Differentially Private Aggregation of Distributed Time-Series with Transformation and Encryption

Vibhor Rastogi Suman Nath

Abstract

We propose the first differentially private aggregation algorithm for distributed time-series data that offers good practical utility without any trusted server. This addresses two important challenges in participatory data-mining applications where (i) individual users wish to publish temporally correlated time-series data (such as location traces, web history, personal health data), and (ii) an untrusted third-party aggregator wishes to run aggregate queries on the data.

To ensure differential privacy for time-series data despite the presence of temporal correlation, we propose the Fourier Perturbation Algorithm (FPA_k). Standard differential privacy techniques perform poorly for time-series data. To answer n queries, such techniques can result in a noise of $\Theta(n)$ to each query answer, making the answers practically useless if n is large. Our FPA_k algorithm perturbs the Discrete Fourier Transform of the query answers. For answering n queries, FPA_k improves the expected error from $\Theta(n)$ to roughly $\Theta(k)$ where k is the number of Fourier coefficients that can (approximately) reconstruct all the n query answers. Our experiments show that $k \ll n$ for many real-life data-sets resulting in a huge error-improvement for FPA_k.

To deal with the absence of a trusted central server, we propose the Distributed Laplace Perturbation Algorithm (DLPA) to add noise in a distributed way in order to guarantee differential privacy. To the best of our knowledge, DLPA is the first distributed differentially private algorithm that can scale with a large number of users: DLPA outperforms the only other distributed solution for differential privacy proposed so far, by reducing the computational load per user from $O(U)$ to $O(1)$ where U is the number of users.

1 Introduction

The ever increasing instrumentation of the physical and the virtual world has given us an unprecedented opportunity to collect useful data from diverse sources and to mine it for understanding important phenomena. Consider the following examples of *participatory data mining applications*.

E1: In participatory sensing applications such as CarTel [16], BikeNet [8], PEIR [26], WeatherUnderground[1], participants contribute various time-series data, e.g., their current locations, speeds, weather information, images, etc. These data can be aggregated and mined for useful information such as community interests (e.g., popular places), congestion patterns in roads, micro-weather, etc.

E2: A Web browser can install plug-ins to monitor users' browsing behaviors such as the numbers of different types of web pages a user visits, the types of products he buys from online stores, etc. Historical data from such plug-ins can be aggregated to understand user behaviors for improving search results or for better targeted advertisement delivery [14].

E3: Health-care applications such as Microsoft HealthVault[2] and Google Health[3] allow users to periodically upload data such as their weights, occurrences of diseases, amounts of exercise, food and drug intake, etc. PIER [26] allows individual users to store such data locally in *personal data vaults*. Such data can be mined in combination to understand disease outbreaks, distribution of weights, relationship of weight gain and drug intake, etc.

[1] `http://www.weatherunderground.com`
[2] `http://www.healthvault.com`
[3] `https://www.google.com/health`

	Centralized	Distributed
Relational data	**E.g., [7, 15, 28]** *Inaccurate for long time-series query sequences*	**E.g., [6, 9, 12, 29]** *Either inefficient or inaccurate [9] for a large number of users*
Time-series data	**E.g., [11, 22]** *No formal privacy guarantees*	**This paper** *Accurate & efficient answers for time-series query sequences under differential privacy*

Table 1: The design space and existing works

As the above examples suggest, aggregate statistics computed from data contributed by a large number of individual participants can be quite useful. However, data owners or publishers may not be always willing to reveal the true values due to various reasons, most notably privacy considerations. The goal of our work is to enable third parties to compute useful aggregates over, while guaranteeing the privacy of, the data from individual publishers.

A widely employed and accepted approach for partial information hiding is based on random perturbation, which introduces uncertainty about individual values. Prior work has shown how to perturb relational data for useful aggregation [7, 9, 20, 27]. However, participatory data mining applications have two unique characteristics that make existing privacy mechanisms inadequate for these applications.

- **Time-series data:** The applications generate time series numerical or categorical data. Data at successive timestamps from the same source can be highly correlated.
- **Distributed sources:** Data publishers may not trust any single third party to see their true data. This means, the querier needs to be able to compute useful aggregates without seeing the true data values.

The above characteristics make most existing privacy solutions, which assume relational data with negligible correlations across tuples [9, 20, 27] or existence of a central trusted entity for carefully introducing noise [7, 15, 28], inadequate for our target applications (as summarized in Table 1).

Thus, to realize widespread adoption of participatory data mining applications, the first challenge that one needs to address is to ensure privacy for time-series data. One factor that makes it challenging is the strong correlation among successive values in the series. This correlation makes answers to different queries over time-series data to also become correlated, e.g. a sequence of queries computing the average weight of a community at successive weeks.

One possible way for achieving privacy is to perturb the answers to such queries independently of one another, thereby ensuring that even revealing a few true answers does not help infer anything about the perturbation of other answers. However, [22] pointed out that if the time-series exhibit certain patterns, then independent perturbation of query answers can be distinguished from the original answers and filtered out. Authors in [11, 22] consider perturbing time series data to defend against several privacy attacks, but they do not provide any formal privacy guarantee, without which data owners may not publish sensitive data in the fear of unforeseen privacy attacks.

On the other hand, formal privacy guarantees like differential privacy that work well for relational data, seem too hard to achieve for time series data. For instance, standard differentially private techniques [7] can result in a noise of $\Theta(n)$ to each query answer, where n is the number of queries to answer, making the query answers practically useless if a long sequence of queries is to be answered.

The second challenge arises from the absence of a trusted aggregator. Most previous works assume that a trusted aggregator, who has access to the raw data, computes target functions on the data and then perturbs the results [7, 20, 27]. In the absence of a trusted aggregator, users need to perturb their data before publishing it to the aggregator [9]. However, if users perturb data independently, the noise variance in the perturbed estimate grows linearly with the number of users, reducing the utility of the aggregate information. To improve utility, cryptographic techniques like Secure Multiparty Computation [6, 12, 29] can be used to compute accurate perturbed estimates in a distributed setting. However the computational performance of such cryptographic techniques does not scale well with a large number of users.

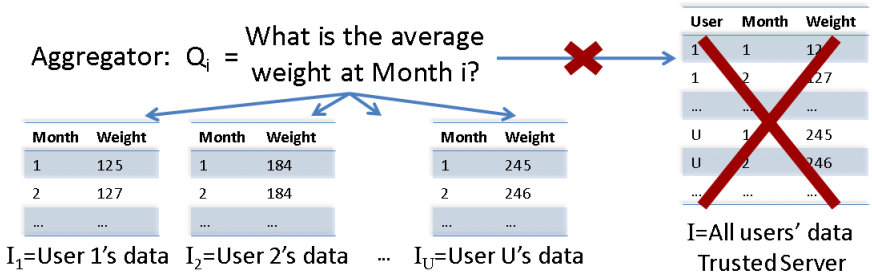


Figure 1: System Model (U users with data $I_1, \ldots, I_U$. The aggregator issues recurring query $\mathbf{Q} = \mathrm{Q}_1, \ldots, \mathrm{Q}_n$. No trusted server has $I = I_1 \cup I_2 \ldots \cup I_U$ to evaluate $\mathbf{Q}(I)$)

In this paper, we address these two challenges of distributed time-series data. We use the state-of-the-art differential privacy as the privacy requirement, and make the following contributions:

► To answer multiple queries over time-series data under differential privacy, we propose the FPA_k algorithm that perturbs the Discrete Fourier Transform (DFT) of the query answers. For answering n queries, FPA_k improves the error from $\Theta(n)$ (error of standard differential privacy techniques) to roughly $\Theta(k)$ where k is the number of DFT coefficients that can (approximately) reconstruct all the n query answers. Our experiments show that a small $k \ll n$ is sufficient for many real-life datasets, resulting in a huge error-improvement for FPA_k. To the best of our knowledge, FPA_k is the first differentially private technique (unlike [11, 22]) that offers practical utility for time-series data.

► We propose the DLPA algorithm that adds noise in a distributed way for providing differential privacy. To the best of our knowledge, DLPA is the first distributed differentially private algorithm that scales with a large number of users: DLPA outperforms the only other proposed distributed algorithm [6], by reducing the computational load per user from $O(U)$ to $O(1)$ where U is the number of users.

► Our distributed solution combines the FPA_k and DLPA algorithms to get the accuracy benefits of the former and the scalability of the latter. We empirically evaluate our solution over three real time-series datasets, namely, *GPS traces*, *daily body-weight readings*, and *traffic volumes*. Our experiments show that our solution improves accuracy of query answers by orders of magnitude and also scales well with a large number of users.

We believe that our work is an important first step towards practical participatory data mining applications. We have implemented some of our techniques in a real online participatory sensing application, which has been publicly available for last three years with several hundreds data publishers. Our privacy techniques now allow users to publish private data without revealing the true values and our application to compute useful aggregates over private data.

2 Preliminaries

2.1 Problem Setup and System Model

Motivated by the participatory applications in Section 1, we consider a system model as shown in Figure 1. The system has two types of parties involved: a set of U *users* and an *aggregator*. The figure shows U users locally storing their personal weight time-series data. We will use the weight time-series as a running example throughout the paper. In general, we model each user u's data as a (uni- or multi-variate) time series data, and denote it as I_u. We also denote $I = I_1 \cup I_2 \ldots \cup I_U$ the combined time-series data of all users. There is no trusted central server, and hence I is never computed. The aggregator, however, wishes to compute aggregate queries over I.

Types of queries. An aggregate query can be a *snapshot* query that returns a single number, e.g., the current average weight over all users. A query can also be *recurring* and ask for periodic answers, e.g.,

the average weight of all users computed once every month of the year. We model a recurring query as a sequence of queries $\mathbf{Q} = \{\mathrm{Q}_1, \ldots, \mathrm{Q}_n\}$ where each Q_i is a snapshot query. We denote $\mathrm{Q}_i(I)$ the value of the snapshot query Q_i on input I, and $\mathbf{Q}(I)$ the vector $\{\mathrm{Q}_1(I), \ldots, \mathrm{Q}_n(I)\}$.

A recurring query can be *historical* that focuses on past data only, or *real-time* that runs on data as it arrives. The data-mining applications we consider are primarily concerned with historical recurring queries, and we develop new algorithms to answer them accurately. Real-time queries are supported only as a sequence of snapshot queries.

Distributed Computation. If a query Q contains predicates on public static attributes, it can be forwarded only to users satisfying the predicates. Otherwise, Q needs to be forwarded to all users. Upon receiving the query Q, a user evaluates Q on his own time series, I_u, perturbs the result, and sends the perturbed results back to the aggregator. The aggregator combines the perturbed results from all users to produce the final result.

A prerequisite for such a distributed system is that the true query answer, $\mathrm{Q}(I)$, is computable distributedly. Of all such queries, we consider only queries of the general form $\mathrm{Q}(I) = \sum_u f_u(I_u)$, where f_u is an arbitrary function that maps user u's data, I_u, to numbers.[4] Such queries, called aggregate-sum queries, are quite general, and as explained in [3], are powerful tools for learning and statistical analysis: many algorithms like correlation, PCA, SVD, decision tree classification, etc. can be implemented using only aggregate-sum queries as a primitive. Queries not included in this class are queries that require a non-sum function to be evaluated collectively over multiple users' data (e.g., aggregate-max or min queries).

Attack Model. We allow both users and the aggregator to be malicious. A malicious user can be of two kinds: (i) Liar: a user who lies about his values, but otherwise follows the protocol correctly, or (ii) Breaker: a user who breaks the protocol, e.g., sends wrong messages. A malicious aggregator can break the protocol. In addition, it can collude with other malicious users.

To ensure privacy for users, we make a flexible assumption that at least a fraction of users (e.g., a majority) are honest. The lower bound h of the number of honest users is known a priori during deciding the noise generation parameters of the system. Remaining users and the aggregator can be arbitrarily malicious. (Similar assumption is generally made in cryptographic solutions.) The assumption is practical and flexible—for a user to know that his true values will remain private, he only needs to know that at least a certain fraction of other users are honest; he does not need to trust any single entity s.a. the aggregator.

On the other hand, to ensure a good utility guarantee for the aggregator, we assume that the aggregator queries a set of users that it generally trusts. Of the users the aggregator chooses to query, there can be at most l liars (l is small) and the remaining users are either honest or colluding/collaborating with the aggregator. There is fundamentally no way to ensure good utility if a large number of users lie about their data.The same is true if even a single user introduces an arbitrarily large noise. So we assume that there are no breakers: in practice, this can be arranged by ensuring that users' messages sent to the aggregator are generated and digitally signed by a trusted software implementing the protocol.[5]

In summary, our privacy guarantees hold even if a large number of users are malicious. This is crucial to make new privacy-aware users feel comfortable to join the system. Our utility guarantees hold if a small ($< l$) number of users lie and try to disrupt the final aggregate. This leaves the responsibility to the aggregator for choosing a good set of users to query. For example, if the aggregator can identify a malicious user (e.g., via some out-of-band mechanism), it can blacklist the user and exclude him from its queries. Our attack model is stronger than many previous techniques [12] that use the honest-but-curious model and disallow malicious agents.

Privacy Goals. We aim to enable the aggregator to estimate answers to aggregate queries. At the same time, an aggregator should not learn anything more, other than the aggregate answer, about honest individual users. Moreover, no user should learn anything about the values of other honest users, even if he colludes with the aggregator or other malicious users. We formalize our privacy requirement using differential privacy, which we discuss next.

[4] Our techniques can support arbitrary queries if run on a centralized server.

[5] Many software security systems rely on trusted software, for instance the clients' antivirus software, to work with untrusted clients [14].

2.2 Privacy Definition and Background

We now discuss differential privacy [5] that we use as our privacy definition and also review the standard perturbation technique [7] used for achieving differential privacy.

Differential Privacy. Informally, an algorithm is differentially private if its output is insensitive to changes in the data I_u of any single user u. This provides privacy because if similar databases, say differing in the data of one user, produce indistinguishable outputs, then the adversary cannot use the output to infer any single user's data. To formalize this notion, denote $I = I_1 \cup I_2 \ldots \cup I_U$ the combined data form U users and $nbrs(I)$ the data obtained from adding/removing one user's data from I, i.e. $nbrs(I)$ consists of I' such that either $I' = I \cup I_u$ for $u \notin \{1, \ldots, U\}$ or $I' = I - I_u$ for some $u \in \{1, \ldots, U\}$.

Definition 2.1 (ϵ-differential privacy [5]). *Denote $A(I)$ the output of an algorithm A on input data I. Then A is ϵ-differentially private if all I, I' such that $I' \in nbrs(I)$, and any output x, the following holds:*

$$Pr[A(I) = x] \leq e^{\epsilon} Pr[A(I') = x]$$

where Pr is a probability distribution over the randomness of the algorithm.

Query Sensitivity. We now look at a standard technique proposed by Dwork et al. [7] for differential privacy. It can be used to answer any query whether it is just a single snapshot query, Q_i, or a recurring query sequence, $\mathbf{Q} = \mathrm{Q}_1, \ldots, \mathrm{Q}_n$.

The technique works by adding random noise to the answers, where the noise distribution is carefully calibrated to the query. The calibration depends on the query *sensitivity*—informally, the maximum amount the query answers can change given any change to a single user's data I_u. If $\mathbf{Q}$ is a query sequence, $\mathbf{Q}(I)$ and $\mathbf{Q}(I')$ are each vectors. Sensitivity then measures the distance between the two vectors. This is typically done using the L_1 distance metric, denoted as $|\mathbf{Q}(I) - \mathbf{Q}(I')|_1$, that measures the Manhattan distance $\sum_i |\mathrm{Q}_i(I) - \mathrm{Q}_i(I')|$ between these vectors. In this paper, we also use the L_2 distance metric, denoted as $|\mathbf{Q}(I) - \mathbf{Q}(I')|_2$ that measures the Euclidean distance $\sqrt{\sum_i (\mathrm{Q}_i(I) - \mathrm{Q}_i(I'))^2}$.

Definition 2.2 (Sensitivity [7]). *Let $\mathbf{Q}$ be any query sequence. For $p \in \{1, 2\}$, the L_p sensitivity of $\mathbf{Q}$, denoted $\Delta_p(\mathbf{Q})$, is the smallest number such that for all I and $I' \in nbrs(I)$,*

$$|\mathbf{Q}(I) - \mathbf{Q}(I')|_p \leq \Delta_p(\mathbf{Q})$$

For a single snapshot query Q_i, the L_1 and L_2 sensitivities are the same, and we write $\Delta(Q_i) = \Delta_1(Q_i) = \Delta_2(Q_i)$.

Example 2.1. *Consider a query Q counting the number of users whose weight in month 1 is greater than 200 lb. Then $\Delta(Q)$ is simply 1 as Q can differ by at most 1 on adding/removing a single user's data. Now consider $\mathbf{Q} = Q_1, \ldots, Q_n$, where Q_i counts users whose weight in month i is greater than 200 lb. Then $\Delta_1(\mathbf{Q})$ is n (for the pair I,I' which differ in a single user having weight > 200 in each month i) and $\Delta_2(\mathbf{Q}) = \sqrt{n}$ (for the same pair I,I').*

Laplace Perturbation Algorithm (LPA). To guarantee differential privacy in presence of a trusted server, [7] proposes the LPA algorithm that adds suitably-chosen noise to the true answers. The noise is generated according to the Laplace distribution. Denote $Lap(\lambda)$ a random variable drawn from the Laplace distribution with PDF: $Pr(Lap(\lambda) = Z) = \frac{1}{2\lambda} e^{-\epsilon|Z|/\lambda}$. $Lap(\lambda)$ has mean 0 and variance $2\lambda^2$. Also denote $\mathbf{Lap}^n(\lambda)$ to be a vector of n independent $Lap(\lambda)$ random variables.

The LPA algorithm takes as input a query sequence $\mathbf{Q}$ and parameter λ controlling the Laplace noise. LPA first computes the true answers, $\mathbf{Q}(I)$, exactly and then perturbs the answers by adding independent $Lap(\lambda)$ noise to each query answer in $\mathbf{Q}(I)$. More formally, it computes and outputs $\tilde{\mathbf{Q}} = \mathbf{Q}(I) + \mathbf{Lap}^n(\lambda)$. Differential privacy is guaranteed if the parameter λ of the Laplace noise is calibrated according to the L_1 sensitivity of $\mathbf{Q}$. The following theorem shown in [7] formalizes this intuition.

Theorem 2.1 (Privacy [7]). *LPA($\mathbf{Q}, \lambda$) is ϵ-differentially private for $\lambda = \Delta_1(\mathbf{Q})/\epsilon$.*

Analyzing accuracy. To analyze the accuracy of the perturbed estimates returned by an algorithm, we quantify their error in the following way.

Definition 2.3 (Utility)**.** *Let $A(\mathbf{Q})$ be an algorithm that returns a perturbed estimate $\tilde{\mathbf{Q}} = \tilde{Q}_1, \ldots, \tilde{Q}_n$ for an input sequence $\mathbf{Q} = Q_1, \ldots, Q_n$. We denote $error_i(A) = \mathbb{E}_A|\tilde{Q}_i - Q_i(I)|$ the expected error in the estimate of the i^{th} query Q_i. Here $\mathbb{E}_A$ is the expectation taken over the possible randomness of A. Also denote $error(A) = \sqrt{\sum_{i=1}^{n} error_i(A)^2}$ the total L_2 error between $\mathbf{Q}$ and $\tilde{\mathbf{Q}}$.*

For example, $error_i(\text{LPA}) = \mathbb{E}|\tilde{\text{Q}}_i - \text{Q}_i| = \mathbb{E}|Lap(\lambda)| = \lambda$. Next we discuss the utility of the LPA algorithm while satisfying ϵ-differential privacy.

Theorem 2.2 (Utility [7])**.** *Fix $\lambda = \Delta_1(\mathbf{Q})/\epsilon$ so that $LPA(\mathbf{Q}, \lambda)$ is ϵ-differentially private. Then for all $i \in \{1, \ldots, n\}$, $error_i(LPA) = \Delta_1(\mathbf{Q})/\epsilon$.*

Example 2.2. *Recall the recurring query $\mathbf{Q}$ of Eg. 2.1 that counts users having weight > 200 in each month $i = \{1, \ldots, n\}$. Then $\Delta_1(\mathbf{Q}) = n$ and LPA gives an $error_i(LPA) = n/\epsilon$ in each query Q_i for ϵ-differential privacy. Also, the total L_2 error is $error(LPA) = \sqrt{\sum_{i=1}^{n} n^2/\epsilon^2} = n^{3/2}/\epsilon$*

3 Sketch of our solution

Before discussing our solution for differential privacy over distributed time-series data, we provide an outline for it in this section. Our solution uses several existing primitives including Discrete Fourier Transform (DFT), homomorphic encryption (that allows aggregation of encrypted values without decryption), and threshold encryption (that requires a threshold number of users for decryption). We will review these techniques in Sections 4 and 5.

Our solution for answering a query sequence $\mathbf{Q}$ of n queries consists of the following two protocol stages. The first stage is a method to improve the accuracy of query answers and is described using a trusted central server. The second stage is then used to obtain a distributed solution.

1) Fourier Perturbation Algorithm (FPA_k). To answer $\mathbf{Q}$ with small error under differential privacy, we design the FPA_k algorithm. FPA_k is based on compressing the answers, $\mathbf{Q}(I)$, of the query sequence using an orthonormal transformation. Intuitively, this means finding a k-length query sequence $\mathbf{F^k} = \text{F}_1^k, \ldots, \text{F}_k^k$, where $k \ll n$, such that the answers, $\mathbf{F^k}(I)$, can be used to approximately compute $\mathbf{Q}(I)$. Then we can perturb $\mathbf{F^k}(I)$ instead of $\mathbf{Q}(I)$ using a lower noise (the noise actually reduces by a factor of n/k) while preserving differential privacy. An additional error creeps in since $\mathbf{F^k}(I)$ may not be able to reconstruct $\mathbf{Q}(I)$ exactly, but for the right choice of $\mathbf{F^k}$, this reconstruction error is significantly lower than the perturbation error caused by adding noise directly to $\mathbf{Q}(I)$. A good $\mathbf{F^k}$ can be found using any orthonormal transformation and we use the Discrete Fourier Transform (DFT) in our algorithm. How to perturb the DFT query sequence $\mathbf{F^k}$ to ensure differential privacy is an important challenge that distinguishes our solution from other Fourier-based perturbation approaches [22]. We discuss this in detail in Sec. 4.

2) Distributed LPA (DLPA). To answer an aggregate-sum query sequence $\mathbf{Q}$ distibutedly under differential privacy, we propose the DLPA algorithm, a distributed version of the LPA algorithm discussed in Sec. 2.2. Our complete solution comprises of using FPA_k for improving accuracy together with DLPA for distributed noise-addition. We describe how they are combined in a moment, but first we explain the DLPA algorithm. We explain DLPA for a single aggregate-sum query Q: the generalization to the sequence $\mathbf{Q}$ is straight-forward and just requires n separate invocations, once for each query Q_i in $\mathbf{Q}$. Since Q is an aggregate-sum query, $\text{Q}(I) = \sum_{u=1}^{U} f_u(I_u)$ where the function f_u maps user u's data to numbers. Denote $x_u = f_u(I_u)$, so that $\text{Q}(I) = \sum_{u=1}^{U} x_u$.

The basic protocol is based on threshold homomorphic encryption and is shown in Fig. 2. To perturb $\text{Q}(I) = \sum_{u=1}^{U} x_u$, each user u adds a share of noise, n_u, to his private value x_u. To keep the estimation error small, the noise shares are chosen such that $\sum_{u=1}^{U} n_u$ is sufficient for differential privacy, but n_u alone is not sufficient: thus the value $x_u + n_u$ can not directly be sent to the aggregator. To address this, the user

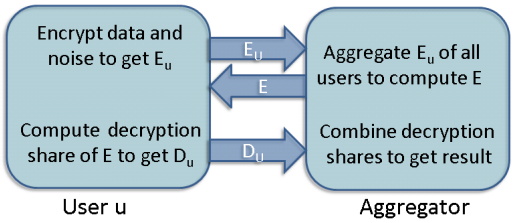


Figure 2: Basic Distributed Protocol (homomorphic property exploited to aggregate users' encryption & threshold property to combine users' decryption shares)

u computes encryptions of x_u and n_u and sends it to the aggregator. Due to encryption, the aggregator can not learn anything about x_u. However, using the *homomorphic* property, it can compute the encryption of the sum of all received values from all users thereby getting an encryption E of $\tilde{\mathrm{Q}} = \sum_u (x_u + n_u)$. This encryption is then sent back to all users who use the *threshold* property to compute their respective decryption shares. Finally, the decryption shares are sent to the aggregator who combines them to compute the final decryption. The end result is a noisy differentially private estimate $\tilde{\mathrm{Q}}$ of $\mathrm{Q}(I)$.

There are two challenges with the basic protocol described above. Firstly, the noise shares have to be generated in a way so that their sum is sufficient for differential privacy. Secondly, the aggregator can be malicious: the distributed protocol requires the aggregator to compute an encryption E of $\tilde{\mathrm{Q}} = \sum_u (x_u + n_u)$ and send a decryption request to the users to help him decrypt E. But, the aggregator can cheat and request the decryption of wrong values, for instance, the encrypted private value of a single user, in which case the users will be inadvertently decrypting the private value of that user. We discuss how to solve these challenges in Sec. 5.

Putting the two together. Now we explain how FPA_k and DLPA together give the complete solution. To answer a query sequence $\mathbf{Q}$ of n aggregate-sum queries, the aggregator first uses FPA_k to compute the k-length sequence $\mathbf{F^k}$. Due to the linearity of the DFT transformation, $\mathbf{F^k}$ is another aggregate-sum query sequence. Now FPA_k requires to perturb the answers of the query sequence $\mathbf{F^k}$ in order to get differential privacy. This is done by applying the DLPA algorithm on $\mathbf{F^k}$. The end result of DLPA is that the aggregator gets a noisy differentially estimate $\tilde{\mathbf{F}}^k$ of $\mathbf{F^k}(I)$. Then the aggregator computes the inverse DFT to reconstruct an estimate $\tilde{\mathbf{Q}}$ from $\tilde{\mathbf{F}}^k$. The final estimate $\tilde{\mathbf{Q}}$ has error characteristics of the FPA_k algorithm, but has been computed in a distributed way using DLPA.

We discuss FPA_k and DLPA in detail in next two sections.

4 Fourier Perturbation Algorithm

We nows describe in detail the FPA_k algorithm for improving accuracy of query answers for long sequences. In this section, we assume a central trusted server: how to distribute the algorithm using DLPA was briefly mentioned in the solution sketch and will be discussed in detail in Sec. 5. The FPA_k algorithm is based on the Discrete Fourier Transform, which we review briefly below.

4.1 The Discrete Fourier Transform

The DFT of a n-dimensional sequence $\mathbf{X}$ is a linear transform giving another n-dimensional sequence, $\mathbf{DFT}(X)$, with j^{th} element given as: $\mathrm{DFT}(\mathbf{X})_j = \sum_{i=1}^{n} e^{\frac{2\pi\sqrt{-1}}{n}ji}\mathbf{X}_i$. Similarly one can compute the Inverse DFT as $\mathrm{IDFT}(\mathbf{X})_j = \frac{1}{n}\sum_{i=1}^{n} e^{\frac{2\pi\sqrt{-1}}{n}ji}\mathbf{X}_i$. Furthermore, $\mathbf{IDFT}(\mathbf{DFT}(\mathbf{X})) = \mathbf{X}$.

Denote $\mathbf{DFT}^k(\mathbf{X})$ as the first k elements of $\mathbf{DFT}(\mathbf{X})$. The elements of $\mathbf{DFT}^k(\mathbf{X})$ are called the Fourier coefficients of the k lowest frequencies and they compactly represent the high-level trends in $\mathbf{X}$. An approximation $\mathbf{X}'$ to $\mathbf{X}$ can be obtained from $\mathbf{DFT}^k(\mathbf{X})$ as follows: Denoting $\mathbf{PAD}^n(\mathbf{DFT}^k(\mathbf{X}))$ the sequence

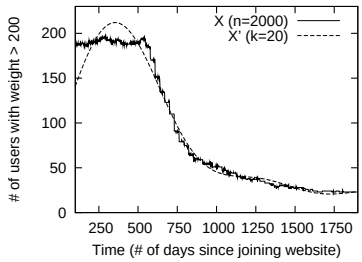


Figure 3: Reconstructed sequence $\mathbf{X}'$ vs. original $\mathbf{X}$

Algorithm 4.1 FPA$_k$(**Inputs:** sequence $\mathbf{Q}$, parameter λ)

1: Compute $\mathbf{F^k} = \mathbf{DFT}^k(\mathbf{Q}(I))$.
2: Compute $\tilde{\mathbf{F}}^k = \text{LPA}(\mathbf{F^k}, \lambda)$
3: Return $\tilde{\mathbf{Q}} = \mathbf{IDFT}(\mathbf{PAD}^n(\tilde{\mathbf{F}}^k))$

of length n obtained by appending $n-k$ zeros to $\mathbf{DFT}^k(\mathbf{X})$, compute $\mathbf{X}' = \mathbf{IDFT}(\mathbf{PAD}^n(\mathbf{DFT}^k(\mathbf{X})))$. Obviously $\mathbf{X}'$ may be different form $\mathbf{X}$ as ignoring the last $n-k$ Fourier coefficients may introduce some error. We denote $\text{RE}_i^k(\mathbf{X})$, short for reconstruction error at the i^{th} position, to be the value $|\text{X}'_i - \text{X}_i|$.

Example 4.1. *To give a sense of the reconstruction error, we consider a sequence* $\mathbf{X}$ *of length* $n = 2000$ *representing the number of people with weight* > 200 *in a real dataset (more details in Section 7), counted once every day over 2000 days. Fig. 3 shows the reconstructed sequence,* $\mathbf{X}'$*, using* $k = 20$ *DFT coefficients along with the original sequence* $\mathbf{X}$*.* $\mathbf{X}$ *shows the temporal trend in the # of overweight people in the dataset. As shown,* $\mathbf{X}'$ *captures the trend accurately showing that the reconstruction error is small even when compressing from* $n = 2000$ *to* $k = 20$ *DFT coefficients.*

4.2 The Algorithm

FPA$_k$ is shown in Algorithm 4.1. It begins by computing a sequence $\mathbf{F^k}$, comprising the first k Fourier coefficients in the DFT of $\mathbf{Q}(I)$. Then it perturbs $\mathbf{F^k}$ using the LPA algorithm with parameter λ to compute a noisy estimate $\tilde{\mathbf{F}}^k$. This perturbation is done to guarantee differential privacy. Finally, the algorithm computes the inverse DFT of $\mathbf{PAD}^n(\tilde{\mathbf{F}}^k)$ to get $\tilde{\mathbf{Q}}$, an approximation to the original query answers $\mathbf{Q}(I)$.

As with LPA, the parameter λ in FPA$_k$ needs to be adjusted in order to get ϵ-differential privacy. Since FPA$_k$ perturbs the sequence $\mathbf{F^k}$, λ has to be calibrated according to the L_1 sensitivity, $\Delta_1(\mathbf{F^k})$, of $\mathbf{F^k}$. Next we discuss the value of λ that makes FPA$_k(\mathbf{Q}, \lambda)$ differentially private.

Theorem 4.1. *Denote* $\mathbf{F^k} = \mathbf{DFT}^k(\mathbf{Q}(I))$ *the first* k *DFT coefficients of* $\mathbf{Q}(I)$*. Then, (i) the* L_1 *sensitivity,* $\Delta_1(\mathbf{F^k})$*, is at most* $\sqrt{k}$ *times the* L_2 *sensitivity,* $\Delta_2(\mathbf{Q})$*, of* $\mathbf{Q}$*, and (ii) FPA*$_k(\mathbf{Q}, \lambda)$ *is* ϵ*-differentially private for* $\lambda = \sqrt{k}\Delta_2(\mathbf{Q})/\epsilon$*.*

Proof (i) holds since $\Delta_2(\mathbf{F^k}) \leq \Delta_2(\mathbf{Q})$ (as the n Fourier coefficients have the same L_2 norm as $\mathbf{Q}$, while $\mathbf{F^k}$ ignores the last $n-k$ Fourier coefficients), and $\Delta_1(\mathbf{F^k}) \leq \sqrt{k}\Delta_2(\mathbf{F^k})$ (due to a standard inequality between the L_1 and L_2 norms of a sequence). (ii) follows since for $\lambda = \sqrt{k}\Delta_2(\mathbf{Q})/\epsilon \geq \Delta_1(\mathbf{F^k})/\epsilon$, $\tilde{\mathbf{F}}^k = LPA(\mathbf{F^k}, \lambda)$ computed in Step 2 is ϵ-differentially private, and $\tilde{\mathbf{Q}}$ in step 3 is obtained using $\tilde{\mathbf{F}}^k$ only.

4.3 Analyzing accuracy

Example 2.2 gives an instance of a n-length query sequence for which LPA results in an error of n/ϵ to each query answer, making the answers useless for a large n. Intuitively speaking, FPA$_k$ applies LPA on a length-k sequence. Hence the noise added during perturbation should be smaller. On the other hand,

ignoring $n-k$ DFT coefficients in FPA_k results in an additional (but often much smaller) reconstruction error. Below we formalize this argument to compare the errors for the two algorithms in greater detail.

In general, LPA results in a large noise whenever the $\mathbf{L}_1$ sensitivity of the query sequence is high. We define below irreducible query sequences that have the worst-possible L_1 sensitivity behavior.

Definition 4.1 (Irreducible queries)**.** *A query sequence $\mathbf{Q} = Q_1, \ldots, Q_n$ is irreducible if its L_1 sensitivity, $\Delta_1(\mathbf{Q})$, is equal to the sum, $\sum_{i=1}^{n} \Delta(Q_i)$, of the sensitivities of its constituent queries Q_i's.*

For all sequences $\mathbf{Q}$, $\Delta_1(\mathbf{Q}) \leq \sum_{i=1}^{n} \Delta(\text{Q}_i)$. Hence irreducible queries have the worst-possible L_1 sensitivities among all query sequences. Recurring queries over time-series data are often irreducible: one instance is the recurring query $\mathbf{Q}$ of Example 2.2. The improvement of FPA_k over LPA is the most on irreducible query sequences (since LPA has the least accuracy for such queries). We compare the accuracies over irreducible queries below (note however that irreducibility is not required for FPA_k).

For simplicity, we assume W.L.O.G that the sensitivity $\Delta(\text{Q}_i) = 1$ for all $\text{Q}_i \in \mathbf{Q}$. If not, we can rescale the queries by defining the sequence $\mathbf{Q}' = \{\text{Q}'_1, \ldots, \text{Q}'_n\}$ given as $\text{Q}'_i = \text{Q}_i/\Delta(\text{Q}_i)$. Then $\Delta(\text{Q}'_i) = \Delta(\text{Q}_i/\Delta(\text{Q}_i)) = \Delta(\text{Q}_i)/\Delta(\text{Q}_i) = 1$. Furthermore, $\mathbf{Q}(I)$ can be computed from $\mathbf{Q}'(I)$ by just multiplying with $\Delta(\text{Q}_i)$ at the i^{th} position. We call such sequences as normalized query sequences.

With this simplification, $\Delta_1(\mathbf{Q}) = n$ (as irreducibility means $\Delta_1(\mathbf{Q}) = \sum_{i=1}^{n} \Delta(\text{Q}_i) = \sum_{i=1}^{n} 1 = n$). Applying Theorem 2.2, we know that $error_i(\text{LPA}) = n/\epsilon$ in each query Q_i. On the other hand the following theorem shows the error of the FPA_k algorithm. Recall that $\text{RE}_i^k(\mathbf{Q}(I))$ is the reconstruction error for the i^{th} query, Q_i, caused by ignoring $n-k$ DFT coefficients of $\mathbf{Q}(I)$ and then computing the inverse DFT.

Theorem 4.2. *Fix $\lambda = \sqrt{k}\Delta_2(\mathbf{Q})/\epsilon$ so that $FPA_k(\mathbf{Q}, \lambda)$ is ϵ-differentially private. Then for all $i \in \{1, \ldots, n\}$, the $error_i(FPA_k)$ is $k/\epsilon + RE_i^k(\mathbf{Q}(I))$.*

Due to lack of space, the proof of the above theorem is deferred to the Appendix (Sec. A). The theorem shows that the error by FPA_k for each query is $k/\epsilon + \text{RE}_i^k(\mathbf{Q}(I))$, while we know that LPA yields an error of n/ϵ. Since the reconstruction error, $\text{RE}_i^k(\mathbf{Q}(I))$, is often small even for $k << n$, we expect the error in FPA_k to be much smaller than in LPA. This hypothesis is confirmed in our experiments that show that FPA_k gives orders of magnitude improvement over LPA in terms of error.

Choosing the Right k. So far we have assumed that k is known to us. Since $error_i(\text{FPA}_k)$ is $k/\epsilon + \text{RE}_i^k(\mathbf{Q}(I))$, a good value of k is important in obtaining a good trade-off between the perturbation error, k/ϵ, and the reconstruction error, $\text{RE}_i^k(\mathbf{Q}(I))$. If k is too big, the perturbation error becomes too big (giving the performance of LPA), while if k is too small the reconstruction error becomes too high.

We can often choose k based on prior assumptions about $\mathbf{Q}(I)$. For instance, if $\mathbf{Q}(I)$ is such that the Fourier coefficients corresponding to $\mathbf{Q}(I)$ decrease exponentially fast, then only a constant number (say k=10) of Fourier coefficients need to be retained during perturbation. Our experiments show that this naive method is applicable in many practical scenarios as Fourier coefficients of many real-word sequences decrease very rapidly [1].

However, for optimal performance, we need to adjust the value of k depending on the exact nature of $\mathbf{Q}(I)$. Computing k after looking at $\text{Q}(I)$, however, compromises differential privacy. An algorithm to efficiently compute k in a differentially private way is described in Sec. 6.1.1.

5 Distributed LPA (DLPA)

In both LPA and FPA_k, we assumed a trusted central server that stores the entire database I, computes the true answers $\text{Q}(I)$, and adds noise for privacy. Next we discuss how to adapt these algorithms for the distributed setting, i.e. database $I = I_1 \cup I_2 \ldots \cup I_U$ where I_u is the data of user u that she keeps to herself. We restrict ourselves to aggregate sum queries:[6] i.e. Q is such that $\text{Q}(I) = \sum_{u=1}^{U} f_u(I_u)$ where f_u is an arbitrary function that maps user u's database I_u to numbers.

[6]Note that FPA_k itself is more general and can support arbitrary queries when used in a centralized setting.

We first construct the distributed LPA algorithm (DLPA for short) and then use it to construct distributed FPA_k. As discussed in Sec. 2, our algorithms guarantee privacy even against a malicious aggregator and malicious users as long as a majority of honest users exist. For the utility of the aggregate estimate to be good, we assume that among the malicious users, there are no breakers and at most l liars.

We explain DLPA over a single aggregate-sum query Q: the generalization to query sequences is straightforward and just requires multiple invocation of the algorithm, once for each query in the sequence. In this section, we also make a simplifying assumption that there are no communication failures: all links between the users and aggregator are maintained throughout the time required to answer the query. We relax this assumption and discuss fault tolerance in Sec. 6.2.

5.1 Basics: Encryption Scheme

DLPA is built upon several cryptographic primitives. Before describing DLPA, we first discuss the encryption technique used in it: the threshold Paillier cryptosystem [10]. The cryptosystem is set up by choosing an integer m such that (i) $m = pq$ where p and q are strong primes (i.e $p = 2p' + 1$ and $q = 2q' + 1$), and (ii) $gcd(m, \phi(m)) = 1$. Once m is chosen, any number in $\mathbb{Z}_m$ (the set $\{0, 1, \ldots, m-1\}$) can be encrypted. Also denote $\mathbb{Z}_m^*$ the subset of numbers in $\mathbb{Z}_m$ that have a multiplicative inverse modulo m (eg. 0 does not have an inverse, but 1 has).

Key generation Choose a random element $\beta \in \mathbb{Z}_m^*$ and set $\lambda = \beta \times lcm(p, q)$. λ is the private key. Also set $g = (1 + m)^a b^m \ mod \ m^2$ for some randomly chosen $(a, b) \in \mathbb{Z}_m^* \times \mathbb{Z}_m^*$. The triplet (m, g, g^λ) forms the public key.

Encryption The encryption function Enc maps a plaintext message $t \in \mathbb{Z}_m$ to ciphertext $c \in \mathbb{Z}_{m^2}^*$. $Enc(t)$ is computed as $g^t r^m \ mod \ m^2$ where $r \in \mathbb{Z}_m^*$ is a randomly chosen number.

Decryption Denote L the function $L(u) = (u - 1)/m$ for any $u = 1 \ mod \ m$. The decryption of ciphertext $c \in \mathbb{Z}_{m^2}^*$ is the function $Dec(c) = \frac{L(c^\lambda \ mod \ m^2)}{L(g^\lambda \ mod \ m^2)}$

The encryption scheme has the following properties.

Homomorphic addition If c_i is a ciphertext for message t_i for $i \in 1, 2$, then $c_1 \cdot c_2$ is a ciphertext for message $t_1 + t_2$.

Distributed decryption Suppose the private key λ is shared by U users as $\lambda = \sum_u \lambda_u$ where λ_u is the private key for user u. Then decryption of a ciphertext c can be done distributedly (i.e. without any party knowing λ) as:

- Each user u computes his decryption share $c_u = c^{\lambda_u}$.
- The decryption shares are combined as $c' = \prod_{u=1}^{U} c_u$.
- Finally the decryption $t = \frac{L(c' \ mod \ m^2)}{L(g^\lambda \ mod \ m^2)}$ is computed.

5.2 Protocol for Computing Exact Sum

Let $\text{Q} = \sum_{u=1}^{U} f_u(I_u)$ be the aggregate-sum query that we wish to answer. For simplicity of presentation, we assume f_u returns numbers in the set $\{1, 2, \ldots, l\}$: our techniques work for any finite $D \subseteq \mathbb{R}$. Denote $x_u = f_u(I_u)$. Then $\text{Q} = \sum_{u=1}^{U} x_u$. We first start with a protocol for computing Q at the aggregator exactly. We will subsequently enhance the protocol to include noise addition. As discussed in the solution sketch (Sec. 3), computing even exact sum is difficult against a malicious aggregator: instead of the aggregator computing the encrypted sum and sending it to the users for decryption, it can send false decryption requests (say the encrypted private value of a single user) and break privacy.

Algorithm 5.1 Encrypt-Sum(x_u, r_u)

1: User u generates a random $r_u \in \mathbb{Z}_m$ computes $c_u = Enc(x_u + r_u)$, and sends c_u to the aggregator.
2: The aggregator computes $c = \prod_{u=1}^{U} c_u$.

We use the threshold Paillier scheme for the protocol. In a key generation phase, the private key λ is generated and distributed among the users as $\lambda = \sum_{i=u}^{U} \lambda_u$. Thus the users all together can perform distributed decryption using their keys λ_u. Note that since the key generation needs to be done only once (irrespective of the # of queries to be answered), expensive secret-sharing protocols [25] can be used for this purpose.

The protocol executes in two phases. In the first phase, the aggregator computes the required Q in encrypted form. Then in the second phase, a distributed decryption protocol is run to recover Q from the encrypted form.

The first phase is shown in Algorithm 5.1. We call it Encrypt-Sum(x_u, r_u): each user u encrypts his private value, x_u, added to a randomly generated r_u. Note that r_u is known only to user u. The aggregator obtains all the encryptions and multiples them to compute c. Due to the homomorphic properties of the encryption, the obtained c is an encryption of $\sum_{u=1}^{U}(x_u + r_u) = \mathrm{Q} + \sum_{u=1}^{U} r_u$. Since r_u's are not known to the aggregator, decrypting c would not reveal any information about Q. However, the following modification of the distributed decryption protocol can be used to obtain Q exactly. Note that g^{λ} is publicly known.

Algorithm 5.2 Decrypt-Sum(c, r_u)

1: The aggregator sends c to each user u for decryption.
2: User u computes decryption share $c'_u = c^{\lambda_u} g^{-r_u \lambda}$.
3: The aggregator collects c'_u from each user, combines them to get $c' = \prod_{i=u}^{U} c'_u$, and computes the final decryption $\overline{\mathrm{Q}} = \frac{L(c' \bmod m^2)}{L(g^{\lambda} \bmod m^2)}$

The above protocol is a minor modification of distributed decryption: user u multiplies an additional factor of $g^{-r_u \lambda}$ while generating his decryption share (step 2). We call this protocol Decrypt-sum(c, r_u). The following proposition shows the correctness of the protocol.

Proposition 5.1. *Let c = Encrypt-sum(x_u, r_u) and $\overline{Q}$ be the decryption computed by Decrypt-sum(c, r_u). Then $\overline{Q} = Q = \sum_{u=1}^{U} x_u$.*

Proof Sketch: The proof appears in the Appendix (Sec. B). Here we give a sketch. As mentioned earlier, c obtained from Encrypt-sum(x_u, r_u) is an encryption of $\mathrm{Q} + \sum_{u=1}^{U} r_u$. Each user u corrects for his r_u in Step 2 of Decrypt-sum(c, r_u). Thus the final $\overline{\mathrm{Q}}$ obtained is equal to Q.

Finally, we show that even though the Encrypt-sum and Decrypt-sum protocols can be used to compute the sum, $\sum_{u=1}^{U} x_u$, no other linear combinations can be computed. We saw that to compute the sum, the aggregator computed $c = \prod_u c_u$, where c_u is the encryption received by the aggregator from user u in the Encrypt-sum protocol (Step 2). Next we show that virtually no other encryption computed from these c_u's can be decrypted in order to breach privacy.

Theorem 5.1. *Suppose that the aggregator runs Encrypt-sum(x_u, r_u) protocol followed by Decrypt-sum(c', r_u) protocol for some c' of his choice. Let $c' = \prod_{u=1}^{U} c_u^{a_u}$ (where c_u are the encryptions sent by user u during Encrypt-sum protocol) such that $a_u - 1$ has an inverse mod m^2 (which implies $a_u \neq 1$) for some u. If the Decrypt-sum protocol decrypts c' correctly to give $\sum_{u=1}^{U} a_u x_u$, then there exists an attacker that breaks the security of the original distributed Paillier cryptosystem.*

Proof Sketch: The formal proof of security is quite involved and appears in the Appendix (Sec. B): here we just highlight the intuition. Let c = Encrypt-sum(x_u, r_u). Suppose the aggregator runs Decrypt-sum(c', r_u)

for $c' \neq c$, i.e. he sends wrong request for decryption, say the encryption of a single user's data (i.e., $a_u = 0$ for all users but one). In Step 2 of Decrypt-sum(c', r_u) protocol, other users will wrongly be correcting for their random r_u's that are not present in c', making the decrypted value $\overline{\mathrm{Q}}$ completely random and useless.

5.3 Protocol for Computing Noisy Sum

Now we describe how to add noise in the distributed setting. As mentioned earlier, LPA requires us to compute $\tilde{\mathrm{Q}} = \mathrm{Q} + Lap(\lambda)$ where $Lap(\lambda)$ is a Laplace random variable with mean 0 and variance λ^2. Denote $N(\mu, \sigma)$ to be a Gaussian random variable with mean μ and variance σ^2. We shall generate Laplace noise using 4 Gaussian variables by exploiting the following property (proved in the Appendix (Sec. B)).

Proposition 5.2. *Let $Y_i \sim N(0, \lambda)$ for $i \in \{1, 2, 3, 4\}$ be four Gaussian random variables. Then $Z = Y_1^2 + Y_2^2 - Y_3^2 - Y_4^2$ is a $Lap(2\lambda^2)$ random variable.*

The advantage of this decomposition is that Gaussian variables can be generated in a distributed fashion: To generate a $N(0, \lambda)$ variable, each user can generate a $N(0, \lambda/h)$ variable ($h = U/2$ is a lower bound on the number of honest users, i.e. a honest majority exists) and then the sum of these h, $N(0, \lambda/h)$, random variables gives the right $N(0, \lambda)$ variable. However, to compute a $Lap(\lambda)$ variable by Theorem 5.2, we need to compute squares of Gaussian random variables: for this we extend the Encrypt-Sum protocol described in the previous section to compute the encryption of $(\sum_{u=1}^{U} y_u)^2$ where y_u is the private $N(0, \lambda/h)$ of each user.

The protocol requires two randomly generated private keys $a_u, b_u \in \mathbb{Z}_m$ for each user u. The keys b_u are such that their sum for all users, $\sum_{u=1}^{U} b_u$, is 0. Denote a the sum $\sum_{u=1}^{U} a_u$. $Enc(a^2)$ is computed and made public in a key generation phase. The keys a_u, b_u need to be generated only once and expensive secret sharing protocols [25] can be used for this purpose. The protocol is shown below.

Algorithm 5.3 Encrypt-Sum-Squared(y_u, r_u) Protocol

1: User u computes $c_u = Enc(y_u + a_u + b_u)$ and sends it to the aggregator.
2: The aggregator computes $c = \prod_{u=1}^{U} c_u$ and sends it to each user u.
3: Each user u generates a random $r_u \in \mathbb{Z}_m$, computes $c_u = c^{y_u - a_u + b_u} Enc(r_u)$.
4: The aggregator collects c_u from each user and computes $c' = (\prod_{u=1}^{U} c_u) Enc(a^2)$

We call the above protocol Encrypt-Sum-Squared(y_u, r_u). Due to the homomorphic properties of the encryption, the final c obtained in Encrypt-Sum-Squared(y_u, r_u) can be shown to be an encryption of $(\sum_u y_u)^2 + \sum_u r_u$.

Finally we can discuss the the noisy-sum protocol to add $Lap(\lambda)$ noise. The protocol is shown in Algorithm 5.4 and is called Encrypt-Noisy-Sum(x_u, r_u): each user generates 4 Gaussian, $N(0, \sqrt{2\lambda}/U)$, random variables in Step 2. Each of these 4 Gaussian variable is used to generate an encryption of a $N(0, \sqrt{\lambda/2})$ Gaussian random variable in Step 3 using the Encrypt-Sum-Squared protocol. Then the user u generates an encryption for his private value x_u using the Encrypt-Sum protocol in Step 4. The random variables r_u^i are generated so as to force the right encryption to be computed by the aggregator in Step 5: the aggregator can only compute $c = \frac{c^1 c^2 c^5}{c^3 c^4}$; all others would have some r_u^i unbalanced. Due to the homomorphic properties of the encryption, the obtained c is an encryption of $\sum_u x_u + (\sum_u y_u^1)^2 + (\sum_u y_u^2)^2 - (\sum_u y_u^3)^2 - (\sum_u y_u^4)^2 + \sum_u r_u$. Also this c can be decrypted using decrypt-sum(c, r_u) protocol (Algorithm 5.2).

Next we state theorems showing the privacy and utility of the protocol. Due to space constraints, the proofs are deferred to the Appendix (Sec. B).

Theorem 5.2 (Privacy). *Let c =Encrypt-Noisy-Sum(x_u,r_u) and $\tilde{Q}$ =decrypt-sum(c, r_u). If there are at least $U/2$ honest users, then $\tilde{Q} = Q + Lap(\lambda)$+Extra-Noise, where $Lap(\lambda)$ is the noise generated by honest users and the Extra-Noise is that generated by malicious users. Thus for $\lambda = \Delta(Q)/\epsilon$, ϵ-differential privacy is guaranteed independent of what the malicious users and aggregator choose to do.*

Algorithm 5.4 Encrypt-Noisy-Sum(x_u, r_u)

1: User u chooses five random numbers $r_u^1, r_u^2, \ldots, r_u^5$ from $\mathbb{Z}_m$ and computes $r_u = r_u^1 + r_u^2 - r_u^3 - r_u^4 + r_u^5$.
2: User u generates four $N(0, \sqrt{2\lambda}/U)$ random variables $y_u^1, \ldots, y_u^4$.
3: Let c^j =Encrypt-Sum-Squared(y_u^j, r_u^j) for $j \in \{1, 2, 3, 4\}$.
4: Let c^5 =Encrypt-Sum(x_u, r_u^5)
5: Aggregator computes $c = \frac{c^1 c^2 c^5}{c^3 c^4}$.

Theorem 5.3 (Utility). *Let c =Encrypt-Noisy-Sum(x_u,r_u) and $\tilde{Q}$ =decrypt-sum(c, r_u). If there are no malicious users, then $\tilde{Q} = Q + Lap(2\lambda)$. Finally, in presence of l malicious users that are all liars and no breakers, $\tilde{Q}$ can deviate from $Q + Lap(2\lambda)$ by at most $l \times \Delta(Q)$.*

Distributed FPA$_k$. Above we discussed how to compute the perturbed estimate $\tilde{\mathrm{Q}}$ for a single query Q. As mentioned earlier, extending distributed LPA for a n-length query sequence $\mathbf{Q}$ is straightforward: apply the Encrypt-Noisy-Sum protocol n times, once for each $\mathrm{Q}_i \in \mathbf{Q}$. This works since LPA consists of n independent perturbations for each of the n queries in the sequence.

Implementing FPA$_k$ over the distributed setting is slightly more involved. Each user u first computes the answers of $\mathbf{Q}$ over his data I_u. We denote the answers as $\mathbf{Q}(I_u) = \mathrm{Q}_1(I_u), \ldots, \mathrm{Q}_n(I_u)$. Next the user u computes the first k DFT coefficients of the answers $\mathbf{Q}(I_u)$. Let us denote these k DFT coefficients by the sequence $\mathbf{F^k}(I_u)$. Recall that $\mathbf{F^k}(I)$ are the k DFT coefficients of $\mathbf{Q}(I)$, where $\mathbf{Q}(I)$ are the answers of $\mathbf{Q}$ over the complete database I. By linearity of the DFT transform, we know that $\mathbf{F^k}(I) = \sum_{u=1}^{U} \mathbf{F^k}(I_u)$: thus $\mathbf{F^k}$ is another aggregate-sum query. Then distributed LPA can be used by the aggregator to compute the perturbed estimate $\tilde{\mathbf{F}}^k$ for $\mathbf{F^k}(I)$. Finally, the aggregator takes the inverse DFT transform of $\tilde{\mathbf{F}}^k$ to compute $\tilde{\mathbf{Q}}$, a perturbed estimate of $\mathbf{Q}$.

6 Extensions

We now describe two useful extensions of our algorithm. The first extension enables us to choose a good value of the parameter k (# of DFT coefficients used in FPA$_k$) if a central trusted server exists, while the second extension allows us to tolerate failures of users *during* the execution of the DLPA algorithm.

6.1 Choosing a Good Value of k

So far we have assumed that the value k is known to us before executing FPA$_k$. This was unavoidable in the distributed setting as each user needs to know k before computing his k DFT coefficients $\mathbf{F^k}(I_u)$. However, assuming a central server holds all the data I, then the server can compute k depending on the exact nature of $\mathbf{Q}(I)$. Computing k after looking at $\mathbf{Q}(I)$, however, compromises differential privacy. We present here an algorithm to efficiently compute k in a differentially private way. This algorithm can be used instead of LPA to accurately answer long query sequences in the centralized setting.

6.1.1 Differentially-private Sampling

Denote $\mathbf{F} = \mathbf{DFT}(\mathbf{Q}(I))$, and $\mathbf{F}^k$ (resp. $\mathbf{F}^{n-k}$) the first k (resp. last $n-k$) Fourier coefficients in $\mathbf{F}$. Next we discuss a sampling procedure that computes k and the perturbed differentially private estimate, $\tilde{\mathbf{F}}^k$, for the first k Fourier coefficients.

Theorem 6.1. *Denote $U(k, \tilde{\mathbf{F}}^k)$ the function $|\mathbf{F}^{n-k}|_2 + |\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2$. Then sampling k and $\tilde{\mathbf{F}}^k$ with probability proportional to $e^{-U(k,\tilde{\mathbf{F}}^k)/\lambda}$ satisfies ϵ-differential privacy for $\lambda = \sqrt{2}\Delta_2(\mathbf{Q})/\epsilon$.*

Theorem 6.1 (proved in the Appendix C.1) shows a differentially private way of sampling k and $\tilde{\mathbf{F}}^k$. However, the sampling may not be efficient. This is because, even assuming that k has been sampled

Algorithm 6.1 SPA(**Inputs:** $\mathbf{Q}(I)$, parameter λ)

1: Compute $\mathbf{F} = \mathbf{DFT}(\mathbf{Q}(I))$
2: Denote $U'(k)$ the function $|\mathbf{F}^{n-k}|_2 + k\sqrt{n}/\epsilon$.
3: Sample a $k \in \{1, \ldots, n\}$ with probability $\propto e^{-U'(k)/\lambda}$.
4: Let g be a $G(1/\lambda^2, (k+1)/2)$ random variable
5: Compute $\tilde{\mathbf{F}}^k = \mathbf{F^k} + \mathbf{N}^k(0, \sqrt{g})$
6: Compute $\tilde{\mathbf{Q}} = \mathbf{IDFT}(\mathbf{PAD}^n(\tilde{\mathbf{F}}^k))$.

already, sampling $\tilde{\mathbf{F}}^k$ needs to be done with probability proportional to $e^{|\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2/\lambda}$. In other words, $\tilde{\mathbf{F}}^k$ has to be sampled with probability based on its L_2 distance with another sequence $\mathbf{F^k}$. This is difficult as now elements of the sequence $\tilde{\mathbf{F}}^k$ cannot be sampled independently: if Pr is the sampling distribution, then $Pr(\tilde{\mathrm{F}}_i^k = x | \tilde{\mathrm{F}}_j^k = y) \neq Pr(\tilde{\mathrm{F}}_i^k = x)$. On the other hand, in FPA_k, $\tilde{\mathbf{F}}^k$ was generated independently by adding $\mathbf{Lap}^k(\lambda)$ random variables to $\mathbf{F^k}$. Nevertheless we discuss next an efficient way to sample from $U(k, \tilde{\mathbf{F}}^k)$.

6.1.2 Sampling Perturbation Algorithm (SPA)

Before giving the algorithm, we recall two kinds of random variables: (i) $N(\mu, \Sigma)$, that represents a normal random variable with mean μ and variance σ^2 (additionally, denote $\mathbf{N}^k(\mu, \sigma)$ a vector of k i.i.d normal variables), and (ii) $G(\theta, r)$, that represents a Gamma random variable with PDF given as $\frac{1}{\Gamma(r)\theta^r} x^{r-1} e^{-r/\theta}$, where $\theta > 0$, and $r > 0$ are parameters, and $\Gamma(.)$ is the gamma function. This PDF is similar to the exponential distribution except for an extra factor of x^{r-1}.

Our sampling-based perturbation algorithm (SPA) is shown in Algorithm 6.1. In the first step it computes the entire DFT of $\mathbf{Q}$. Then in steps 2 and 3, it samples a value of k. Intuitively speaking, $U'(k) = |\mathbf{F}^{n-k}|_2 + k\sqrt{n}/\epsilon$ computed in the step 2 is the sum of the reconstruction error, $|\mathbf{F}^{n-k}|_2$ (this is the loss incurred by ignoring all elements in $\mathbf{F}^{n-k}$), and the perturbation error, $k\sqrt{n}/\epsilon$ (an additional factor of $\sqrt{n}$ appears as this is the perturbation error for the whole sequence). In step 3, those values of k are more likely to be picked that give a lower $U'(k)$, i.e. give a better tradeoff between the reconstruction error and the perturbation error.

Once k has been sampled, the algorithm continues to sample $\tilde{\mathbf{F}}^k$. This is done by first picking a gamma random variable g in Step 4, and then perturbing $\mathbf{F^k}$ by adding $\mathbf{N}^k(0, \sqrt{g})$ noise vector in Step 5. Even though, $\mathbf{N}^k$ represents a vector of k independent normal variables, the $\tilde{\mathbf{F}}^k$ vector has not been generated in an independent fashion: g generated in Step 4 makes all elements of $\tilde{\mathbf{F}}^k$ correlated (for instance if g is picked to be 0, then all $\tilde{\mathbf{F}}^k = \mathbf{F^k}$). This makes sure that $\tilde{\mathbf{F}}^k$ has been generated in the right way, confirmed by the following theorem proved in the Appendix C.1.

Theorem 6.2 (Privacy). *$SPA(\mathbf{Q}, \lambda)$ is ϵ-differentially private for $\lambda = \sqrt{2}\Delta_2(\mathbf{Q})/\epsilon$.*

Finally, we show that it always makes sense to run our SPA algorithm as opposed to LPA. If $\mathbf{Q}$ is compressible, SPA will sample a good $k << n$ decreasing the error significantly (as confirmed in our experiments). However, if $\mathbf{Q}$ is not compressible, the following theorem (proved in the Appendix C.1) shows that no matter what the total error of SPA would be at most a factor $\sqrt{2}\log n$ times worse than LPA for any normalized irreducible query sequence[7] $\mathbf{Q}$.

Theorem 6.3 (Utility). *Let $\mathbf{Q}$ be any normalized irreducible query sequence. Fix $\lambda_1 = \Delta_1(\mathbf{Q})/\epsilon$ and $\lambda_2 = \sqrt{2}\Delta_2(\mathbf{Q})/\epsilon$ such that $LPA(\mathbf{Q}, \lambda_1)$ and $FPA_k(\mathbf{Q}, \lambda_2)$ are ϵ-differentially private. Then $error(SPA) \leq \sqrt{2} \cdot (\log n) \cdot error(LPA)$.*

[7] Recall that normalized query sequences have individual query sensitivities rescaled to 1 while irreducible sequences have L_1 sensitivity equal to the sum of individual query sensitivities.

6.2 Fault-tolerance

Our distributed protocol has two phases. In the Encrypt-Noisy-Sum(x_u, r_u) phase, the aggregator computes an encryption c for the noisy sum of the private values, x_u, and the random values, r_u, of all users. Then in the Decrypt-Sum phase, all users need to correct for their respective random values r_u. In addition, since the secret key λ is shared as the sum $\sum_{u=1}^{U} \lambda_u$ of the private keys λ_u's, all users need to send their decryption shares in order to decrypt. This makes the protocol susceptible to failures: if a single user does not respond in the decrypt-sum phase, no decryption can be obtained.

The solution is to (i) use a (T, U)-threshold decryption [10] scheme in which any T valid decryption shares out of U possible ones can be used to decrypt, and (ii) instead of choosing a completely random $r_u \in \mathbb{Z}_m$ during the encrypt-sum protocol, a user chooses $r_u \sim Lap(\Delta(\mathrm{Q})/\epsilon)$, i.e. r_u is chosen from the Laplace distribution sufficient for guaranteeing differential privacy. This r_u is sufficient: to minimize noise the aggregator has an incentive for adding all users' data when computing the aggregate encryption to be sent back for decryption, and leaving out a single user's data results in a Laplace noise sufficient for differential privacy.

Having seen the extensions still ensure privacy, let us see how they help in case of failures. Firstly, if f users fail to send their decryption shares, the (T, U)-threshold property ensures that a decryption can still be computed as long as $f < U - T$. Furthermore, not using the decryption share of f users means that the random value r_u of each of the f users is left uncorrected for, at the end of decryption. This results in an extra noise of f $Lap(\Delta(\mathrm{Q})/\epsilon)$ variables. It can be shown that the expected sum of of these f variables increases as $\sqrt{f}\Delta(\mathrm{Q})/\epsilon$. In other words, the noise increases as the square root of the number of user failures f as demonstrated in our experiments. As long as f is small, this behavior is acceptable. For a large f, the distributed protocol for computation of Q has to be repeated.

Note that in the above solution, we are concerned about failures of users that happen after the first phase and before the second phase of our protocol. Failures that happen before or after the complete execution of the protocol for a query do not affect the accuracy of our protocol. Since execution of our protocol for a single query takes less than a few seconds in practice, failures within this small time window is rare and are of small size. In Section 7.4, we experimentally show that the impact of such failures is small.

7 Experiments

We have implemented our algorithms in Java, using the BigInteger library for cryptographic components of the algorithms. This section evaluates our prototype using a 2.8 GHz Intel Pentium PC with 1GB RAM.

Data sets. We use three real data sets in our evaluation.

- **GPS:** This is the GPS trace from Microsoft's Multiperson Local Survey (MLS) project [18]. The GPS trace was collected from 253 voluntary drivers, mostly in the Seattle, Washington area, covering about 135,000 kilometers of driving. The entire trace has approximately 2.3 million timestamped latitude/longitude points comprising about 16,000 discrete trips. The median interval between recorded points on a trip is 6 seconds and 62 meters. Thus, successive points on a trip are highly correlated.
- **Weight:** This is trace of body weights, collected from an online weight-monitoring website[8]. The trace contains daily weight data of about 300 users for a period of up to 5 years.
- **Traffic:** This data, collected from Department of Transportation of San Antonio, Texas[9], reports volume and speed data at about 30 intersections in the city. We use a 6-months long trace, where data is reported once every 15 minutes.

Queries. For evaluating utility, we consider the following 6 query sequences. The first two queries are on the **GPS** data set, the next two queries are on the **Weight** data set, and the last two queries are on the **Traffic** dataset.

[8] `http://www.hackersdata.com`
[9] `http://www.transguide.dot.state.tx.us/`

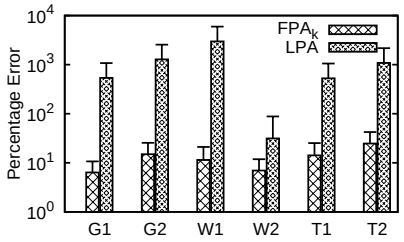


Figure 4: Average errors of FPA$_k$ and LPA

- Query $G1$: A query sequence whose value at each timestamp is an histogram over locations counting the number of people at each location. The locations are obtained by dividing the map into a 50×50 grid. The query sequence has 2000 different timestamps spread uniformly over 2 weeks.
- Query $G2$: The number of people at a fixed location evaluated at 2000 different timestamps spread uniformly over 2 weeks.
- Query $W1$: Number of people with weight > 200 lb on each day for a total of 2000 days.
- Query $W2$: A query sequence whose value at each timestamp is the correlation coefficient between two month-long time-series: (i) people with weight > 200 on day i of the month, and (ii) people with weight decreasing on day i of the month.
- Query $T1$ and Query $T2$: Occupancy at two different city intersections with an interval of 15 minutes for a total of 6 months.

As the examples suggest, our algorithm can support a wide variety of queries. Unless otherwise stated, we use $k = 30$ for the FPA$_k$ algorithm and 1-differential privacy ($\epsilon = 1$) as the privacy requirement in our experiments.

7.1 Accuracy of Answers

We first evaluate the accuracy of outputs of our FPA$_k$ algorithm and compare it with the LPA algorithm. We report error percentage of a sequence of output estimate, which is the total L_2 error in the estimate (See Def. 2.3) normalized by the maximum possible L_2 value of a query answer.

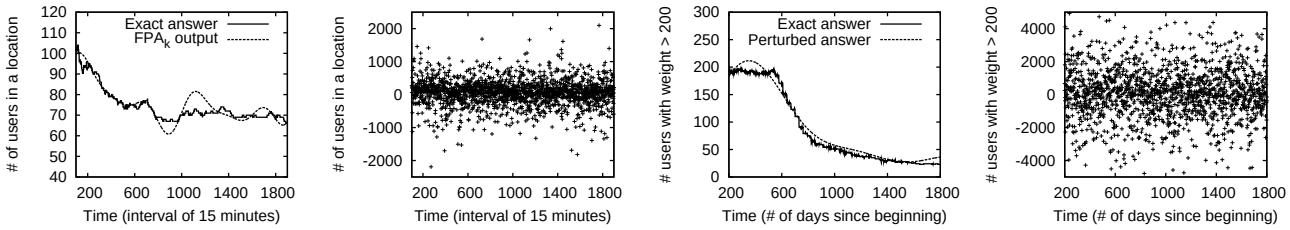
Figure 4 reports the average error percentage of FPA$_k$ and LPA algorithms for the 6 query sequences. Average error percentage is computed as the average over 100 runs of each algorithm, the variance in the error percentages are represented by the error bars. As shown, FPA$_k$ has orders of magnitude better error percentage than LPA (graph shows y-axis using a log-scale). In fact, LPA has an error percentage $\gg 100\%$ showing that estimates obtained are completely meaningless.[10] On the other hand, FPA$_k$ has error percentages of $< 20\%$ that are mostly acceptable in practice.

Figure 5 demonstrates the errors of both algorithms more visually. Fig 5(a) and (c) plot outputs of the FPA$_k$ algorithm along with the exact answers of $W1$ and $G2$. Fig 5(b) and (d) plot the estimates obtained by LPA for the same two queries for the same privacy parameter. The graphs show that FPA$_k$ estimates follow the curve of the exact answer very closely, while LPA results in estimates that are practically meaningless.

7.2 Effect of DFT Parameters

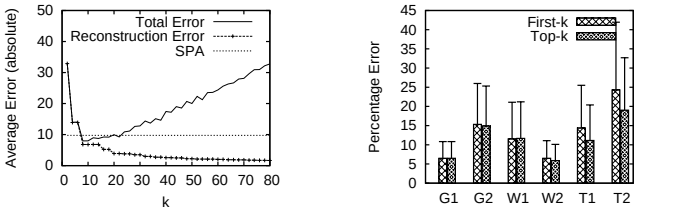
Value of k. To understand the impact of k on the accuracy of FPA$_k$, we vary the value of k in evaluating the query $W1$. Fig. 6(a) demonstrates the results. It shows that as k increases the total error decreases at first, then reaches a minimum, and finally increases. This is because the total error of FPA$_k$ is a combination of

[10] Laplace noise can send query answers outside their range making the error $> 100\%$. Truncation at range boundaries can make error $= 100\%$, but is not done here to reflect the true magnitude of the noise.



(a) $G2$ output with FPA_k (b) $G2$ output with LPA (c) $W1$ output with FPA_k (d) $W1$ output with LPA

Figure 5: Point-wise errors of FPA_k and LPA over time



(a) FPA_k with different k (b) First vs. Top k coefficients

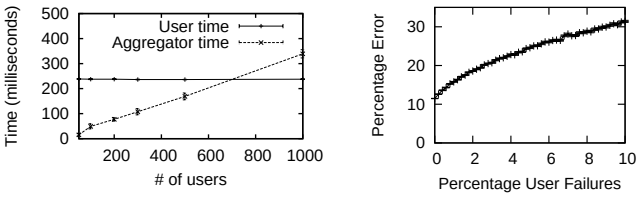
Figure 6: Effect of DFT parameters

reconstruction error and perturbation error. Reconstruction error decreases with increasing values of k (also shown in Figure 6(a)), while perturbation error increases with increasing k. The total error is minimized for an optimal value of k that finds a sweet spot between these two types of errors. This highlights the importance of choosing a good value of k for FPA_k.

In the distributed case, there is no way to choose the best k by looking at query answers. Thus k has to be predetermined, which is the reason we used a fixed value of $k = 30$ in our previous experiments (and it worked reasonably well for all queries). However the graph shows that for the $W1$ query, $k = 10$ would have given a better trade-off, and the results of FPA_k would have been better if we used $k = 10$. In a centralized setting, our sampling perturbation algorithm (SPA) can be used to privately sample the value of k. The error for that algorithm is also shown in the figure by the horizontal line and it is quite near the optimal error for $k = 10$.

First-k vs. Top-k DFT coefficients. Our algorithms, both FPA_k and SPA, choose the first k DFT coefficients, for a given value of k. The choice of leading coefficients is unavoidable in a distributed setting since all the users need to pick the same k DFT coefficients for our algorithms to work. Even in the presence of the centralized server, choosing the best set of k DFT coefficients in a differentially private way is inefficient. However, if possible, it is always better to choose the k largest DFT coefficients (i.e., top-k coefficients), since they give lower reconstruction error than the first-k coefficients. This leads to the natural question: how much do we sacrifice in accuracy for using the first-k DFT coefficients (which gives us differential privacy) instead of using the top-k coefficients (which does not give differential privacy)?

Figure 6(b) answers the above question. It shows the errors of FPA_k for different queries and compares it with the errors of a hypothetical algorithm that uses the top-k DFT coefficients (but may not necessarily be differentially private). Both the algorithms use $k = 30$ coefficients. The graph shows (with a linear scale for y-axis) that we do not loose much even if we just pick the first k coefficients (indicating that they generally are the largest coefficients). A substantial difference occurs only in $T1$ and $T2$ queries since they have periodic behavior at slightly higher frequencies indicating there largest coefficients are not the first k. Even for these queries, the difference is less than 5%.



(a) Computation cost (b) Errors for user failures

Figure 7: Evaluation of Distributed LPA

7.3 Computation & Communication Overhead

Fig. 7(a) shows the computational overhead of our algorithm for computing a single aggregate-sum query. The graph shows the computation times (averaged over 100 runs) at a user and at the aggregator, as a function of the number of users involved in the query. The average computation time at a user is independent of the number of users and remains nearly constant. On the other hand, the time required by the aggregator increases linearly with the number of users. In all cases, the computation overhead is quite small, most of which is spent for cryptographic operations. We have also measured the communication overhead of our prototype and found that the algorithm has a small overhead: 0.5 Kb for each user (considering both incoming and outgoing messages) and 0.5 Kb times the # of users for the aggregator.

7.4 Effect of Failures

Fig. 7(b) shows the fault-tolerance behavior of our algorithm based on the extensions described in Sec. 6.2. The fault-tolerant algorithm is implemented and used to answer the $W1$ query of length $n = 2000$ using $k = 30$ DFT coefficients. We again report the percentage error: the total L_2 error in the computed estimate normalized by the maximum L_2 value of $\mathbf{Q}$. The percentage user failure is the percentage of users who fail during the protocol.

The graph shows that the error increases with the square root of the number of users failing and is quite reasonable with even say 5% failure rate. Note that the accuracy of our algorithm is affected only by the failures that happen *during* execution of the algorithm to answer a query, which is typically less than a few seconds in a distributed system. Failure within this short time should be rare and small in size in practice. In a very rare occasion, if a large number of users fail during this small time window, the algorithm should be started from the beginning.

8 Related Work

Relational Data. Many approaches for relational data have been proposed that support formal definitions of privacy s.a. differential privacy [7, 15, 28], $\rho_1\rho_2$ breach [9], adversarial privacy [23, 24], etc. Among these, most relevant to our work are those that focus on query answering [7,15,28]. [7] proposes the LPA algorithm that adds independent Laplace noise to each query answer. The noise is calibrated according to the L_1 sensitivity. Recurring query sequences over time-series data have high L_1 sensitivity ($O(n)$ for a n length sequence) and thus LPA does not accurately answer such queries.

To improve accuracy of query answers under differential privacy, [28] focuses on range-count queries, i.e. count queries where the predicate on each attribute is a range. The main idea is to take a sequence of range-count queries that are disjoint and then perturb the entire Discrete Wavelet Transform of the sequence. Such disjoint range-count queries have a L_1 sensitivity of $O(1)$ and can be accurately answered using their technique. However the technique can not be used to answer recurring sequences over time-series data accurately, owing to their high L_1 sensitivity.

Another approach for answering query sequences accurately is proposed in [15], where constraints over the answers of multiple queries are exploited to improve accuracy. Usually no constraints exist over recurring query sequences and thus the method is largely inapplicable for time-series data.

Distributed protocols. Many Secure Multiparty Computation (SMC) techniques have been proposed to evaluate functions securely over data from multiple users [12]. In general, such techniques are slow [29] and infeasible for practical scenarios. However, if the function to be evaluated is a sum function, then efficient techniques using threshold homomorphic functions have been proposed [4, 13]. Such techniques often require a communication mechanism with a broadcast facility, or a mechanism enabling a user to verify the computation done by other users. This is not possible in our setting where the aggregator does all the computations and the users do not have enough resources to check whether those computations are correct. In addition, we need to compute noisy sum (in order to guarantee differential privacy) which is more difficult than computing just the sum of the inputs. To the best of our knowledge, [6] is the only known technique that computes noisy sum. However, it uses expensive secret-sharing protocols leading to a computation load of $O(U)$ per user, where U is the number of users. This makes the technique infeasible for large U.

Techniques for Time-series data. Most work on time-series data assume the centralized setting: a trusted server publishes an anonymized version of the data of all users and aggregate queries can then be run on the published data. [11] publishes data by adding a virtual user whose data is sampled from a public noise distribution. [22] works on a single time-series data by obtaining an orthonormal transform (s.a. DFT), then adding noise only to large coefficients of the transform, and finally obtaining the inverse transform. The main difference between our technique and theirs is the lack of formal privacy guarantee. Techniques [11, 22] show privacy by protecting against specific attacks (s.a. linear least-square regression or linear filtering in [22]), however no formal privacy guarantee (s.a differential privacy) is provided. Furthermore no distributed solution is discussed. Same holds for most works on location data privacy (See [19] and the references therein).

9 Conclusion

We have proposed novel algorithms to privately answer queries on distributed time-series data. Our first algorithm FPA_k can answer long query sequences over correlated time series data in a differentially private way. FPA_k perturbs k DFT coefficients of an answer sequence, thereby improving the accuracy for an n-length query sequence from $\Theta(n)$ of existing algorithms to roughly $\Theta(k)$, if the k DFT coefficients can accurately reconstruct all the query answers. For achieving differential privacy in distributed setting, we propose DLPA algorithm that implements Laplace noise addition in a distributed way with $O(1)$ complexity per user. Our experiments with three real data sets show that our solution improves accuracy of query answers by orders of magnitude and also scales well with a large number of users.

Acknowledgements. We would like to thank Josh Benaloh, Melissa Chase, Seny Kamara, Frank McSherry, and Mariana Raykova for many helpful discussions. Eric Horvitz, John Krumm, and Paul Newson kindly gave us access to the **GPS** dataset.

References

[1] Agrawal, R., Faloutsos, C., and Swami, A. N. Efficient similarity search in sequence databases. In *FODO* (1993).

[2] Blaesild, P. The two-dimensional hyperbolic distribution and related distributions, with an application to johannsen's bean data. In *Biometrika, Vol. 68* (1981).

[3] Blum, A., Dwork, C., McSherry, F., and Nissim, K. Practical privacy: The suLQ framework. In *PODS* (2005).

[4] Cramer, R., Damgard, I., and Nielsen, J. B. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT* (2001).

[5] Dwork, C. Differential privacy: A survey of results. In *TAMC* (2008).

[6] Dwork, C., Kenthapadi, K., McSherry, F., Mironov, I., and Naor, M. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT* (2006).

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

[8] Eisenman, S. B., Miluzzo, E., Lane, N. D., Peterson, R. A., Ahn, G.-S., and Campbell, A. T. The bikenet mobile sensing system for cyclist experience mapping. In *ACM SenSys* (2007).

[9] Evfimievski, A., Gehrke, J., and Srikant, R. Limiting privacy breaches in privacy preserving data mining. In *PODS* (2003).

[10] Fouque, P., Poupard, G., and Stern, J. Sharing decryption in the context of voting or lotteries. In *FC: International Conference on Financial Cryptography* (2000), LNCS, Springer-Verlag.

[11] Ganti, R. K., Pham, N., Tsai, Y.-E., and Abdelzaher, T. F. PoolView: stream privacy for grassroots participatory sensing. In *ACM SenSys* (2008).

[12] Goldreich, O. Secure multi-party computation.

[13] Gritzalis, D. Secure electronic voting.

[14] Guha, S., Reznichenko, A., Haddadi, H., and Francis, P. Serving ads from localhost for performance, privacy, and profit. In *HotNets* (2009).

[15] Hay, M., Rastogi, V., Miklau, G., and Suciu, D. Boosting the accuracy of differentially-private histograms through consistency. In *VLDB 2010*.

[16] Hull, B., Bychkovsky, V., Zhang, Y., Chen, K., Goraczko, M., Miu, A., Shih, E., Balakrishnan, H., and Madden, S. Cartel: a distributed mobile sensor computing system. In *ACM SenSys* (2006).

[17] Jonhson, N., Kotz, S., and Balakrishnan, N. Continuous univariate distributions (second ed., vol. 1, chapter 18).

[18] Krumm, J. A survey of computational location privacy. *Personal and Ubiquitous Computing, 2008* (2008).

[19] Krumm, J., and Horvitz, E. Predestination: Where do you want to go today? *IEEE Computer* (2007).

[20] Machanavajjhala, A., Gehrke, J., Kifer, D., and Venkitasubramaniam, M. l-diversity: Privacy beyond k-anonymity. In *ICDE* (2006).

[21] McSherry, F., and Talwar, K. Mechanism design via differential privacy. In *FOCS* (2007).

[22] Papadimitriou, S., Li, F., Kollios, G., and Yu, P. S. Time series compressibility and privacy. In *VLDB* (2007).

[23] Rastogi, V., Hay, M., Miklau, G., and Suciu, D. Relationship privacy: Output perturbation for queries with joins. In *PODS 2009*.

[24] Rastogi, V., Suciu, D., and Hong, S. The boundary between privacy and utility in data publishing. In *VLDB* (2007).

[25] Shamir, A. How to share a secret. *Communications of the ACM 22*, 11 (1979).

[26] Shilton, Katie, B. J., Estrin, D., Govindan, R., , and Kang, J. Designing the personal data stream: Enabling participatory privacy in mobile personal sensing. In *37th Research Conference on Communication, Information and Internet Policy (TPRC)* (2009).

[27] Sweene, L. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* (2002).

[28] Xiao, X., Wang, G., and Gehrke, J. Differential privacy via wavelet transforms. In *ICDE 2010*.

[29] Yao, A. C.-C. Protocols for secure computations (extended abstract). In *FOCS* (1982).

A Fourier Perturbation Algorithm (FPA_k)

In this section, we prove Theorem 4.2. We first restate it below.

Theorem (4.2). *Fix $\lambda = \sqrt{k}\Delta_2(\mathbf{Q})/\epsilon$ so that $FPA_k(\mathbf{Q}, \lambda)$ is ϵ-differentially private. Then for all $i \in \{1, \ldots, n\}$, the $error_i(FPA_k)$ is $k/\epsilon + RE_i^k(\mathbf{Q}(I))$.*

Proof: Let $\tilde{\mathbf{Q}} = \text{FPA}_k(\mathbf{Q}, \lambda)$ be the perturbed query responses $\{\tilde{\text{Q}}_1, \ldots, \tilde{\text{Q}}_n\}$ returned by the FPA_k algorithm. The variance and expected error in each $\tilde{\text{Q}}_i$ is calculated below. Recall that $\tilde{\text{Q}}_i$ is obtained by using the IDFT of the perturbed k DFT coefficients $\tilde{\mathbf{F}}^k$.

$$Var(\tilde{\text{Q}}_i) = \sum_{j=1}^{k} \frac{Var(\tilde{\text{F}}_j^k)}{n^2} = \frac{k\lambda^2}{n^2} = \frac{k(k\Delta_2^2(\mathbf{Q})/\epsilon^2)}{n^2} = \frac{k^2n^2/\epsilon^2}{n^2} = k^2/\epsilon^2.$$

The above holds because W.L.O.G, we assume $\Delta(\text{Q}_i) = 1$ for all $i \in \{1, \ldots, n\}$, and thus $\Delta_2^2(\mathbf{Q}) = n$.

Now, denoting $\mu_i = \mathbb{E}\tilde{\text{Q}}_i$ and $\text{RE}_i^k(\mathbf{Q}(I))$ the error obtained due to ignoring $n - k$ Fourier coefficients, we have:

$$\begin{aligned} error_i(\text{FPA}_k) &= \mathbb{E}|\tilde{\text{Q}}_i - \text{Q}_i| \leq \mathbb{E}|\mu_i - \text{Q}_i| + \mathbb{E}|\tilde{\text{Q}}_i - \mu_i| = \text{RE}_i^k(\mathbf{Q}(I)) + \mathbb{E}|\tilde{\text{Q}}_i - \mu_i| \\ &\leq \text{RE}_i^k(\mathbf{Q}(I)) + \sqrt{\mathbb{E}|\tilde{\text{Q}}_i - \mu_i|_2^2} \quad \text{(By Jensens' inequality)} \\ &= \text{RE}_i^k(\mathbf{Q}(I)) + \sqrt{Var(\text{Q}_i)} = \text{RE}_i^k(\mathbf{Q}(I)) + k/\epsilon \end{aligned}$$

Hence proved.

B Distributed Laplace Perturbation Algorithm (DLPA)

B.1 Protocol for Exact Sum

In this section, we give the proofs of Proposition 5.1 and Theorem 5.1. We first restate Proposition 5.1 below.

Proposition (5.1). *Let c = Encrypt-sum(x_u, r_u) and $\overline{Q}$ be the decryption computed by Decrypt-sum(c, r_u). Then $\overline{Q} = Q = \sum_{u=1}^{U} x_u$.*

Proof: Denote $\text{R} = \sum_{u=1}^{U} r_u$. Let c' be the product of decryption shares obtained in Step 3 of Decrypt-sum(c, r_u). Then, we know that:

$$c' = \prod_{u=1}^{U} c'_u = \prod_{u=1}^{U} c^{\lambda_u} g^{-r_u\lambda} = c^{\sum_{u=1}^{U}\lambda_u} g^{-\lambda\sum_{u=1}^{U} r_u} = c^{\lambda} g^{-\lambda\text{R}} \tag{1}$$

Since, c =Encrypt-sum(x_u, r_u), we know that c is an encryption of $\sum_{u=1}^{U}(x_u + r_u) = \text{Q} + \text{R}$. Hence, $c^{\lambda} = g^{\lambda(\text{Q}+\text{R})}$ *mod* m^2. From Eq (1), we know that $c' = c^{\lambda} g^{-\lambda\text{R}}$, and hence doing multiplications in $\mathbb{Z}_{m^2}$ we have

$$c' = c^{\lambda} g^{-\lambda\text{R}} = g^{\lambda(\text{Q}+\text{R})} g^{-\lambda\text{R}} = g^{\lambda\text{Q}}$$

Thus $\frac{L(c' \bmod m^2)}{L(g^{\lambda} \bmod m^2)} = \frac{\lambda\text{Q}}{\lambda} = \text{Q}$. Hence proved.

Next, we restate Theorem 5.1 and prove it.

Theorem (5.1). *Suppose that the aggregator runs Encrypt-sum(x_u, r_u) protocol followed by Decrypt-sum(c', r_u) protocol for some c' of his choice. Let $c' = \prod_{u=1}^{U} c_u^{a_u}$ (where c_u are the encryptions sent by user u during Encrypt-sum protocol) such that $a_u - 1$ has an inverse mod m^2 (which implies $a_u \neq 1$) for some u. If the Decrypt-sum protocol decrypts c' correctly to give $\sum_{u=1}^{U} a_u x_u$, then there exists an attacker that breaks the security of the original distributed Paillier cryptosystem.*

Proof: Recall that $c_u = Enc(x_u + r_u)$ is the encryption sent by the user u in the Encrypt-sum(x_u,r_u) protocol. Suppose the aggregator is able to decrypt an encryption $\tilde{c} = \prod_{i=1}^{n} c_u^{a_u}$ such that $a_u - 1$ has an inverse mod m^2 some $u \in \{1, \ldots, U\}$. W.L.O.G we assume that u is the last user U, i.e. $a_U \neq 1$. If this is possible, we show how this can be used by an attacker to break the security of the original distributed Paillier cryptosystem. More formally, we prove that if the attacker A_1 exists then so does attacker A_2.

Attacker A_1**:** Let $c_u = Enc(x_u + r_u)$ and $\tilde{c} = \prod_{u=1}^{U} c_u^{a_u}$ where $a_U - 1$ has an inverse mod m^2. Given $\tilde{c}'_u = \tilde{c}^{\lambda_u} g^{-r_u \lambda}$ for $u \in \{1, \ldots, U\}$, A_1 can compute $\sum_{u=1}^{U} a_u x_u$.

Attacker A_2**:** Let $\hat{c_u}$ be an encryption of x_u. Given $\hat{c_u}$ for $u \in \{1, \ldots, U\}$ and private key shares λ_u for $u \in \{1, \ldots, U-1\}$, A_2 can compute $\sum_{u=1}^{U} a_u x_u$.

Note that A_2 only uses $U - 1$ decryption keys to decrypt. This should not be possible for a (U, U)-threshold system in which the private key λ is unknown and distributed as $\sum_{u=1}^{U} \lambda_u$. Now let us suppose attacker A_1 exists. We describe below how A_2 can use A_1 to launch his attack.

Algorithm B.1 A_2's simulation of A_1

1: A_2 receives an encryption $\hat{c_u} = Enc(x_u)$ for $u \in \{1, \ldots, U\}$ and $U - 1$ private key shares $\lambda_1, \ldots, \lambda_{U-1}$
2: A_2 generates U numbers $r_1, \ldots, r_U$ chosen uniformly at random from $\mathbb{Z}_m$.
3: A_2 computes $\tilde{c} = \prod_{u=1}^{U} (\hat{c_u} g^{r_u})^{a_u}$. Thus $\tilde{c}$ is an encryption of $\sum_{u=1}^{U} a_u (x_u + r_u)$.
4: A_2 generates $U - 1$ decryption shares as $\tilde{c}'_u = \tilde{c}^{\lambda_u} g^{-\lambda r_u}$ for $u = \{1, \ldots, U-1\}$.
5: A_2 chooses $r \in \mathbb{Z}_m$ uniformly at random, and generates the U^{th} decryption share as $\tilde{c}'_u = \tilde{c}^{-\sum_{u=1}^{U-1} \lambda_u} g^{-\lambda r}$.
6: A_2 sends these shares $\tilde{c}'_1, \ldots, \tilde{c}'_U$ to A_1 who computes $\sum_{u=1}^{U} a_u x_U$.

Proof of correct simulation: Now we show that $\tilde{c}'_1, \ldots, \tilde{c}'_U$ are correct inputs to A_1. It is easy to see that $\tilde{c}$ is an encryption of $\sum_{u=1}^{U} a_u (x_u + r_u)$. Furthermore, $\tilde{c}'_u$ for $u \in 1, \ldots, U-1$ have been generated correctly, i.e. exactly as generated in the Decrypt-sum protocol (Step 2). The only remaining share is $\tilde{c}'_U$, which has been generated using only $\tilde{c}^{\lambda_1}, \ldots, \tilde{c}^{\lambda_{U-1}}$ in the simulation above (as A_2 does not know λ_U) and thus may not be equal to $\tilde{c}^{\lambda_U} g^{-\lambda r_U}$ as required by the Decrypt-sum protocol. However we show that $\tilde{c}'_U$ generated in the above simulation is statistically indistinguishable from $\tilde{c}^{\lambda_U} g^{-\lambda r_U}$. Note that

$$\begin{aligned} \tilde{c}^{\lambda_U} g^{-\lambda r_U} &= \tilde{c}^{\lambda - \sum_{u=1}^{U-1} \lambda_u} g^{-\lambda r_U} = \tilde{c}^{-\sum_{u=1}^{U-1} \lambda_u} \tilde{c}^{\lambda} g^{-\lambda r_U} \\ &= \tilde{c}^{-\sum_{u=1}^{U-1} \lambda_u} g^{\lambda (\sum_{u=1}^{U} a_u (x_u + r_u))} g^{-\lambda r_U} = \tilde{c}^{-\sum_{u=1}^{U-1} \lambda_u} g^{\lambda t + \lambda (a_U - 1) r_U} \end{aligned}$$

Here $t = \sum_{i=1}^{U-1} a_u (x_u + r_u)$. Also denote $\mathbf{C} = \tilde{c}^{-\sum_{u=1}^{U-1} \lambda_u}$. So from the above equations, we have $\tilde{c}^{\lambda_U} g^{-\lambda r_U} = \mathbf{C} g^{\lambda t + \lambda (a_U - 1) r_U}$.

Now we needed to show the indistinguishability of $\tilde{c}^{\lambda_U} g^{-\lambda r_U}$ and $\tilde{c}'_U$. Thus we need to show indistinguishability of $\mathbf{C} g^{\lambda t + \lambda (a_U - 1) r_U}$ and $\tilde{c}'_U$. As $\tilde{c}'_U = \tilde{c}^{-\sum_{u=1}^{U-1} \lambda_u} g^{-\lambda r} = \mathbf{C} g^{-\lambda r}$ in the above simulation, we need to show indistinguishability of $\mathbf{C} g^{(\lambda t + \lambda (a_U - 1) r_U}$ and $\mathbf{C} g^{-\lambda r}$.

We show this in two steps. First we show the following lemma:

Lemma B.1. *For any r_U and r'_U in $\mathbb{Z}_m$, let $\tilde{c}$ and $\tilde{c}'$ be encryptions of $t + a_U (x_U + r_U)$ and $t + a_U (x_U + r'_U)$, respectively. Denote $\mathbf{C} = \tilde{c}^{-\sum_{u=1}^{U-1} \lambda_u}$ and $\mathbf{C}' = \tilde{c}'^{-\sum_{u=1}^{U-1} \lambda_u}$. Then $\mathbf{C} g^{\lambda t + \lambda (a_U - 1) r_U}$ is indistinguishable from $\mathbf{C}' g^{\lambda t + \lambda (a_U - 1) r_U}$.*

Proof of Lemma: Suppose there exists a polynomial-time attacker B_1 that can distinguish between $\mathbf{C} g^{\lambda t + \lambda (a_U - 1) r_U}$ and $\mathbf{C}' g^{\lambda t + \lambda (a_U - 1) r_U}$. Then we construct a polynomial-time attacker B_2 that can break

the original Paillier cryptosystem, i.e. distinguish between the encryptions of two messages m and m'. To construct B_2 using B_1, we will simply use $m = t + a_U(x_U + r_U)$ and $m' = t + a_U(x_U + r'_U)$. Then $\tilde{c}$ and $\tilde{c}'$ are the encryptions of m and m' respectively. Then B_2 will just compute $\mathbf{C} = \tilde{c}^{-\sum_{u=1}^{U-1} \lambda_u}$ and $\mathbf{C}' = \tilde{c}'^{-\sum_{u=1}^{U-1} \lambda_u}$. Finally B_2 will use B_1 to distinguish between $\mathbf{C}g^{\lambda t + \lambda(a_U - 1)r_U}$ and $\mathbf{C}'g^{\lambda t + \lambda(a_U - 1)r_U}$. If B_1 distinguishes them correctly, then B_2 correctly distinguishes between m and m'. Hence proved.

From Lemma B.1, we know that for arbitrary r_U and r'_U, $\mathbf{C}g^{\lambda t + \lambda(a_U - 1)r_U}$ and $\mathbf{C}'g^{\lambda t + \lambda(a_U - 1)r_U}$ are indistinguishable (recall that $\mathbf{C}$ is a power of an encryption of $t + a_U(x_U + r_U)$, while $\mathbf{C}'$ is that of $t + a_U(x_U + r'_U)$). In particular, if r_U and r'_U are both chosen uniformly at random, then $\mathbf{C}g^{\lambda t + \lambda(a_U - 1)r_U}$ and $\mathbf{C}'g^{\lambda t + \lambda(a_U - 1)r_U}$ are indistinguishable. On switching r_U and r'_U (as they are both chosen uniformly at random) in $\mathbf{C}'g^{\lambda t + \lambda(a_U - 1)r_U}$, we get that $\mathbf{C}'g^{\lambda t + \lambda(a_U - 1)r_U}$ is same as $\mathbf{C}g^{\lambda t + \lambda(a_U - 1)r'_U}$. Thus $\mathbf{C}g^{\lambda t + \lambda(a_U - 1)r_U}$ and $\mathbf{C}g^{\lambda t + \lambda(a_U - 1)r'_U}$ are indistinguishable.

Now denote $r = -(t + (a_U - 1)r'_U)$ *mod* m^2. If r'_U is chosen uniformly at random from $\mathbb{Z}_m$ and $a_U - 1$ has an inverse mod m^2 (i.e., $a_U \neq 1$ and a_U is not 1 *mod* p or 1 *mod* q), then r also has uniform distribution in $\mathbb{Z}_m$. Hence $\mathbf{C}g^{\lambda t + \lambda(a_U - 1)r'_U}$ and $\mathbf{C}g^{-\lambda r}$ are indistinguishable. Finally, this gives that $\mathbf{C}g^{\lambda t + \lambda(a_U - 1)r_U}$ and $\mathbf{C}g^{-\lambda r}$ are indistinguishable, yielding that $\tilde{c}_u$ and $\tilde{c}'_u$ are indistinguishable.

B.2 Protocol for Computing Noisy Sum

We give here the proofs of Proposition 5.2 and Theorems 5.2, and 5.3.

Proposition (5.2)**.** *Let $Y_i \sim N(0, \lambda)$ for $i \in \{1, 2, 3, 4\}$ be four Gaussian random variables. Then $Z = Y_1^2 + Y_2^2 - Y_3^2 - Y_4^2$ is a Lap($\lambda^2/2$) random variable.*

Proof: Denote $Z_1 = Y_1^2 + Y_2^2$ and $Z_2 = Y_3^2 + Y_4^2$. Then Z_1 and Z_2 are chi-square distributions with $k = 2$ degree of freedom, which is the same as a Exponential distribution with parameter $1/(2\lambda^2)$ (See [17]). Furthermore, if Z_1 and Z_2 follow exponential distributions then $Z = Z_1 - Z_2$ follows the $Lap(2\lambda^2)$ distribution. Hence Proved.

Theorem (5.2)**.** *Let c =Encrypt-Noisy-Sum(x_u,r_u) and $\tilde{Q}$ =decrypt-sum(c, r_u). If there are at least $U/2$ honest users, then $\tilde{Q} = Q$+Lap(λ)+Extra-Noise, where Lap(λ) is the noise generated by honest users and the Extra-Noise is that generated by malicious users. Thus for $\lambda = \Delta(Q)/\epsilon$, ϵ-differential privacy is guaranteed independent of what the malicious users and aggregator choose to do.*

Proof: Due to the homomorphic properties, the encryption c obtained from Encrypt-Noisy-Sum(x_u,r_u) is the encryption of $\sum_u x_u + (\sum_u y_u^1)^2 + (\sum_u y_u^2)^2 - (\sum_u y_u^3)^2 - (\sum_u y_u^4)^2 + \sum_u r_u$. W.L.O.G. assume that $\{1, \ldots, U/2\}$ is the set of $U/2$ honest users. Since honest users follow the protocol correctly, y_u^i is a $N(0, \sqrt{2\lambda}/U)$ random variable for $i \in \{1, 2, 3, 4\}$ and $u \in \{1, \ldots, U/2\}$. Thus, $Y_i = \sum_{u=1}^{U/2} y_u^i$ are $N(0, \sqrt{\lambda/2})$ random variables for $i \in \{1, 2, 3, 4\}$. Also since c is an encryption of $\sum_u x_u + (\sum_u y_u^1)^2 + (\sum_u y_u^2)^2 - (\sum_u y_u^3)^2 - (\sum_u y_u^4)^2 + \sum_u r_u$, it is an encryption of $\sum_u x_u + Y_1^2 + Y_2^2 - Y_3^2 - Y_4^2 + \sum_u r_u$+Extra-Noise, where Extra-noise are terms that contain the y_u^i for possibly malicious users $u \in \{U/2 + 1, \ldots, U\}$. Thus c is an encryption of Q + Z + R+Extra-Noise, where Q = $\sum_u x_u$, $Z = Y_1^2 + Y_2^2 - Y_3^2 - Y_4^2$, and R = $\sum_u r_u$. Finally, since Y_i is a $N(0, \sqrt{\lambda/2})$ random variable, applying Proposition 5.2, we get Z is a $Lap(\lambda)$ variable. Hence $\tilde{\text{Q}}$ =decrypt-sum(c, r_u), the decryption of c, is Q + $Lap(\lambda)$+Extra-Noise.

Theorem (5.3)**.** *Let c =Encrypt-Noisy-Sum(x_u,r_u) and $\tilde{Q}$ =decrypt-sum(c, r_u). If there are no malicious users, then $\tilde{Q} = Q + Lap(2\lambda)$. Finally, in presence of l malicious users that are all liars and no breakers, $\tilde{Q}$ can deviate from $Q + Lap(2\lambda)$ by at most $l \times \Delta(Q)$.*

Proof: Denote $Y'_i = \sum_{u=1}^{U} y_u^i$. If all users are honest, then c is an encryption of Q+Z+R, where Q = $\sum_u x_u$, $Z = Y_1'^2 + Y_2'^2 - Y_3'^2 - Y_4'^2$, and R = $\sum_u r_u$. Finally, since Y'_i is a $N(0, \sqrt{\lambda})$ random variable, applying Proposition 5.2, we get Z is a $Lap(2\lambda)$ random variable. Hence $\tilde{\text{Q}}$ =decrypt-sum(c, r_u), the decryption of c, is Q + $Lap(2\lambda)$.

In presence of l liars, the value of Q can be distorted by at most $l \times \Delta(\mathrm{Q})$, since each user can lie and distort Q by at most its sensitivity. Also liars need to follow the protocol exactly, hence the noise will still be $Lap(2\lambda)$. Thus the decrypted value $\tilde{\mathrm{Q}}$ can deviate from $\mathrm{Q} + Lap(2\lambda)$ by at most $l \times \Delta(\mathrm{Q})$.

C Extensions

C.1 Sampling Perturbation Algorithm

We give here the proofs of Theorems 6.1, 6.2, and 6.3.

Theorem (6.1). *Denote $U(k, \tilde{\mathbf{F}}^k)$ the function $|\mathbf{F}^{n-k}|_2 + |\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2$. Then sampling k and $\tilde{\mathbf{F}}^k$ with probability proportional to $e^{-U(k,\tilde{\mathbf{F}}^k)/\lambda}$ satisfies ϵ-differential privacy for $\lambda = \sqrt{2}\Delta_2(\mathbf{Q})/\epsilon$.*

Proof: The proof of this theorem follows from the privacy of exponential mechanism [21]. The only requirement is to show that the sensitivity of $U(k, \tilde{\mathbf{F}}^k) = |\mathbf{F}^{n-k}|_2 + |\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2$ is $\sqrt{2}\Delta_2(\mathbf{Q})$. This is evident since $\Delta(|\mathbf{F}^{n-k}|_2 + |\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2) \leq \Delta(|\mathbf{F}^{n-k}|_2 + |\mathbf{F}^k|_2) \leq \Delta(\sqrt{2}|\mathbf{F}|_2) = \Delta(\sqrt{2}|\mathbf{Q}|_2) \leq \sqrt{2}\Delta_2(\mathbf{Q})$. Hence proved.

Theorem (6.2). *SPA$(\mathbf{Q}, \lambda)$ is ϵ-differentially private for $\lambda = \sqrt{2}\Delta_2(\mathbf{Q})/\epsilon$.*

Proof: Recall that $U(k, \tilde{\mathbf{F}}^k) = |\mathbf{F}^{n-k}|_2 + |\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2$. Denote $Pr(k_0, \tilde{\mathbf{F}}_0^k \leftarrow U(k, \tilde{\mathbf{F}}^k))$ the probability that $k = k_0$ and $\tilde{\mathbf{F}}^k = \tilde{\mathbf{F}}_0^k$ when sampling k and $\tilde{\mathbf{F}}^k$ according to the function $U(k, \tilde{\mathbf{F}}^k)$. Then $Pr(k_0, \tilde{\mathbf{F}}_0^k) = e^{-U(k_0,\tilde{\mathbf{F}}_0^k)/\lambda} d\tilde{\mathbf{F}}^k$, where, informally speaking, $d\tilde{\mathbf{F}}^k$ denotes a infinitesimally small change in $\tilde{\mathbf{F}}^k$. Denote ∞^k the k-length vector with each element ∞. Denote $Z_k = \int_{-\infty^k}^{\infty^k} e^{-|\tilde{\mathbf{F}}^k - \mathbf{F}^k|/\lambda} d\tilde{\mathbf{F}}^k$. Note that Z_k is independent of $\mathbf{F}^k$: the integral corresponding to Z_k is invariant under translations in $\tilde{\mathbf{F}}^k$. Denote $z_k = \log Z_k$.

Define $\bar{U}(k, \tilde{\mathbf{F}}^k) = U(k, \tilde{\mathbf{F}}^k) - z_k + k\sqrt{n}/\epsilon$. Then $\Delta(\bar{U}(k, \tilde{\mathbf{F}}^k)) = \Delta(U(k, \tilde{\mathbf{F}}^k))$, which we know from Theorem 6.1 to be $\sqrt{2}\Delta_2(\mathbf{Q})/\epsilon = \lambda$. Finally, we show below that SPA$(\mathbf{Q}, \lambda)$ samples k and $\tilde{\mathbf{F}}^k$ according to probability $e^{-\bar{U}(k,\tilde{\mathbf{F}}^k)/\lambda}$ and hence by exponential mechanism [21] satisfies ϵ-differential privacy.

Firstly, note that $Pr(k \leftarrow \bar{U}(k, \tilde{\mathbf{F}}^k))$ is proportional to $e^{-\bar{U}(k,\tilde{\mathbf{F}}^k)/\lambda}$. Plugging in the value of $\bar{U}(k, \tilde{\mathbf{F}}^k)$ from above, we get that the probability is proportional to as $e^{(-U(k,\tilde{\mathbf{F}}^k)+z_k-k\sqrt{n}/\epsilon)/\lambda}$. Simplifying this expression, we get $Pr(k \leftarrow \bar{U}(k, \tilde{\mathbf{F}}^k))$ is proportional to $e^{-|\mathbf{F}^{n-k}|_2/\lambda} \cdot (-Z_k) \cdot e^{z_k} \cdot e^{-k\sqrt{n}/(\epsilon\lambda)}$, which is equal to $e^{-U'(k)/\lambda}$, as in step 3 of SPA algorithm.

Finally, note that once k is fixed, the probability $Pr(\tilde{\mathbf{F}}^k \leftarrow \bar{U}(k, \tilde{\mathbf{F}}^k)|k)$ is simply proportional to $e^{|\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2/\lambda}$. Thus only thing we need to show is that the steps 4 and 5 of SPA compute an $\tilde{\mathbf{F}}^k$ sampled with probability proportional to $e^{|\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2/\lambda}$. This follows from the fact that the distribution proportional to $e^{|\tilde{\mathbf{F}}^k - \mathbf{F}^k|_2/\lambda}$ corresponds to the multidimensional hyperbolic distribution, and steps 4 and 5 give a way of sampling from the distribution, as shown in [2]. Hence proved.

Theorem (6.3). *Let $\mathbf{Q}$ be any normalized irreducible query sequence. Fix $\lambda_1 = \Delta_1(\mathbf{Q})/\epsilon$ and $\lambda_2 = \sqrt{2}\Delta_2(\mathbf{Q})/\epsilon$ such that LPA$(\mathbf{Q}, \lambda_1)$ and FPA$_k(\mathbf{Q}, \lambda_2)$ are ϵ-differentially private. Then error(SPA) $\leq \sqrt{2} \cdot (\log n) \cdot$ error(LPA).*

Proof: Since $\mathbf{Q}$ is a normalized query sequence, $\Delta_1(\mathbf{Q}) = n$. Then $\lambda = \Delta_1(\mathbf{Q})/\epsilon = n/\epsilon$ and $error(\mathrm{LPA}) = n^{3/2}/\epsilon$. Also if k is the value sampled by SPA, then $error(\mathrm{SPA})$ is less than $|\mathbf{F}^{n-k}| + \sqrt{2}k\sqrt{n}/\epsilon$. In particular, for $k = n$, we have $error(\mathrm{SPA}) = \sqrt{2}error(\mathrm{LPA})$. Since SPA samples k with probability proportional to $|\mathbf{F}^{n-k}| + \sqrt{2}k\sqrt{n}/\epsilon$, it can be shown that expectation over k the value of $|\mathbf{F}^{n-k}| + \sqrt{2}k\sqrt{n}/\epsilon$ can be no worse that $(\sqrt{2}n\sqrt{n}/\epsilon)\log n = \sqrt{2} \cdot (\log n) \cdot error(\mathrm{LPA})$. Hence proved.