliminaries

### 2.1 Secure Data Analysis with Trusted Processors and Memory Access Leakage

A visualization of our framework is given in Figure 1 (left). We use Intel Software Guard Extensions (SGX) as an example of a trusted processor. Intel SGX [26] is a set of CPU instructions that allows user-level code to allocate a private region of memory, called an enclave (which we also refer to as a TEE), which is accessible only to the code running in an enclave. The enclave memory is available in raw form only inside the physical processor package, but it is encrypted and integrity protected when written to memory. As a result, the code running inside of an enclave is isolated from the rest of the system, including the operating system. Additionally, Intel SGX supports software attestation [2] that allows the enclave code to get messages signed with a private key of the processor along with a digest of the enclave. This capability allows users to verify that they are communicating with a specific piece of software (i.e., a differentially-private algorithm) running in an enclave hosted by

<sup>2</sup>“Introducing Azure confidential computing”, accessed May 16, 2018.

<sup>3</sup>“Introducing Asylo: an open-source framework for confidential computing”, accessed May 16, 2018.

the trusted hardware. Once this verification succeeds, the user can establish a secure communication channel with the enclave (e.g., using TLS) and upload data. When the computation is over, the enclave, including the local variables and data, is deleted.

An enclave can access data that is managed and protected by the trusted processor (e.g., data in registers and caches) or software that is not trusted (e.g., an operating system). As a result, in the latter case, data has to be encrypted and integrity protected by the code running inside of an enclave. Unfortunately, encryption and integrity are not sufficient to protect against the adversary described in the introduction that can see the addresses of the data being accessed even if the data is encrypted. There are several ways the adversary can extract the addresses, i.e., the *memory access pattern*. Some typical examples are: an adversary with physical access to a machine can attach probes to a memory bus, an adversary that shares the same hardware as the victim enclave code (e.g., a co-tenant) can use shared resources such as caches to observe cache-line accesses, while a compromised operating system can inject page faults and observe page-level accesses. Memory access patterns have been shown to be sufficient to extract secrets and data from cryptographic code [29, 39, 7, 42, 38], from genome indexing algorithms [9], and from image and text applications [53]. (See Figure 1 (right) for a simple example of what can be extracted by observing accesses of a histogram computation.) As a result, accesses leaked through memory side-channels undermine the confidentiality promise of enclaves [53, 47, 25, 31, 13, 9, 34].

Memory side-channels should not be confused with vulnerabilities introduced by floating-point implementations [33]. We also do not consider side-channel attacks based on timing of individual operations or power analysis.

## 2.2 Data-Oblivious Algorithms

Data-oblivious algorithms [21, 37, 50, 22] are designed to protect memory addresses against the adversary described in §2.1: they produce data access patterns that appear to be independent of the sensitive data they compute on. They can be seen as external-memory algorithms that perform computation inside of small private memory while storing the encrypted data in the external memory and accessing it in a *data-independent* manner. We formally capture this property below. Suppose external memory is represented by an array  $a[1, 2, \dots, M]$  for some large value of  $M$ .

**Definition 2.1** (Access pattern). *Let  $\text{op}_j$  be either a read( $a[i]$ ) operation that reads data from the location  $a[i]$  to private memory or a write( $a[i]$ ) operation that copies some data from the private memory to the external memory  $a[i]$ . Then, let  $s := (\text{op}_1, \text{op}_2, \dots, \text{op}_t)$  denote an access pattern of length  $t$  of algorithm  $\mathcal{A}$  to the external memory.*

Note that the adversary can see only the addresses accessed by the algorithm and whether it is a read or a write. It cannot see the data since it is encrypted using probabilistic encryption that guarantees that the adversary cannot tell if two ciphertexts correspond to the same record or two different ones.

**Definition 2.2** (Data-oblivious algorithm). *An algorithm  $\mathcal{A}$  is data-oblivious if for any two inputs  $I_1$  and  $I_2$ , and any subset of possible memory access patterns  $S \subseteq \mathcal{S}$ , where  $\mathcal{S}$  is the set of all possible memory access patterns produced by an algorithm, we have:*

$$\Pr[\mathcal{A}(I_1) \in S] = \Pr[\mathcal{A}(I_2) \in S]$$

It is instructive to compare this definition with the definition of differential privacy. The definition of oblivious algorithms can be thought of as a generalization of DP to memory access patterns, where  $\epsilon = 0$  and the guarantee should hold even for *non-neighboring* databases. Similar to external memory algorithms, the overhead of a data-oblivious algorithm is measured in the number of accesses it makes to external memory, while computations on private memory are assumed to be constant. Some algorithms naturally satisfy Definition 2.2 while others require changes to how they operate. For example, scanning an array is a data-oblivious algorithm since for any array of the same size every element of the array is accessed. Sorting networks [6] are also data-oblivious as element identifiers accessed by compare-and-swap operations are fixed based on the size of the array and not its content. On the other hand, quicksort is not oblivious as accesses depend on the comparison of the elements with the pivot element. As can be seen from Figure 1 (right), a naive histogram algorithm is also not data-oblivious. (See Appendix C for overheads of several oblivious algorithms.)

**Oblivious RAM (ORAM)** is designed to hide the indices of accesses to an array of size  $n$ , i.e., it hides how many times and when an index was accessed. There is a naive and inefficient way to hide an

access by reading and writing to every index. Existing ORAM constructions incur sublinear overhead by using specialized data structures and re-arranging the external memory [21, 24, 43, 49, 52, 23]. The best known ORAM construction has  $O((\log n)^2 / \log \log n)$  [30] overhead. Since it incurs high constants, Path ORAM [50] with the overhead of  $O((\log n)^2)$  is a preferred option in practice. ORAM can be used to transform any RAM program whose number of accesses does not depend on sensitive content; otherwise, the number of accesses needs to be padded. However, if one is willing to reveal the algorithm being performed on the data, then for some computations the overhead of specialized constructions can be asymptotically lower than of the one based on ORAM.

**Data-oblivious shuffle [36]** takes as a parameter an array  $a$  (stored in external memory) of size  $n$  and a permutation  $\pi$  and permutes  $a$  according to  $\pi$  such that  $\pi$  is not revealed to the adversary. The Melbourne shuffle [36] is a randomized data-oblivious shuffle that makes  $O(cn)$  deterministic accesses to external memory, assuming private memory of size  $\sqrt[n]{n}$ , and fails with negligible probability. For example, for  $c = 2$  the overhead of the algorithm is constant as any non-oblivious shuffle algorithm has to make at least  $n$  accesses. The Melbourne shuffle with smaller private memories of size  $m = \omega(\log n)$  incurs slightly higher overhead of  $O(n \log n / \log m)$  as showed in [40]. We will use `oblivious_shuffle( $a$ )` to refer to a shuffle of  $a$  according to some random permutation that is hidden from the adversary.

### 3 Algorithmic Framework and Oblivious Differential Privacy

We now introduce the definition of Oblivious Differential Privacy (ODP), and give an algorithmic framework for the design of ODP-algorithms for a system based on trusted processors (§2.1). As we mentioned earlier, off-the-shelf DP-algorithms may not be suitable for TEEs for two main reasons.

**Small Private Memory:** The private memory, which is protected from the adversary, available for an algorithm within a trusted processor is much smaller than the data the algorithm has to process. We believe that a reasonable assumption on the size of private memory is *polylogarithmic* in the input.

**Access Patterns Leak Privacy:** An adversary who sees memory access patterns of an algorithm to external memory can learn useful information about the data, compromising the differential privacy guarantees of the algorithm.

Therefore, the algorithm designer needs to guarantee that memory access patterns do not reveal any private information, and the overall algorithm is differentially private<sup>4</sup>. We formalize this by introducing the notion of *Oblivious Differential Privacy*, which combines the notions of differential privacy and oblivious algorithms.

**Definition 3.1.** *Let  $D_1$  and  $D_2$  be any two neighboring databases that have exactly the same size  $n$  but differ in one record. A randomized algorithm  $\mathcal{A}$  that has small private memory (i.e., sublinear in  $n$ ) and accesses external memory is  $(\epsilon, \delta)$ -obliviously differentially private (ODP), if for any subset of possible memory access patterns  $S \subseteq \mathcal{S}$  and any subset of possible outputs  $O$  we have:*

$$\Pr[\mathcal{A}(D_1) \in (O, S)] \leq e^\epsilon \cdot \Pr[\mathcal{A}(D_2) \in (O, S)] + \delta.$$

We believe that the above definition gives a systematic way to design DP algorithms in TEE settings. Note that our definition of neighboring databases is slightly different from the textbook definition, where one assumes that a single record is added or deleted. This distinction is important to satisfy obliviousness, and we discuss more in §4. An algorithm that satisfies the above definition guarantees that the private information released through output of the algorithm *and* through the access patterns is quantified by the parameters  $(\epsilon, \delta)$ . Similar to our definition, Wagh *et al.* [51] and more recently, in a parallel work, Chan *et al.* [11] also consider relaxing the definition of obliviousness for hiding access patterns from an adversary. However, their definitions do not consider output of the algorithm. This turns out to be crucial since algorithms that satisfy the definition in [51, 11] may not satisfy DP when the output is released, which is the main motivation for using differentially private algorithms.

*Remark:* In the real world, the implementation of a TEE relies on cryptographic algorithms (e.g., encryption and digital signatures) that are computationally secure and depend on a security parameter of the system. As a result any differentially private algorithm operating inside of a TEE has a non-zero parameter  $\delta$  that is negligible in the security parameter.

<sup>4</sup>Here, data collector runs algorithms on premise. See Appendix A for restrictions in the cloud setting.

**Connections to Streaming Algorithms:** One simple strategy to satisfy Definition 3.1 is to take a DP-algorithm and guarantee that every time the algorithm makes an access to the public memory, it makes a pass over the entire data. However, such algorithm incurs a multiplicative overhead of  $n$  on the running time, and the goal would be to minimize the number of passes made over the data. Interestingly, these algorithms precisely correspond to the *streaming algorithms*, which are widely studied in big-data analysis. In the streaming setting, one assumes that we have only  $O(\log n)$  bits of memory and data stream consists of  $n$  items, and the goal is to compute functions over the data. Quite remarkably, several functions can be approximated very well in this model. See [35] for an extensive survey. Since there is a large body of work on streaming algorithms, we believe that many algorithmic ideas there can be used in the design of ODP algorithms. We give example of such algorithms for distinct elements §4.1 and heavy hitters problem in §4.3.

**Privacy Budget:** Finally, we note that the system based on TEEs can support interactive data analysis where the privacy budget is hard-coded (and hence verified by each user before they supply their private data). The data collector’s queries decrease the budget appropriately and the code exits when privacy budget is exceeded. Since the budget is maintained as a local variable within the TEE it is protected from replay attacks while the code is running. If TEE exits, the adversary cannot restart it without notifying the users since the code requires their secret keys. These keys cannot be used for different instantiations of the same code as they are also protected by TEE and are destroyed on exit.

## 4 Obliviously Differentially Private Algorithms

In this section, we show how to design ODP algorithms for the three most commonly used statistical queries: counting the number of distinct elements in a dataset, histogram of the elements, and reporting heavy hitters. The algorithms for these problems exhibit two common themes: 1) The error introduced by ODP algorithms is substantially smaller than LDP algorithms; 2) The interplay of ideas from the streaming and oblivious algorithms literature in the design of ODP algorithms.

Before we continue with the construction of ODP algorithms, we make a subtle but important point. Recall that in our Definition 3.1, we require that two neighboring databases have exactly the same size. If the neighboring databases are of different sizes, then the access patterns can be of different lengths, and it is impossible to satisfy the ODP-definition. This definition does not change the privacy guarantees as in many applications the size of the database is known in advance; e.g., number of users of a system. However, it has implications on the sensitivity of the queries. For example, histogram queries in our definition have sensitivity of 2 where in the standard definition it is 1.

### 4.1 Number of Distinct Items in a Database

As a warm-up for the design of ODP algorithms, we begin with the distinct elements problem. Formally, suppose we are given a set of  $n$  users and each user  $i$  holds an item  $v_i \in \{1, 2, \dots, m\}$ , where  $m$  is assumed to be much larger than  $n$ . This is true if a company wants to understand the number of distinct websites visited by its users or the number of distinct words that occur in text data. Let  $n_v$  denote the number of users that hold the item  $v$ . The goal is to estimate  $n^* := |\{v : n_v > 0\}|$ .

We are not aware of a reasonable solution that achieves an additive error better than  $\Omega(n)$  for this problem in the LDP setting. In a sharp contrast, the problem becomes very simple in our framework. Indeed, a simple solution is to do an *oblivious sorting* [1, 6] of the database elements, and then count the number of distinct elements by making another pass over the database. Finally, one can add Laplace noise with the parameter  $\frac{1}{\epsilon}$ , which will guarantee that our algorithm satisfies the definition of ODP. This is true as a) the sensitivity of the query is 1, as a single user can increase or decrease the number of distinct elements by at most 1; b) we do oblivious sorting. Furthermore, the expected (additive) error of such an algorithm is  $1/\epsilon$ . The performance of this algorithm depends on the underlying sorting algorithm:  $O(n \log n)$  with AKS [1] and  $O(n(\log n)^2)$  with Batcher’s sort [6]. Though the former is asymptotically superior to Batcher’s sort, it has high constants [41].

Recall that  $n^*$  denotes the number of distinct elements in a database. Thus we get:

**Theorem 4.1.** *There exists an oblivious sorting based  $(\epsilon, 0)$ -ODP algorithm for the problem of finding the number of distinct elements in a database that runs in time  $O(n \log n)$ . With probability at least  $1 - \theta$ , the number of distinct elements output by our algorithm is  $(n^* \pm \log(1/\theta) \frac{1}{\epsilon})$ .*

While above algorithm is optimal in terms of error, we propose a more elegant *streaming algorithm* that does the entire computation in the private memory! The main idea is to use a *sketching technique* to maintain an approximate count of the distinct elements in the private memory and report this approximate count by adding noise from  $\text{Lap}(1/\epsilon)$ . This will guarantee that our algorithm is  $(\epsilon, 0)$ -ODP, as the entire computation is done in the private memory and the Laplace mechanism is  $(\epsilon, 0)$ -DP. There are many streaming algorithms (e.g., Hyperloglog) [20, 28] which achieve  $(1 \pm \alpha)$ -approximation factor on the number of distinct elements with a space requirement of  $\text{polylog}(n)$ . We use the following (optimal) algorithm in [28].

**Theorem 4.2.** *There exists a streaming algorithm that gives a  $(1 \pm \alpha)$  multiplicative approximation factor to the problem of finding the distinct elements in a data stream. The space requirement of the algorithm is at most  $\frac{\log n}{\alpha^2} + (\log n)^2$  and the guarantee holds with probability  $1 - 1/n$ .*

It is easy to convert the above algorithm to an ODP-algorithm by adding noise sampled from  $\text{Lap}(\frac{1}{\epsilon})$ .

**Theorem 4.3.** *There exists a single pass (or online)  $(\epsilon, 0)$ -ODP algorithm for the problem of finding the distinct elements in a database. The space requirement of the algorithm is at most  $\frac{\log n}{\alpha^2} + (\log n)^2$ . With probability at least  $1 - 1/n - \theta$ , the number of distinct elements output by our algorithm is  $(1 \pm \alpha)n^* \pm \log(1/\theta)\frac{1}{\epsilon}$ .*

The additive error of  $\pm \log(1/\theta)\frac{1}{\epsilon}$  is introduced by the Laplace mechanism, and the multiplicative error of  $(1 \pm \alpha)$  is introduced by the sketching scheme. Although this algorithm is not optimal in terms of the error compared to Theorem 4.1, it has the advantage that it can maintain the approximate count in an *online* fashion or by making a *single pass* over the database!

## 4.2 Histogram

Let  $D$  be a database with  $n$  records. We assume that each record in the database has a unique identifier. Let  $\mathcal{D}$  denote all possible databases of size  $n$ . Each record (or element)  $r \in \mathcal{D}$  has a *type*, which, without loss of generality, is an integer in the set  $\{1, 2, \dots, k\}$ .

For a database  $D \in \mathcal{D}$ , let  $n_i$  denote the number of elements of type  $i$ . Then the histogram function  $h : \mathcal{D} \rightarrow \mathbb{R}^k$  is defined as

$$h(D) := (n_1, n_2, \dots, n_k).$$

A simple differentially private histogram algorithm  $\mathcal{A}_{\text{hist}}$  returns  $h(D) + (X_1, X_2, \dots, X_k)$  where  $X_i$  are i.i.d. random variables drawn from  $\text{Lap}(2/\epsilon)$ . This algorithm is *not* obviously differentially private as the access pattern reveals to the adversary much more information about the data than the actual output. In this section, we design an ODP algorithm for the histogram problem. Let  $\hat{n}_i$  denote the number of elements of type  $i$  output by our histogram algorithm. We prove the following theorem in this paper.

**Theorem 4.4.** *For any  $\epsilon > 0$ , there is an  $(\epsilon, \frac{1}{n^2})$ -ODP algorithm for the histogram problem that runs in time  $O(\tilde{n} \log \tilde{n} / \log \log \tilde{n})$  where  $\tilde{n} = \max(n, k \log n / \epsilon)$ . Furthermore, with probability  $1 - \theta$  it holds that*

$$\max_i |\hat{n}_i - n_i| \leq \log(k/\theta) \cdot \frac{2}{\epsilon}.$$

We compare the bound of the above theorem with the one we can get in the local model. In LDP setting, there is a lower bound of  $\Omega(\sqrt{n})$  on the error of any algorithm.

To prove that our algorithm is ODP, we need to show that the distribution on the access patterns produced by our algorithm for any two neighboring databases is approximately the same. The same should hold true for the histogram output by our algorithm. We achieve this as follows. We want to use the simple histogram algorithm that adds Laplace noise with parameter  $2/\epsilon$ , which we know is  $\epsilon$ -DP. This is true since the histogram queries have sensitivity of 2; that is, if we change the record associated with the single user, then the histogram output changes for at most 2 types. We refer the reader to [18] for a full proof of this claim. Note that if the private memory size is larger than  $k$ , then the task is trivial. One can build the entire DP-histogram in the private memory by making a single pass over the database, which clearly satisfies our definition of ODP. However, in many applications  $k \gg O(\log n)$ . A common example is a histogram on bigrams of words which is commonly used in text prediction. When  $k \gg \log n$  the private memory is not enough to store the entire histogram,

and we need to make sure that memory access patterns do not reveal too much information to the adversary. Indeed, it is instructive to convince oneself that a naive implementation of the differentially private histogram algorithm in fact completely reveals the histogram to an adversary who sees the memory accesses.

One can make the naive histogram algorithm satisfy Definition 3.1 by accessing the entire public memory for every read/write operation, incurring an overhead of  $O(nk)$ . For  $k = \text{polylog}(n)$ , one can access the histogram array through an oblivious RAM (see Appendix B for details). Oblivious sorting algorithms could also be used to solve the histogram problem (see §4.3 and §B). However, sorting is usually an expensive operation in practice. Here, we give an arguably simpler and faster algorithm for larger values of  $k$  that satisfies the definition of ODP. (At a high level our algorithm is similar to the one which appeared in the independent work by Mazloom and Gordon [32], who use a differentially private histogram to protect access patterns of graph-parallel computation based on garbled circuits, as a result requiring a different noise distribution and shuffling algorithm.)

We first give a high-level overview of the algorithm. The pseudo-code is given in Algorithm 1. Let  $T = n + 20k \log n/\epsilon$ .

1. Sample  $k$  random variables  $X_1, X_2, \dots, X_k$  from  $\text{Lap}(2/\epsilon)$ . If any  $|X_i| > 10 \log n/\epsilon$ , then we set  $X_i = 0$  for all  $i = 1, 2, \dots, k$ . For all  $i$ , set  $X_i = \lceil X_i \rceil$ .
2. We create  $(10 \log n/\epsilon + X_i)$  fake records of type  $i$  and append it to the database  $D$ . This step together with step 1 ensures that  $(10 \log n/\epsilon + X_i)$  is always positive. The main reason to restrict the Laplace distribution's support to  $[-10 \log n/\epsilon, 10 \log n/\epsilon]$  is to ensure that we only have positive noise. If the noise is negative, we cannot create fake records in the database simulating this noise.
3. Next, we create  $(10k \log n/\epsilon - \sum_i X_i)$  dummy records in the database  $D$ , which do not correspond to any particular type in  $1..k$ . The dummy records have type  $k + 1$ . The purpose of this step is to ensure that the length of the output is exactly  $T$ .
4. Let  $\hat{D}$  be the augmented database that contains both dummy and fake records, where the adversary cannot distinguish between database, dummy and fake records as they are encrypted using probabilistic encryption. Obviously shuffle [36]  $\hat{D}$  so that the mapping of records to array  $a[1, 2, \dots, T]$  is uniformly distributed.
5. Initialise  $b$  with  $k$  zero counters in external memory. Scan every element from the array  $a[1, 2, \dots, T]$  and increment the counter in histogram  $b$  associated with type of  $a[i]$ . If the record corresponds to a dummy element, then access the array  $b[1, 2, \dots, k]$  in *round-robin fashion* and do a fake write without modifying the actual content of  $b$ .

We shall show that above algorithm is  $(\epsilon, \frac{1}{n^2})$ -differentially private for any  $\epsilon > 0$ . Towards that we need the following simple lemma for our proofs.

**Lemma 4.1.**  $\Pr \left[ \max_{1 \leq i \leq k} |X_i| \geq \frac{10 \log n}{\epsilon} \right] \leq \frac{1}{n^2}$  where  $X_i$  is drawn from  $\text{Lap}(\frac{2}{\epsilon})$ .

*Proof.* If  $Y \sim \text{Lap}(b)$ , then we know that  $\Pr[|Y| \geq t \cdot b] \leq e^{-t}$ . Therefore,

$$\Pr \left[ \max_i |X_i| \geq \frac{10 \log n}{\epsilon} \right] \leq \sum_{i=1}^k \Pr \left[ |X_i| \geq \frac{10 \log n}{\epsilon} \right] \leq k \cdot 1/n^5 \leq 1/n^2$$

where the last inequality follows from the fact that  $k \leq n$ . □

For rest of the proof, we will assume that  $|X_i| \leq \frac{10 \log n}{\epsilon}$  for all  $i$ . If any  $|X_i| > \frac{10 \log n}{\epsilon}$ , then we will not concern ourselves in bounding the privacy loss as it will be absorbed by the  $\delta$  parameter. Rounding of  $X_i$  to a specific integer value can be seen as a post-processing step, which DP output is immune to. Hence, going forward, we will ignore this minor point.

We prove that our algorithm satisfies Definition 3.1 in two steps: In the first step we assume that adversary does not see the access pattern, and show that output of the algorithm is  $(\epsilon, \frac{1}{n^2})$ -differentially private.

---

<sup>5</sup>Data stored in external memory is highlighted in grey. Recall that it is encrypted.



---

**Algorithm 1** Oblivious Differentially Private Histogram  $\mathcal{A}_{\text{hist}}^{\text{ODP}}(D, k)^5$

---

```

 $\hat{D} \leftarrow \text{add\_fake\_dummy}(D, k)$ 
 $D' \leftarrow \text{oblivious\_shuffle}(\hat{D})$ 
 $b \leftarrow \{0\}^k$ 
 $\text{ptr} \leftarrow 1$ 
for  $r \in D'$  do
   $i \leftarrow \text{get\_type}(r)$ 
  if  $i = k + 1$  then
     $b_{\text{ptr}} \leftarrow b_{\text{ptr}} + 0$ 
     $\text{ptr} \leftarrow \text{ptr} \bmod k + 1$ 
  else
     $b_i \leftarrow b_i + 1$ 
  end if
end for
for  $i \in 1 \dots k$  do
   $\hat{b}_i \leftarrow b_i - 10 \log n / \epsilon$ 
end for
return  $\hat{b}$ 

```

---



---

**Algorithm 2** Utility procedure for Algorithm 1

---

```

procedure  $\text{add\_fake\_dummy}(D, k)$ 
  for  $i \in 1 \dots k$  do
     $X_i \leftarrow \text{draw a sample from } \text{Lap}(\frac{2}{\epsilon})$ 
  end for
  If  $\exists i |X_i| > 10 \log n / \epsilon: \forall i X_i \leftarrow 0$ 
   $\forall i, X_i \leftarrow \lceil X_i \rceil$ 
  for  $i \in 1 \dots k$  do
    for  $j \in 1 \dots (10 \log n / \epsilon + X_i)$  do
       $r' \leftarrow \text{set\_type}(\text{dummy}, i)$ 
       $D.\text{append}(r')$ 
    end for
  end for
   $L \leftarrow 10k \cdot \log n / \epsilon - \sum_{i=1}^k X_i$ 
  for  $i \in 1 \dots L$  do
     $r' \leftarrow \text{set\_type}(\text{dummy}, k + 1)$ 
     $D.\text{append}(r')$ 
  end for
return  $D$ 

```

---

**Lemma 4.2.** *Our algorithm is  $(\epsilon, \frac{1}{n^2})$ -differentially private with respect to the histogram output.*

*Proof.* We sketch the proof for completeness; see [18] for full proof of the lemma. If the sensitivity of a query is  $\Delta$ , then we know that the Laplace mechanism, which adds a noise drawn from the distribution  $\text{Lap}(\frac{\Delta}{\epsilon})$ , is  $(\epsilon, 0)$  differentially private. Since the histogram query has sensitivity of 2,  $\text{Lap}(\frac{2}{\epsilon})$  is  $(\epsilon, 0)$ . However, our algorithm outputs the actual histogram without any noise if  $\max_i |X_i| > \frac{10 \log n}{\epsilon}$ , which we argued in Lemma 4.1 happens with probability at most  $\frac{1}{n^2}$ . Therefore, our algorithm is  $(\epsilon, \frac{1}{n^2})$ -differentially private.  $\square$

In the second step we show that memory access patterns of our algorithm satisfies oblivious differential privacy. That is, given any memory access pattern  $s$  produced by our algorithm and two neighboring databases  $D_1$  and  $D_2$  we need to show

$$\Pr[\mathcal{A}(D_1) \rightarrow s] \leq e^\epsilon \cdot \Pr[\mathcal{A}(D_2) \rightarrow s] + \delta$$

To prove this claim we need to set up some notation. Recall that  $\hat{D}$  denotes the augmented database with dummy records and fake records. We define *layout* as the actual mapping of the database  $\hat{D}$  to the array  $a[1], a[2], \dots, a[T]$ . We denote the set of all layouts by  $\mathcal{L}$ ; clearly,  $|\mathcal{L}| = T!$ , where we assume that dummy/fake records also have unique identifiers. Associated with every layout  $\ell$  is a *configuration*  $c(\ell)$ . A configuration  $c(\ell)$  is a  $T$ -dimensional vector from the set  $\{1, 2, \dots, k + 1\}^T$ . The  $j$ th coordinate of  $c(\ell)$ , denoted by  $c_j(\ell)$ , simply denotes a type  $i \in [k + 1]$ . Recall that dummy records are of type  $k + 1$ . We extend the histogram function to the augmented database in a natural fashion:  $h(\hat{D}) := (n_1, n_2, \dots, n_{k+1})$ , where  $n_i$  denotes the number of records of type  $i$  in  $h(\hat{D})$ . The shuffle operation guarantees that mapping of the database  $\hat{D}$  to the array  $a[1, 2, \dots, T]$  is uniformly distributed; that is, it picks a layout  $\ell \in \mathcal{L}$  uniformly at random.

Given a layout  $\ell \in \mathcal{L}$ , the memory access pattern produced by our algorithm is completely deterministic. Furthermore, observe that any two layouts  $\ell_1, \ell_2 \in \mathcal{L}$ , which have the same configuration  $c$ , lead to same access pattern. Both these statements follow directly from the definition of our algorithm, and the fact that public accesses to the array  $b[1, 2, \dots, k]$  depend only on the type of the records.

Thus, we get the following simple observation.

**Proposition 4.1.** *The mapping  $q : \mathcal{C} \rightarrow \mathcal{S}$  is a one-to-one mapping between the set of all configurations  $\mathcal{C}$  and the set of all access patterns produced by our algorithm  $\mathcal{S}$ .*

Going forward, we will concern ourselves only with the distribution produced by our algorithm on  $\mathcal{C}$  rather than the set  $\mathcal{S}$ . We will argue that for any two neighboring databases, the probability mass

our algorithm puts on a given configuration  $c \in \mathcal{C}$  satisfies the definition of DP. The configuration produced by our algorithm depends only on two factors: a) The histogram  $h(\hat{D})$  of the augmented database  $\hat{D}$  b) Output of the shuffle operation. Up to permutations, a configuration  $c$  output by our algorithm is completely determined by the histogram  $h(\hat{D})$  produced by our algorithm, which is a random variable. Let  $g : \mathbb{R}^{k+1} \rightarrow \{1, 2, \dots, k+1\}^T$  denote the mapping from all possible histograms on augmented databases to all possible configurations of length  $T$ . However, given a layout of the database  $\hat{D}$ , the shuffle operation produces a random permutation of the records of the database. This immediately implies the following lemma.

**Lemma 4.3.** *Fix a configuration  $c \in \{1, 2, \dots, k+1\}^T$ . Then,*

$$\frac{\Pr[\mathcal{A}(D_1) \in c]}{\Pr[\mathcal{A}(D_2) \in c]} = \frac{\Pr[\mathcal{A}(D_1) \in g^{-1}(c)]}{\Pr[\mathcal{A}(D_2) \in g^{-1}(c)]}$$

The lemma implies that to show that access patterns produced by our algorithm is  $(\epsilon, \delta)$ -ODP it is enough to show that distribution on the set of all histograms  $\mathbb{R}^{k+1}$  satisfies  $(\epsilon, \delta)$ -DP. However, the number of dummy elements  $n_{k+1}$  in any histogram  $h \in \mathbb{R}^{k+1}$  output by our algorithm is completely determined by the random variables  $X_1, X_2, \dots, X_k$ . Hence it is enough to show that distribution on the set of all histograms on the first  $k$  types satisfies  $(\epsilon, \delta)$ -DP, which we already argued in Lemma 1 is  $(\epsilon, \delta)$ -DP.

Now we have all the components to prove the main Theorem 4.4.

*Proof of Theorem 4.4.* For any histogram  $h \in \mathbb{R}^{k+1}$ , let  $\text{truncated}(h)$  denote the histogram restricted to the first  $k$  elements. Consider any neighboring databases  $D$  and  $D'$  and fix a memory access pattern  $s$  produced by our algorithm. From Lemma 4.3 and Proposition 4.1 we have that

$$\frac{\Pr[\mathcal{A}(D) \rightarrow s]}{\Pr[\mathcal{A}(D') \rightarrow s]} = \frac{\Pr[\mathcal{A}(D) \rightarrow g^{-1}(q^{-1}(s))]}{\Pr[\mathcal{A}(D') \rightarrow g^{-1}(q^{-1}(s))]}.$$

Since the number of dummy records is completely determined by the random variables  $X_1, X_2, \dots, X_k$ , we have

$$\frac{\Pr[\mathcal{A}(D) \rightarrow g^{-1}(q^{-1}(s))]}{\Pr[\mathcal{A}(D') \rightarrow g^{-1}(q^{-1}(s))]} = \frac{\Pr[\mathcal{A}(D) \rightarrow \text{truncated}(g^{-1}(q^{-1}(s)))]}{\Pr[\mathcal{A}(D') \rightarrow \text{truncated}(g^{-1}(q^{-1}(s)))]}$$

which is  $(\epsilon, 1/n^2)$ -DP from Lemma 4.2. Note that our overall mechanism is  $(\epsilon, 1/n^2)$ -ODP, since the memory access patterns can be used to construct the histogram output produced by our algorithm. Therefore, we do not lose the privacy budget of  $2\epsilon$ .

Let us focus on showing that  $\max_i |\hat{n}_i - n_i| \leq \log(\frac{k}{\theta}) \cdot \frac{2}{\epsilon}$  with probability at least  $1 - \theta$ . Consider,

$$\Pr \left[ \max_i |\hat{n}_i - n_i| \geq \log(k/\theta) \cdot \frac{2}{\epsilon} \right] \leq \sum_{i=1}^k \Pr \left[ |\hat{n}_i - n_i| \geq \log(k/\theta) \cdot \frac{2}{\epsilon} \right] \leq k \cdot \frac{\theta}{k} \leq \theta. \quad (1)$$

Now it only remains to bound the running time of the algorithm. First observe that the size of the augmented database is precisely  $T$ . The shuffle operation takes  $O(T \log T / \log \log T)$  and the histogram construction takes precisely  $T + k$  time. Therefore the total running time is  $O(\tilde{n} \log \tilde{n} / \log \log \tilde{n})$  where  $\tilde{n} = \max(n, k \log n / \epsilon)$ .  $\square$

### 4.3 Private Heavy Hitters

As a final example, we consider the problem of finding frequent items, also called the heavy hitters problem, while satisfying the ODP definition. In this problem, we are given a database  $D$  of  $n$  users, where each user holds an item from the set  $\{1, 2, \dots, m\}$ . In typical applications such as finding the most frequent websites or finding the most frequently used words in a text corpus,  $m$  is usually much larger than  $n$ . Hence reporting the entire histogram on  $m$  items is not possible. In such applications, one is interested in the list of  $k$  most frequent items, where we define an item as frequent if it occurs

more than  $n/k$  times in the database. In typical applications,  $k$  is assumed to be a constant or sublinear in  $n$ . The problem of finding the heavy hitters is one of the most widely studied problems in the LDP setting [5, 3]. In this section, we show that the heavy hitters problem becomes very simple in our model and achieves substantially better accuracy than in the local model. Let  $\hat{n}_i$  denote the number of occurrences of the item  $i$  output by our algorithm and  $n_i$  denotes the true count.

**Theorem 4.5.** *Let  $\tau > 1$  be some constant, and let  $\epsilon > 0$  be the privacy parameter. Suppose  $n/k > \frac{\tau}{\epsilon} \log m$ . Then, there exists a  $(\epsilon, \frac{1}{m^{\tau-1}})$ -ODP algorithm for the problem of finding the top  $k$  most frequent items that runs in time  $O(n \log n)$ . Furthermore, for every item  $i$  output by our algorithm it holds that i) with probability at least  $(1 - \theta)$ ,  $|\hat{n}_i - n_i| \leq \log(m/\theta) \cdot \frac{2}{\epsilon}$  and ii)  $n_i \geq n/k - \log(m/\theta) \cdot \frac{2}{\epsilon}$ .*

We remark that the  $k$  items output by an ODP-algorithm do not exactly correspond to the top  $k$  items due to the additive error introduced by the algorithm. We can use the above theorem to return a list of *approximate heavy hitters*, which satisfies the following guarantees: 1) Every item with frequency higher than  $n/k$  is in the list. 2) No item with frequency less than  $n/k - 2 \log(m/\theta) \cdot \frac{2}{\epsilon}$  is in the list.

We contrast the bound of this theorem with the bound one can obtain in the LDP setting. An *optimal* LDP algorithm can only achieve a guarantee of  $|\hat{n}_i - n_i| \leq \sqrt{n \cdot \log(n/\theta)} \cdot \frac{\log m}{\epsilon}$ . We refer the reader to [5, 3] for more details about the heavy hitters problem in the LDP setting. For many applications such as text mining, finding most frequently visited websites within a sub-population, this difference in the error turns out to be significant. See experiments and details in [8].

Our algorithm for Theorem 4.5 proceeds as follows: It sorts the elements in the database by type using oblivious sorting. It then initialises an encrypted list  $b$  and fills it in while scanning the sorted database as follows. It reads the first element and saves in private memory its type, say  $i$ , and creates a counter set to 1. It then appends to  $b$  a tuple: type  $i$  and the counter value. When reading the second database element, it compares its type, say  $i'$ , to  $i$ . If  $i = i'$ , it increments the counter. If  $i \neq i'$ , it resets the counter to 1 and overwrites the type saved in private memory to  $i'$ . In both cases, it then appends to  $b$  another tuple: the type and the counter from private memory. It proceeds in this manner for the rest of the database. Once finished, it makes a backward scan over  $b$ . For every new type it encounters, it adds  $\text{Lap}(2/\epsilon)$  to the corresponding counter and, additionally, extends the tuple with a flag set to 0. For all other tuples, a flag set to 1 is added instead. It then sorts  $b$ : by the flag in ascending order and by differentially-private counter values in descending order.

Let  $n^*$  be the number of distinct elements in the database. Then the first  $n^*$  tuples of  $b$  hold all the distinct types of the database together with their differentially-private frequencies. Since these elements are sorted, one can make a pass, returning the types of the top  $k$  most frequent items with the highest count (which includes the Laplace noise). Although this algorithm is not  $(\epsilon, 0)$ -ODP, it is easy to show that it is  $(\epsilon, \frac{1}{m^{\tau-1}})$ -ODP when  $n/k > \tau \log m$ , which is the case in all applications of the heavy hitters. Indeed, in most typical applications  $k$  is a constant. We are now ready to prove the above theorem.

*Proof of Theorem 4.5.* The running time of the algorithm is dominated by oblivious sorting and is  $O(n \log n)$  if using AKS sorting network. The algorithm accesses the database and list  $b$  independent of the data, hence, the accesses are also obviously differentially private. In the rest of the proof, we show that the output of the algorithm is differential-private.

Fix any two neighboring databases  $D$  and  $D'$ . Suppose  $\text{distinct}(D)$ ,  $\text{distinct}(D')$  denote the number of distinct items that appear in databases  $D$  and  $D'$ . If  $\text{distinct}(D) = \text{distinct}(D')$ , then the distinct items appearing in the two databases should be the same as  $D$  and  $D'$  differ only in 1 row. In this case, the histogram constructed on the distinct items by our algorithm is differentially private since it is an oblivious implementation of the simple private histogram algorithm [18]. Any post-processing done on top of a differentially private output, such as reporting only top  $k$  items does not violate the DP guarantee. Hence our overall algorithm is DP.

Now consider the case when  $\text{distinct}(D) \neq \text{distinct}(D')$ . In this case, let  $i^*$  denote the item that appears in  $D'$  but not in  $D$ . Clearly,  $n_{i^*} = 1$  in  $D'$  and  $n_{i^*} = 0$  in  $D$ . Let  $I$  denote the set of items that are common in the datasets  $D$  and  $D'$ . If one restricts the histogram output to the set  $I$ , it satisfies the guarantees of DP. However, our algorithm never reports  $i^*$  in the list of heavy hitters on the database  $D$ . On the other hand, there is a non-zero probability that our algorithm reports  $i^*$  in the list of top  $k$  items. This happens if the Laplace noise added to  $i^*$  is greater than  $n/k \geq \frac{\tau}{\epsilon} \log m$ , which

occurs with the probability at most  $\frac{1}{m^r}$ . Since there are at most  $m$  items, by the union bound the probability of this event happening is  $\frac{1}{m^{r-1}}$ .  $\square$

**Frequency Oracle Based on Count-Min Sketch** Another commonly studied problem in the context of heavy hitters is the frequency oracle problem. Here, the goal is to answer the number of occurrences of an item  $i$  in the database. While this problem can be solved by computing the answer upon receiving a query and adding Laplace noise, there is a simpler approach which might be sufficient in many applications. One can maintain a count-min sketch, a commonly used algorithm in the streaming literature [12], of the frequencies of items by making a single pass over the data. An interesting aspect of this approach is that entire sketch can be maintained in the private memory, hence one does not need to worry about obliviousness. Further, entire count-min sketch can be released to the data collector by adding Laplace noise. An advantage of this approach is that the data collector can get the frequencies of any item he wants by simply referring to the sketch, instead of consulting the DP algorithm. It would be interesting to find more applications of the streaming techniques in the context of ODP algorithms.

## References

- [1] Miklós Ajtai, János Komlós, and Endre Szemerédi. An  $O(n \log n)$  sorting network. In *ACM Symposium on Theory of Computing (STOC)*, 1983.
- [2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
- [3] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Guha Thakurta. Practical locally private heavy hitters. In *Conference on Neural Information Processing Systems (NIPS)*, pages 2285–2293, 2017.
- [4] Raef Bassily and Adam D. Smith. Local, private, efficient protocols for succinct histograms. In *ACM Symposium on Theory of Computing (STOC)*, pages 127–135, 2015.
- [5] Raef Bassily and Adam D. Smith. Local, private, efficient protocols for succinct histograms. In *ACM Symposium on Theory of Computing (STOC)*, pages 127–135, 2015.
- [6] Kenneth E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conf.*, 1968.
- [7] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 2005.
- [8] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.
- [9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.
- [10] Nicholas Carlini, Chang Liu, Jernej Kos, Úlfar Erlingsson, and Dawn Song. The secret sharer: Measuring unintended neural network memorization & extracting secrets. *CoRR*, abs/1802.08232, 2018.
- [11] T-H. Hubert Chan, Kai-Min Chung, Bruce Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. Cryptology ePrint Archive, Report 2017/1033, 2017. <https://eprint.iacr.org/2017/1033>.
- [12] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005.
- [13] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.
- [14] Apple Differential Privacy Team. Learning with privacy at scale, 2017.
- [15] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. In *Conference on Neural Information Processing Systems (NIPS)*, 2017.

- [16] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)*, pages 265–284, 2006.
- [17] Cynthia Dwork, Moni Naor, Omer Reingold, Guy N. Rothblum, and Salil Vadhan. On the complexity of differentially private data release: Efficient algorithms and hardness results. In *ACM Symposium on Theory of Computing (STOC)*, pages 381–390, 2009.
- [18] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9, August 2014.
- [19] Úlfar Erlingsson, Vasyli Pihur, and Aleksandra Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1054–1067, 2014.
- [20] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2), September 1985.
- [21] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3), 1996.
- [22] Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2011.
- [23] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2011.
- [24] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.
- [25] J. Gotzfried, M. Eckert, S. Schinzel, and T. Muller. Cache attacks on Intel SGX. In *European Workshop on System Security (EuroSec)*, 2017.
- [26] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
- [27] Noah Johnson, Joseph P. Near, and Dawn Song. Towards practical differential privacy for SQL queries. *PVLDB*, 11(5):526–539, January 2018.
- [28] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Symposium on Principles of Database Systems (PODS)*, pages 41–52, 2010.
- [29] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO*, 1996.
- [30] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.
- [31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [32] Sahar Mazloom and S. Dov Gordon. Differentially private access patterns in secure computation. Cryptology ePrint Archive, Report 2017/1016, 2017. <https://eprint.iacr.org/2017/1016>.
- [33] Ilya Mironov. On significance of the least significant bits for differential privacy. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
- [34] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2017.
- [35] S. Muthukrishnan. Data streams: algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1, 2003.
- [36] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 8573. Springer, 2014.

- [37] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, 2016.
- [38] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *RSA Conference Cryptographer’s Track (CT-RSA)*, 2006.
- [39] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *Cryptology ePrint Archive, Report 2002/169*, 2002.
- [40] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Cacheshuffle: An oblivious shuffle algorithm using caches. *CoRR*, abs/1705.07069, 2017.
- [41] Mike Paterson. Improved Sorting Networks with  $O(\log N)$  Depth. *Algorithmica*, 5(1):65–92, 1990.
- [42] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.
- [43] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Advances in Cryptology—CRYPTO*, 2010.
- [44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.
- [45] Aaron Roth and Tim Roughgarden. Interactive privacy via the median mechanism. In *ACM Symposium on Theory of Computing (STOC)*, pages 765–774, New York, NY, USA, 2010. ACM.
- [46] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [47] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clementine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to conceal cache attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.
- [48] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.
- [49] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.
- [50] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.
- [51] Sameer Wagh, Paul Cuff, and Prateek Mittal. Root ORAM: A tunable differentially private oblivious RAM. *CoRR*, abs/1601.03378, 2016.
- [52] Peter Williams, Radu Sion, and Bogdan Carbutar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.
- [53] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [54] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

## A Cloud Computing Setting

If the framework is hosted on the cloud, we also consider a second adversary  $A_C$  who is hosting the framework (e.g., malicious cloud administrator or co-tenants). Since  $A_C$  has access to the infrastructure of the framework its adversarial capabilities are the same as those of the adversarial data collector  $A_D$  in §1. However, in this case, in addition to protecting user data from  $A_D$ , the result of the computation also needs to be protected. We note that if  $A_D$  and  $A_C$  are colluding, then the cloud scenario is equivalent to the on-premise one. If they are not colluding,  $A_C$  may still be able to gain access to the framework where its outsourced computation is performed. It can do so either by using malware or as a malicious co-tenant [44].

In this setting, our framework remains as described in §3 with the exception that algorithms that run inside of the TEE need to be data-oblivious per Definition 2.2 in order to hide the output from  $A_C$ . Since Definition 2.2 is stronger than Definition 3.1 when output is not revealed, user data is also protected from  $A_D$ .

## B ORAM-based Histogram Algorithm

In this section we outline a standard differentially private algorithm for histogram computation and its ORAM-based transformation to achieve ODP. We note that the algorithm in §4.2 is more efficient than this transformation for large values of  $k$ .

Algorithm 3 is the standard differentially private algorithm from [18]. It is not data-oblivious since accesses to the histogram depend on the content of the database and reveal which records have the same type. (See Figure 1 (right) for a visualization.)

Algorithm 4 is a data-oblivious version of Algorithm 3. It uses oblivious RAM (see §2.2) to hide accesses to the histogram. In particular, we use  $\text{ORAM}(b)$  to denote the algorithm that returns a data-oblivious data structure  $\tilde{b}$  initialized with an array  $b$ .  $\tilde{b}$  supports queries  $\tilde{b}.\text{read}(i)$  and  $\tilde{b}.\text{write}(i, \text{data}_i)$  for  $i \in [1, k]$ . That is, it returns  $b.\text{read}(i)$  (or similarly stores  $\text{data}_i$  in the  $i$ th index of  $b$  with write) while hiding  $i$ . The performance of the resulting algorithm depends on the underlying oblivious RAM construction. For example, if we use the scheme by Kushilevitz *et al.* [30] (that relies on AKS sort), it makes  $O(n(\log k)^2 / (\log \log k))$  accesses to external memory, or  $O(n(\log k)^2)$  if we use Path ORAM [50] that has much smaller constants.

The histogram problem can be solved also using two oblivious sorts, similar to the heavy hitters algorithm described in §4.3 but returning all  $n^*$  types and their counters since  $k = n^*$  when histogram support is known. The overhead of this method is the overhead of the underlying sorting algorithm.

---

**Algorithm 3** DP histogram [18]  
 $\mathcal{M}_{\text{hist}}^{\text{DP}}(D, k)$

---

```

 $b \leftarrow \{0\}^k$ 

for  $r \in D$  do
   $i \leftarrow \text{get\_type}(r)$ 
   $b_i \leftarrow b_i + 1$ 
end for
for  $i \in 1 \dots k$  do
   $\hat{b}_i \leftarrow b_i + \text{Lap}(\frac{2}{\epsilon})$ 
end for
return  $\hat{b}$ 

```

---



---

**Algorithm 4** ORAM-based DP histogram  
 $\mathcal{M}_{\text{hist}}^{\text{ODP}}(D, k)$

---

```

 $b \leftarrow \{0\}^k$ 
 $\tilde{b} \leftarrow \text{ORAM}(b)$ 
for  $r \in D$  do
   $i \leftarrow \text{get\_type}(r)$ 
   $b_i \leftarrow \tilde{b}.\text{read}(i)$ 
   $\tilde{b}.\text{write}(i, b_i + 1)$ 
end for
for  $i \in 1 \dots k$  do
   $\hat{b}_i \leftarrow \tilde{b}.\text{read}(i) + \text{Lap}(\frac{2}{\epsilon})$ 
end for
return  $\hat{b}$ 

```

---

## C Comparison of Data-Oblivious Algorithms

Table 1: Asymptotical performance of several data-oblivious algorithms on arrays of  $n$  records,  $c \geq 2$  is a constant,  $m$  is the size of private memory,  $k$  is the number of types in a histogram, and  $\tilde{n} = \max(n, k \log n/\epsilon)$ .

	<b>Algorithm</b>	<b>Private Memory (<math>m</math>)</b>	<b>Overhead</b>
RAM	Path ORAM [50]	$\omega(\log n)$	$(\log n)^2$
	Kushilevitz <i>et al.</i> [30]	1	$\frac{(\log n)^2}{\log \log n}$
Sort	AKS Sort [1]	1	$n \log n$
	Batcher's Sort [6]	1	$n(\log n)^2$
Shuffle	Melbourne Shuffle [36, 40]	$\sqrt[c]{n}$	$cn$
		$\omega(\log n)$	$n \frac{\log n}{\log m}$
Histogram	In private memory, $k = O(m)$	$k$	$n$
	ORAM-based (Algorithm 4)	1	$n \frac{(\log k)^2}{\log \log k}$
	Oblivious Sort-based (§B)	1	$n \log n$
	ODP Histogram (§4.2, Algorithm 1)	$\omega(\log n)$	$\tilde{n} \frac{\log \tilde{n}}{\log \log \tilde{n}}$