Aggregate Measurement via Oblivious Shuffling

Erik Anderson[1], Melissa Chase[1], F. Betül Durak[1], Esha Ghosh[1], Kim Laine[1], and Chenkai Weng[2]

[1]Microsoft
[2]Northwestern University

November 9, 2021

Abstract

We introduce a secure histogram aggregates method which is suitable for many applications such as ad conversion measurements. Our solution relies on three-party computation with linear complexity and guarantees differentially private histogram outputs. We formally analyse the security and privacy of our method and compare it with existing proposals. Finally, we conclude our report with a performance analysis.

1 Introduction

In today's web, various entities perform cross-site tracking of user activity, *e.g.*, to inform sites about where users are potentially experiencing issues and to derive insights about a marketing campaign. This cross-site tracking is often accomplished via third-party cookies, where website A loads content which gets and sets a cookie tied to website F. When website F is also loaded across a variety of other websites, it is thus able to track an individual user's activities for a variety of purposes and with varying privacy implications.

Due to these concern, browser vendors have started to move to greatly restrict when third-party cookies can be set. For instance, Apple Safari blocks all third-party cookies by default. Similarly, Google Chrome's Privacy Sandbox effort proposes to phase out third-party cookies as they exist today. Nearly every other browser vendor is moving in the same direction. Analytics built on top of third-party cookies, however, are a vital part of the current ad-funded web ecosystem that helps provide many valuable sites and services free of charge to users.

In many measurement-related scenarios, the most valuable information is about what events happen most frequently, such as how often an ad placement on a publisher site results in the user completing a purchase of a specific item. These insights are currently derived by having an entity collect information about each individual user and then aggregate insights across users. There's no promise that the data about a specific user won't be used for other, more privacy-invasive purposes. By enabling aggregate measurements to be made while providing technical guarantees against individual tracking, many of today's measurement scenarios can be handled in a privacy-preserving way that avoids the concerns that apply to existing methods that utilize third-party cookies.

In the ad conversion setting, a party called the *Reporting Origin* collects reports from browsers. A report collected from the client is a (key, value) pair, where the key is formed with set of predefined attributes such as user identification, location, age, gender, device ID, or ad category. Each of these attributes is allocated a certain number of bits; in practice, the total length of key is expected to be 32 to 40 bits. The value in the collected reports represents some predefined values such as the dollar amount the user spent, or how much time they spent browsing the advertised website. The Reporting Origin is designed to run data analysis on collected reports, such as compute histograms of attributes, or sums of values corresponding to specific attributes. The current way to doing this is unnecessarily privacy-invasive, especially if the aim of the Reporting Origin is to only do aggregate data analysis. An ideal solution would protect individual users' privacy, while allowing data analysis across many users. In this work, we propose a privacy-preserving aggregate protocol based on secure multi-party computation and differential privacy, which allows legitimate institutes/individuals to analyze data collected from multiple browsers. A key requirement is that the information they learn cannot be used to trace back understanding to any specific user.

1.1 Related Work

There exists a big line of prior academic work to solve privacy-preserving aggregate systems. The most intuitive ones are based on general purpose multi-party computations [16, 20]; some are new primitives called Distributed Point Functions (DPF) [14, 7, 8, 6]; and the rest has different flavour [10, 5]. In this report, we mainly focus on DPF [6] and Prio [10], as these approaches are currently being proposed for standardization by IETF [4, 13, 1]. We detail these proposals in Section 5 and compare their complexities with our solution shortly.

There is another prior work for secure histogram computation, although it is not the main focus of the paper [17]. However, this work tightly focuses on access pattern hiding secure graph computation. More specifically, they work in a model where each client can submit an arbitrary number of inputs and their protocol can hide the number of submission by a specific client with differential privacy. For the histogram computations, the protocol uses 2-party shuffling and it only outputs the exact histogram with no differential privacy protection.

1.2 Overview

We focus on histogram aggregates on attributes of key and propose an efficient (linear time and communication complexity in the number of reports)

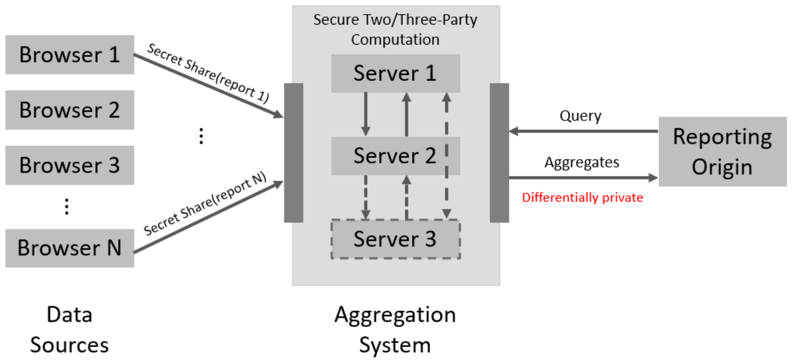

Figure 1: Our System Architecture.

privacy-preserving aggregation system, where the Reporting Origin learns only the histograms of keys and does not learn any other data associated to a specific client. It is based on honest-majority secure three-party computation, which is efficient. The system architecture of our protocol is given in Figure 1. The Reporting Origin can only query the aggregation system through an interface that returns differentially private histogram results. At a high level, the clients generate secret shares of reports for two servers; these two servers add dummy reports for each attribute key, where the number of dummy reports is sampled from a Laplace distribution. The original reports and dummy reports are permuted with the help of third server through an oblivious shuffling sub-routine. Finally, shares are revealed to form the results.

1.2.1 Bucketization

Even though our protocol can enable aggregates over *value*'s, such as sum or mean, we will specifically focus on histogram aggregates defined on key attributes. Therefore, we will omit the value from reports from now on and will add them back when we define other aggregate protocols in the next version of this work. Furthermore, we will consider the full report with a key encoding multiple attributes. For example, a 16-bit key in a single report could represent two attributes: a 10-bit Ads Category and a 6-bit Client Region. Our protocol allows **running histogram aggregates on *any* attribute reported in a single** key even after a key is fully reported (as 16 bits from our example) without needing to report each attribute separately. This enables us to compute histograms in a layered manner. For example, we can run the histogram on only Ads Category attribute first and continue building the histogram on Client Region, when Ads Category is equal to a specific value (see Figure 2).

Due to secret sharing of the keys within the reports, we can achieve a higher level of flexibility than the proposals in related work. We can also improve the performance of our protocol when the size of the attributes becomes too large to parse at once, as we will clarify later in Section 6.

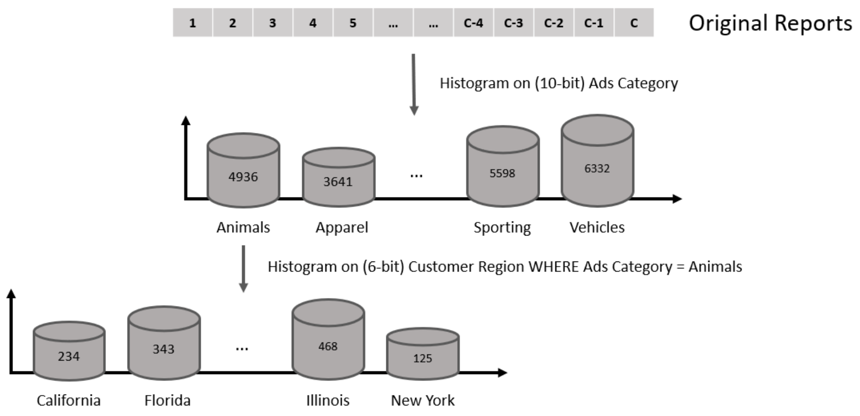

Figure 2: An example demonstrating the flexible functionality of our attribute aggregation protocol. The outputs will be differentially private histogram counts, *i.e.*, the size of each bucket plus a noise term.

We refer to our histogram aggregate protocol as Bucketization, as it can be seen as putting reports in buckets representing the attribute keys. Thus, we call each aggregate point (an attribute in key) a bucket.

1.2.2 Security guarantees

We formally analyse the security and privacy guarantees of our protocol in different threat models. First of all, our Bucketization protocol provides security against malicious clients. Informally, it is achieved with no added cost when the malicious browsers are limited to a small fraction of the total browsers.[1] Second, our protocol achieves $(\epsilon, \delta = \mathsf{p}\frac{\mathsf{e}^{\epsilon}}{1-\mathsf{p}})$ differential privacy (DP) for a very small p, meaning that the reported histograms reveal only noisy aggregate counts. Third, we prove privacy against semi-honest servers in the aggregate system.[2]

A stronger notion of security assumes a malicious server who may deviate from the protocol execution. The malicious server's goal is to learn sensitive information about the clients. Finally, we prove privacy against a malicious server in an honest majority setting: when there are three servers, only one server is allowed to be malicious. We can also extend our proof to provide privacy against a Reporting Origin that colludes with one of the servers. Our proofs are based on a leakage model.

As in [10, 6], robustness in the case of a malicious server is not guaranteed. We give the comparison between DPF, Prio, and our protocol for security and

[1]This seems like a reasonable assumption to make in the real world, where the number of malicious clients, *e.g.*, those infected with a *particular* malware, is generally much smaller than the number of trustworthy clients.

[2]Semi-honest adversaries execute the protocol honestly but try to learn additional information from the protocol transcript.

privacy guarantees in Table 1.

Protocol	Robustness against malicious clients	Semi-honest servers	Privacy against a malicious server	Correctness against a malicious server	DP
DPF [6]	requires sketching protocol	yes	yes	no	yes
Prio [10]	requires zero-knowledge range proofs	yes	yes	no	no
Bucketization	no cost	yes	yes	no	yes

Table 1: Threat model and security guarantee comparison.

1.2.3 Complexity

The time complexity of our aggregation protocol is $O(C + B\bar{M})$, where C is the number of client reports, B is the number of buckets, and $\bar{M}$ is the average noise added to each bucket ($\bar{M} = O\left(-(1/\epsilon)\ln\delta\right)$ to achieve (ϵ, δ)-DP). The communication complexity is $O(\ell(C+B\bar{M}))$, where ℓ is the reported key length. The communication required from clients in their reports is only $O(\ell C)$ with some encryption overhead. We give the comparison between DPF, Prio, and our protocol for time and communication complexity in Table 2.

Protocol	Time Complexity	Server-to-Server Comm.	Client-to-Server Comm.
DPF [6]	$(\ell C^2/t)$DPF.Eval calls	$\ell \log_2(p)$	$\lambda\ell$
Prio [10]	$O(BC)$	$O(C\log_2(p))$	$B\log_2(p)$
Bucketization	$O(C + B\bar{M})$	$O(\ell(C + B\bar{M}))$	ℓ

Table 2: ℓ is the reported key length; t is the pruning threshold ($t = 1$ for full histogram); $\log_2(p)$ is the size of the finite field; λ is the security parameter; C is the number of client reports; B is the number of buckets ($B = 2^\ell$ for full histogram); $\bar{M}$ is the noise added on average to each bucket in Bucketization. DPF row does not account the DP protection (our scheme without DP would mean $\bar{M} = 0$). Note that the time complexity of DPF.Eval is exponential in ℓ.

As shown, our protocol has linear time and communication complexity in the number of reports. When we consider the communication complexity of the aggregate system as the total communication required for an end-to-end execution, we must include the communication required from clients to servers as well. In that sense, the linear complexity in the number of clients is inevitable. In our work, we add a similar communication complexity for the server-to-server communication. As the network between servers is normally much better, we do not see this as a drawback. Furthermore, our approach comes with a time complexity advantage compared to other protocols. DPF takes a completely orthogonal approach and introduces a time complexity quadratic in the number of clients, while avoiding the linear communication complexity between servers. We give a detailed description of an existing proposal using DPF in Section 5. We compare the performance of our protocol with [6] (DPF) in Section 6.

Prio defines a complete toolbox for secure and private aggregation. However, it comes with the cost of having clients encode their reports to help aggregation by the servers and creating proofs that their reports are well-formed. We explain further how Prio approaches to solve private aggregate protocols in Section 5. We do not compare Prio performance with our protocol as there is no known codebase for histograms to use. However, there is an ongoing effort to build such [3].

2 Preliminaries

In this section, we will first define the notations we use throughout the paper and give two main building blocks of our final protocol: a differential privacy mechanism and an oblivious random shuffling protocol. Later, in Section 3, we will instantiate these primitives.

2.1 Notation

We refer to web browsers (in the real-world scenario motivating this work) as clients. We denote the Reporting Origin as $\mathcal{R}$ and the three helper servers as $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$. We denote by C the total number of clients (or, more precisely, the total number of reports) and by $\mathsf{b} \in \{1, 2, 3\}$ the server index. Unless explicitly stated otherwise, all indices start from 1. We use $\boldsymbol{k}$ to denote the list of keys formed with μ attributes; the size of $\boldsymbol{k}$ is C. The report from client i is denoted by $\boldsymbol{k}_i$. The m-th attribute $\boldsymbol{k}_i[\mathsf{m}]$ of $\boldsymbol{k}_i$ is an element of a group G_m. We assume that G_m includes a "dummy value" $\perp$ and that $\boldsymbol{k}_i[\mathsf{m}] \in G_\mathsf{m} \setminus \{\perp\}$. In other words, the $\perp$ value is reserved for our protocol.

There are several ways to implement the groups G_m. For example, if the value of the m-th attribute, $\boldsymbol{k}_i[\mathsf{m}]$, is an ℓ_m bit string and that $1 \dots 1$ is never used to report, we set $G_\mathsf{m} = \mathbb{Z}_2^{\ell_\mathsf{m}}$ and $\perp = 1 \dots 1$. For the rest of the paper, we will follow this notation. If all ℓ_m-bit strings must be used, we let $G_\mathsf{m} = \mathbb{Z}_{2^{\ell_\mathsf{m}}+1}$ and $\perp = 2^{\ell_\mathsf{m}}$. We let L_m be the order of G_m and let $G = G_1 \times G_2 \times \dots \times G_\mu$. Hence, $\boldsymbol{k}_i$ is an element of G.

In the case of binary representation, an ℓ-bit $\boldsymbol{k}_i$ will have μ different attributes, such that $\ell = \sum_{\mathsf{m}=1}^{\mu} \ell_\mathsf{m}$, where each attribute key is ℓ_m bits for $\mathsf{m} \in [\mu]$.

$d_1 = 00\ 010$
$d_2 = 01\ 001$
$d_3 = 00\ 000$
$d_4 = 00\ 010$
$d_5 = 10\ 010$
$d_6 = 00\ 010$
$d_7 = 01\ 110$
$d_8 = 01\ 001$
$d_9 = 10\ 000$
$d_{10} = 10\ 010$

Figure 3: A visualization of our small example: the list D with $C = 10$ reports with $\mu = 2$ attributes and total of 5 bits.

When we bucketize for an attribute index m with ℓ_{m} bits, we obtain $L_{\mathsf{m}} = 2^{\ell_{\mathsf{m}}}$ buckets, which we represent as $\{\mathcal{B}_1, \ldots, \mathcal{B}_{L_{\mathsf{m}}}\}$. Among these buckets, one bucket is reserved for the dummy value. We denote the bucket of dummies by $\mathcal{B}_{\perp}$.

We denote the vector of reports (keys) by $D = (d_i = \boldsymbol{k}_i)_{i \in [C]}$, where d_i (or $\boldsymbol{k}_i$ here) denotes the i-th report in D, while d_i^{b} denotes the server b's share of the i-th report. We denote server b's shares of the full dataset D by D^{b}.

2.1.1 Example

Throughout the paper, we will describe the high-level idea of the protocols with a small example. We work with 10 client reports, where each report consists of two attributes: Gender (represented with 2 bits to allow "he/she/they" and $\perp$) and Category (represented with 3 bits). In this example, we take $G_1 = \mathbb{Z}_2^2$ and $G_2 = \mathbb{Z}_2^3$; addition is bitwise XOR. Since we have $C = 10$, $\boldsymbol{k}$ will be formed with 10 reported keys. Each key will have $\mu = 2$ attributes and 5 bits in total: $\boldsymbol{k}_i[1] = \mathsf{x}_1\mathsf{x}_2$ and $\boldsymbol{k}_i[2] = \mathsf{y}_1\mathsf{y}_2\mathsf{y}_3$. If we want to build a histogram on attribute "Gender" ($\mathsf{m} = 1$), we will obtain $L_{\mathsf{m}} = 2^2$ buckets: $\{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$ where $\mathcal{B}_4$ is reserved for dummy records and filled with values for reports with $\boldsymbol{k}_i[1] = 11$. An example of a corresponding dataset $\boldsymbol{k}$ (without considering any secret sharing yet) is given in Figure 3. Notice that there are no records with $\boldsymbol{k}_i[1] = 11$ or $\boldsymbol{k}_i[2] = 111$, as these buckets are reserved for dummy records.

2.2 Secret Sharing

The GMW protocol [16] was the first multi-party computation protocol based on secret sharing. In GMW, a secret is information-theoretically shared between multiple parties. Let G denote a finite additive group. In the two party case, a client willing to share a value $k \in G$ to two servers will first uniformly sample $r \leftarrow G$, send r to one server, and $k - r$ to the other server. Neither share alone reveals any information about the secret value k. Such a scheme works for both Boolean circuits and arithmetic circuits; it requires no communication for the

$d_1^1 = 10\ 011$
$d_2^1 = 00\ 011$
$d_3^1 = 01\ 111$
$d_4^1 = 11\ 110$
$d_5^1 = 01\ 001$
$d_6^1 = 10\ 001$
$d_7^1 = 00\ 011$
$d_8^1 = 01\ 101$
$d_9^1 = 00\ 011$
$d_{10}^1 = 11\ 101$

(a) Shares of $\mathcal{S}_1$: D^1

$d_1^2 = 10\ 001$
$d_2^2 = 01\ 010$
$d_3^2 = 01\ 111$
$d_4^2 = 11\ 100$
$d_5^2 = 11\ 011$
$d_6^2 = 10\ 011$
$d_7^2 = 01\ 101$
$d_8^2 = 00\ 100$
$d_9^2 = 10\ 011$
$d_{10}^2 = 01\ 111$

(b) Shares of $\mathcal{S}_2$: D^2

Figure 4: Secret shares of D held by $\mathcal{S}_1$ and $\mathcal{S}_2$.

addition of two secret values, or addition or multiplication with public constant values.

Example

Continuing with our example, we secret share the reports in D for $\mathcal{S}_1$ and $\mathcal{S}_2$ as follows: $d_i = d_i^1 \oplus d_i^2$, for $i \in [10]$. We depict the shares in Figure 4.

2.3 Differential Privacy

Differential privacy [11, 12] protects the privacy of individual records in a database, while still allowing meaningful queries to be made. For each (ordered) dataset $D \in \chi^C$, we define an (unordered) database $\bar{D} \in \mathbb{N}^\chi$ by $\bar{D}(j) = \#\{i : D_i = j\}$. Let $\mathcal{M}$ be a randomized algorithm with domain $\mathbb{N}^\chi$ and let $\bar{D}, \bar{D}' \in \mathbb{N}^\chi$ be two neighboring databases that differ on only one record.[3] We say that a mechanism $\mathcal{M}$ is (ϵ, δ)-differentially private ((ϵ, δ)-DP) for parameters $\epsilon \geq 0$ and $\delta \in [0, 1]$ if for any $S \subseteq \mathsf{Range}(\mathcal{M})$ and any neighboring D and D',

$$\mathsf{Pr}[\mathcal{M}(\bar{D}) \in S] \leq e^\epsilon \mathsf{Pr}[\mathcal{M}(\bar{D}') \in S] + \delta$$

The property of differential privacy is maintained through *post-processing*. Informally, it means that once differential privacy is achieved for the output of a particular query, the data curator can make any computations with this output without violating the formal differential privacy guarantees.

2.3.1 Laplace Mechanism

The Laplace mechanism is one of classical DP mechanisms, where we add noise drawn from the Laplace distribution to the output of statistical aggregate. The probability density function of Laplace distribution is $\mathsf{Lap}(X, b) =$

[3]Formally, neighboring means L_1 norm of two databases $\|\bar{D} - \bar{D}'\|_1 \leq 1$

$(1/2b)\ \exp(-|X|/b)$, with zero mean and standard deviation $\sigma = \sqrt{2b^2}$, for a parameter b. For any function $f : \mathbb{N}^{\chi} \to \mathbb{R}^L$, the *Laplace mechanism* $\mathcal{M} : \mathbb{N}^{\chi} \to \mathbb{R}^L$ is defined as

$$\mathcal{M}(f(\bar{D})) = f(\bar{D}) + Y\,.$$

Here $Y_j \leftarrow \mathsf{Lap}(X, \Delta f/\epsilon)$ are the i.i.d noise terms randomly drawn from the Laplace distribution for each bucket j and Δf is the sensitivity of the function f, which quantifies how a single record can change the output. Since we are interested in histogram queries, L is the number of buckets, and the sensitivity $\Delta f = 1$. Thus, the Laplace distribution we consider from now on is $\mathsf{Lap}\,(X, 1/\epsilon)$.

2.4 Oblivious Random Shuffling

We will make use of an oblivious shuffling protocol that runs between two or three servers. It inputs a dataset, which is initially additively secret shared between two servers and outputs additive secret shares of a shuffled dataset. Obliviousness means that none of the servers learns the mapped positions before and after the shuffling for any element in the dataset.

To achieve obliviousness, a prior work by Chase *et al.* [9] uses the idea of an oblivious permutation for two-party oblivious random shuffling. In their work, each party samples a permutation and initiates an oblivious permutation to permute a secret shared dataset. Since the dataset is permuted twice, with each permutation known only by one of the parties, the mapping of the elements positions before and after the permutations are kept secret. However, a third-party approach could be used to improve the efficiency of the protocol.

Mohassel *et al.* [19] proposed an oblivious permutation protocol in the honest majority three-party setting with linear computation and communication cost. We will instantiate a modified version of their protocol in Section 3.2.

3 Subroutines

3.1 Differential Privacy Mechanism with Constraints

To achieve differential privacy (DP) in histograms, we add noise to the counts of each bucket in our Bucketization protocol. Ideally, we can achieve it with Laplace mechanism. However, the way we design our protocol does not allow to apply it in a straightforward way. Briefly, our protocol requires to generate positive integers as noise because sampled noise becomes the number of dummy records generated for each bucket. So, our ultimate aim is to find a way to allow adding both positive or negative noise sampled from standard Laplace distribution. Therefore, we will tweak the Laplace mechanism in a way that will allow us to use Laplace noise as it is. This will make the analysis of DP simpler.

Alternatively, we could follow the proposal from Medina *et al.* [18] as a general framework for private optimization without constraints violations. Clearly, their work could be used in order to achieve discrete positive noise, however, we could achieve DP with a simpler method, which we describe next.

3.1.1 Truncated and Shifted Laplace Mechanism

As we stated before, our aim is to use the standard Laplace distribution in a way that it leads us to generate discrete positive or negative noise. First, we define the truncation of Laplace. We sample a noise n from $\mathsf{Lap}\,(X, 1/\epsilon)$: $n \leftarrow \mathsf{Lap}(X, 1/\epsilon)$ (rounded to the closest integer[4]) which could be negative. If $n < -M$, we sample again until $n \geq -M$. We obtain a PDF for n which is $\Pr(X) = 0$ if $X < -M - 1/2$ and $\Pr(X) = \frac{\mathsf{Lap}(X,1/\epsilon)}{1-\mathsf{p}}$ otherwise, with p defined as resampling probability (which we will define shortly). Finally, we define shifting as follows: we add M to n. These steps ensure that $n + M$ is always non-negative and we obtain $n + M$ as number of dummy records. Note that M is a public parameter. Later on, when the buckets are computed, we will subtract the value M from each counts and it will leave us with the original Laplace noise n added to the buckets.

Let $-M - \frac{1}{2}$ define a truncation point in Laplace distribution. Given the PDF of standard Laplace distribution, we define resampling probability p as:

$$\mathsf{p} = \Pr\left[X < -M - \frac{1}{2}\right] = \int_{-\infty}^{-M-1/2} \mathsf{Lap}\left(X, \frac{1}{\epsilon}\right) = \frac{1}{2} e^{-\epsilon(M+1/2)}$$

Then, we compute $M = -(1/\epsilon)\ln(2\mathsf{p}) - 1/2$. As an example, if we want $\mathsf{p} \approx 1/1\,000\,000$, M becomes $M = \lceil 13.12/\epsilon - 1/2 \rceil$.

3.1.2 Computation of average dummy records

We define the added noise as $n + M$ dummy records. We compute the average of $n + M$ as $\bar{M}$ as follows:

$$\mathbb{E}(n) = \int_{-M-1/2}^{\infty} \frac{x\,\mathsf{Lap}(x)}{1-\mathsf{p}}\,dx = \frac{\mathsf{p}}{1-\mathsf{p}}\left(\frac{1}{\epsilon} + M + \frac{1}{2}\right)$$

$\bar{M} = M + \frac{\mathsf{p}}{1-\mathsf{p}}\,(1/\epsilon + M + 1/2)$ which is very close to M.

We provide a suggested way to implement noise generation in Appendix A.1. The protocol for noise generation is shown in Figure 5.

3.1.3 Example

We continue our example from previous Section 2.1. We want to build a histogram on the attribute “Gender” ($\mathsf{m} = 1$) with $L_{\mathsf{m}} = 4$ buckets $\{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$. In our Π_{NoiseGen} protocol, **Noise generation** step will sample a number for each

[4]This is safe due to the Post-Processing Theorem.

Protocol Π_{NoiseGen}

Parameters. An attribute index m. L_{m} buckets. There exists privacy parameters (ϵ, δ). There exists a Laplace distribution $\mathsf{Lap}(X, 1/\epsilon)$. For a resampling probability p such that $\delta = \mathsf{p}\frac{e^{\epsilon}}{1-\mathsf{p}}$, we define $M = -\frac{1}{\epsilon}\ln(2\mathsf{p}) - \frac{1}{2}$.

Input. Each server $\mathcal{S}_{\mathsf{b}}$ inputs $(\mathsf{dp}, D^{\mathsf{b}}, \mathsf{m})$ where D^{b} is the full dataset share of the server b and m is the (queried) attribute index of the key.

Noise generation. For each $\mathsf{b} \in \{1, 2\}$, for each bucket $j \in [L_{\mathsf{m}}]$, $\mathcal{S}_{\mathsf{b}}$ randomly samples noise (until it is larger than $-M - \frac{1}{2}$ by rejection sampling) from the distribution $\mathsf{Lap}(X, 1/\epsilon)$ and rounds it to the nearest integer. We call this rounded noise as n_j^{b}. All the noise values are recorded as $\mathcal{N}^{\mathsf{b}} = (n_j^{\mathsf{b}} + M)_{j \in [L_{\mathsf{m}}]}$.

Generating dummy records. For each $\mathsf{b} \in \{1, 2\}$ and each bucket $j \in [L_{\mathsf{m}}]$, $\mathcal{S}_{\mathsf{b}}$ creates $n_j^{\mathsf{b}} + M$ dummy records as follows: for $i \in [n_j^{\mathsf{b}} + M]$, set $\boldsymbol{k}_{i,j}^{\mathsf{b}}[\mathsf{m}] = j$ and $\boldsymbol{k}_{i,j}^{\mathsf{b}}[v] \leftarrow \perp (\in G_v)$ for $v \in [\mu] \backslash \{\mathsf{m}\}$ (which form one dummy record $\boldsymbol{k}_{i,j}^{\mathsf{b}}$). $\mathcal{S}_{\mathsf{b}}$ forms all of $n^{\mathsf{b}} = \sum_{j \in [L]} (n_j^{\mathsf{b}} + M)$ dummy records $d_{i,j}^{\mathsf{b}} = \boldsymbol{k}_{i,j}^{\mathsf{b}}$ as $D_{\mathsf{dum}}^{\mathsf{b}}$.

Appending shares to dummy records. Set D_{priv}^1 and D_{priv}^2 as follows: $(D_{\mathsf{priv}}^1)_i = D_i^1$ for $i < C$; $(D_{\mathsf{priv}}^1)_i = (D_{\mathsf{dum}}^1)_{i-C}$ for $C \leq i < C + n^1$; and $(D_{\mathsf{priv}}^1)_i = 0$ for $C + n^1 \leq i < C + n^1 + n^2$. $\mathcal{S}_2$ computes similarly except that it puts all 0 keys before the dummy records of $\mathcal{S}_1$.

Output. Each server $\mathcal{S}_{\mathsf{b}}$ outputs $D_{\mathsf{priv}}^{\mathsf{b}}$.

Figure 5: The protocol of DP noise generation.

$d_1^1 = 10\ 011$
$d_2^1 = 00\ 011$
$d_3^1 = 01\ 111$
$d_4^1 = 11\ 110$
$d_5^1 = 01\ 001$
$d_6^1 = 10\ 001$
$d_7^1 = 00\ 011$
$d_8^1 = 01\ 101$
$d_9^1 = 00\ 011$
$d_{10}^1 = 11\ 101$
$d_{11}^1 = 00\ 111$
$d_{12}^1 = 00\ 111$
$d_{13}^1 = 01\ 111$
$d_{14}^1 = 11\ 111$
$d_{15}^1 = 11\ 111$
$d_{16}^1 = 00\ 000$
$d_{17}^1 = 00\ 000$

(a) D^1_{priv}

$d_1^2 = 10\ 001$
$d_2^2 = 01\ 010$
$d_3^2 = 01\ 111$
$d_4^2 = 11\ 100$
$d_5^2 = 11\ 011$
$d_6^2 = 10\ 011$
$d_7^2 = 01\ 101$
$d_8^2 = 00\ 100$
$d_9^2 = 10\ 011$
$d_{10}^2 = 01\ 111$
$d_{11}^2 = 00\ 000$
$d_{12}^2 = 00\ 000$
$d_{13}^2 = 00\ 000$
$d_{14}^2 = 00\ 000$
$d_{15}^2 = 00\ 000$
$d_{16}^2 = 00\ 111$
$d_{17}^2 = 11\ 111$

(b) D^2_{priv}

Figure 6: Output of Π_{NoiseGen} on small example dataset D. Last 7 entries in D_{priv} are dummy records; 5 of them were are added by $\mathcal{S}_1$ and 2 of them by $\mathcal{S}_2$.

of these 4 buckets (with $\mathcal{B}_4$ being the reserved bucket for dummy records). Suppose the noise vector $\mathcal{N}^1 = (2, 1, 0, 2)$ for $\mathcal{S}_1$. It means that total $n^1 = 5$ records will be created: two records $\boldsymbol{k}^1_{1,1}, \boldsymbol{k}^1_{2,1}$ to append for $\mathcal{B}_1$; one record $\boldsymbol{k}^1_{1,2}$ to append for $\mathcal{B}_2$; two records $\boldsymbol{k}^1_{1,4}, \boldsymbol{k}^1_{2,4}$ to append for $\mathcal{B}_4$.

In step **Generating dummy records**, each fake report will have the form $\boldsymbol{k}^1_{i,j} = [j||111]$ where j represented in binary with 2 bits and other attribute set to reserved value 111. Same steps repeat for $\mathcal{S}_2$ with different total noise vector $\mathcal{N}^2$ and n^2.

Finally, in the step **Appending shares to dummy records**, these 5 dummy reports from $\mathcal{S}_1$, as well as n^2 fake reports (all key bits filled with 0's) will be appended to end of true reports from 10 clients. This way, we let the communication between $\mathcal{S}_1$ and $\mathcal{S}_2$ be the exchange of n^1 and n^2 only. Let the noise vector of $\mathcal{S}_2$ be $\mathcal{N}^2 = (1, 0, 0, 1)$. We depict the noise addition following our examples in Figure 6.

Let's continue further with our example. Suppose we reveal the buckets for the first attribute. Then, the servers only reveal the first attributes and puts the **shared** second attributes in corresponding buckets:

$$\mathcal{B}_1 = \{d^{\mathsf{b}}_1, d^{\mathsf{b}}_3, d^{\mathsf{b}}_4, d^{\mathsf{b}}_6, d^{\mathsf{b}}_{11}, d^{\mathsf{b}}_{12}, d^{\mathsf{b}}_{16}\}$$

$$\mathcal{B}_2 = \{d^{\mathsf{b}}_2, d^{\mathsf{b}}_7, d^{\mathsf{b}}_8, d^{\mathsf{b}}_{13}\}$$

$$\mathcal{B}_3 = \{d^{\mathsf{b}}_5, d^{\mathsf{b}}_9, d^{\mathsf{b}}_{10}\}$$

$$\mathcal{B}_4 = \{d^{\mathsf{b}}_{14}, d^{\mathsf{b}}_{15}, d^{\mathsf{b}}_{17}\}\,.$$

Again, note that we do not reveal the second attributes. Then, the histogram is built with the counts of $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ after discarding the dummy bucket.

3.2 Oblivious Random Shuffling

In our work, we take the Mohassel *et al.* [19] protocol and modify it into an efficient honest-majority three-party oblivious random shuffling protocol achieving linear complexity. All the operations are information-theoretic, thus having only a small computational overhead.

The oblivious random shuffling protocol is described in Figure 7 and depicted in Figure 8. In **Initialize** step, we allow only one server to sample the permutation and random masks and then send it to the other server. We do not need commitment and interactions because of the third server we introduced. This is because when one of the servers is malicious, the sampling from the (third) honest server is enough to create uniform distribution. This would not be true if we had two-party computation.

Protocol Π_{RandShuf}

Notation. When the operator $\{+,-\}$ are applied to vectors, they mean element-wise addition and subtraction.

Input. For $\mathsf{b} \in \{1,2\}$, S_{b} inputs $(\mathsf{shuffle}, D^{\mathsf{b}})$ where $D^{\mathsf{b}} := \{(\boldsymbol{k}^{\mathsf{b}}_i)\}_{i\in[C]}$, with $\boldsymbol{k}^{\mathsf{b}}_i \in G$ (G is defined as a product group: $G = G_1 \times \ldots \times G_\mu$).

Initialize. For $(\mathsf{b}_1, \mathsf{b}_2) \in \{(1,2),(2,3),(1,3)\}$, S_{b_1} and S_{b_2} jointly sample (only one of the corresponding server samples and sends privately to the other server) a permutation $\pi_{b_1 b_2}$ and a random vector $R_{b_1 b_2} \in G^C$.

Shuffling.

1. $\mathcal{S}_2$ computes $A := \pi_{23}(\pi_{12}(D^2) + R_{12}) + R_{23}$ and sends A to $\mathcal{S}_1$.
2. $\mathcal{S}_1$ computes $B := \pi_{12}(D^1) - R_{12}$ and sends B to $\mathcal{S}_3$. It also computes $A' := \pi_{13}(A) - R_{13}$ and outputs A'.
3. $\mathcal{S}_3$ computes $B' := \pi_{13}(\pi_{23}(B) - R_{23}) + R_{13}$ and outputs B'.

Output. $\mathcal{S}_1$ outputs A' and S_3 outputs B'.

Figure 7: The protocol of oblivious random shuffling.

Informally, what we prove is that the views of any corrupt party can be perfectly simulated. At the initialization phase, each pair of parties jointly sample a random injective permutation and a random mask vector. They can

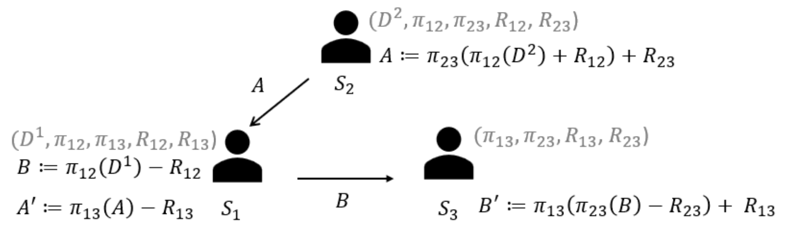
$(D^2, \pi_{12}, \pi_{23}, R_{12}, R_{23})$
$A \coloneqq \pi_{23}(\pi_{12}(D^2) + R_{12}) + R_{23}$
S_2
A
$(D^1, \pi_{12}, \pi_{13}, R_{12}, R_{13})$
$B \coloneqq \pi_{12}(D^1) - R_{12}$
$A' \coloneqq \pi_{13}(A) - R_{13}$
S_1
B
$(\pi_{13}, \pi_{23}, R_{13}, R_{23})$
S_3
$B' \coloneqq \pi_{13}(\pi_{23}(B) - R_{23}) + R_{13}$

Figure 8: Illustration of Protocol Π_{RandShuf}. Gray tuples indicates the inputs known to each server.

be simulated by the simulator uniformly samples and sends to the adversaries random injective permutations and mask vectors. The simulation of shuffling phase is done as follows.

1. Corrupt $\mathcal{S}_1$: The only message that $\mathcal{S}_1$ receives is A from $\mathcal{S}_2$. $A \coloneqq \pi_{23}(\pi_{12}(D^2) + R_{12}) + R_{23}$, where π_{23} and R_{23} are known to S_1. Since the random vector R_{23} masks the permuted shares, A is indistinguishable from a random vector from S_1's view. The simulator can replace it with a random vector of same size.

2. Corrupt $\mathcal{S}_2$: $\mathcal{S}_2$ receives no messages, thus there is no need to simulate.

3. Corrupt $\mathcal{S}_3$: The only message that $\mathcal{S}_3$ receives is B from $\mathcal{S}_2$. As is the same situation to $\mathcal{S}_1$'s, B is indistinguishable from a random vector from $\mathcal{S}_3$'s view, so the simulator can replace it with a random vector of the same size.

This informal analysis ignores the final output. Later, in the formal treatment, we will assume collusion between Reporting Origin and one of the aggregate servers. Reporting Origin learns the final output and we assume the leakage of A' and B', as well. The formal privacy of the protocol in the semi-honest server model is proven in Appendix A.2.

4 Secure Report Bucketization For Histograms

The goal of report bucketization is to arrange reports collected from C clients into appropriate buckets according to a subset of attributes encoded in set of keys $\boldsymbol{k}$, while preserving privacy of the individual reports. For an attribute $\boldsymbol{k}[\mathsf{m}]$, with ℓ_{m} bits, there are $L_{\mathsf{m}} = 2^{\ell_{\mathsf{m}}}$ buckets in total. Our full protocol involves four parties: a Reporting Origin $\mathcal{R}$ and three helper servers $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$.

We assume that no two servers in the aggregation system collude, but $\mathcal{R}$ may collude with one of the servers. We build our system with three helper servers in a way that two of them, say $\mathcal{S}_1$ and $\mathcal{S}_2$, receive the secret shared inputs, and other two, say $\mathcal{S}_1$ and $\mathcal{S}_3$, output the results of the histogram computations. In between, our design is to run DP noise generation with two helper servers and oblivious random shuffling with three helpers to enable faster protocols.

The input to our protocol is a vector of reports $D = (\boldsymbol{k}_i)_{i \in [C]}$, which is initially secret shared between two non-colluding helper servers. To ensure that the helpers get the same ordering of reports, the clients may attach an ephemeral ID along with the encryption of the shared reports (under the public key of the servers) before passing them to the servers. This is equivalent to including a *Leader* server, which is a trusted entity whose only job is to maintain the order of the records.[5] At the end of the protocol, two helpers obtain a new secret shared vector $D_{\mathsf{priv}} = (\boldsymbol{k}_i)_{i \in [C+n']}$, where all reports having the same set of attributes (called "buckets") are stored consecutively in D_{priv}. Note that the size of D_{priv} becomes $C + n'$ with additional n' dummy reports with dummy attributes. The n' dummy reports were created to ensure that the outputs revealed to the reporting origin and helper servers are differentially private.

4.1 Private Histogram Protocol Description

In this section, we describe our Bucketization protocol as a full procedure $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ in Figure 10. Let $D = (d_i = \boldsymbol{k}_i \in G)_{i \in [C]}$ be the dataset of reports (report keys) collected from clients. $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ takes D and a query index m (to indicate on which attribute the histogram is built) as an input.

The procedure is triggered by $\mathcal{R}$, with helper servers $\mathcal{S}_1$ and $\mathcal{S}_2$ receiving the shares of the reports as $D^1 = \left(d_i^1 = \boldsymbol{k}_i^1\right)_{i \in [C]}$ and $D^2 = \left(d_i^2 = \boldsymbol{k}_i^2\right)_{i \in [C]}$, such that $\boldsymbol{k}_i = \boldsymbol{k}_i^1 + \boldsymbol{k}_i^2$ for all $i \in [C]$.

After secret sharing the reports in D, the first step is to achieve differential privacy by $\mathcal{S}_1$ and $\mathcal{S}_2$ independently adding dummy reports as noise, which is done by invoking the noise generation subroutine, as defined in Figure 5. This step inputs the shares of the dataset and the query attribute index m, and outputs a new dataset with appended dummy records $D_{\mathsf{priv}} = (\boldsymbol{k}_i)_{i \in [C+n']}$. The look of the protocol with addition of dummy records (and oblivious shuffling, which we will describe shortly) is depicted in Figure 9.

After generating D_{priv}, the servers $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ execute an honest-majority three-party oblivious random shuffling protocol, as defined in Figure 7. At the end of this step, the real reports and dummy reports will be mixed up by the random permutation so that none of the helpers can trace back any report to the original report, neither can they distinguish between the authentic reports and dummy reports. The output will be a permuted dataset $D_{\mathsf{priv\_perm}} = (\boldsymbol{k}'_i)_{i \in [C+n']}$, secret shared between the helper servers $\mathcal{S}_1$ and $\mathcal{S}_3$.

For each $i \in [C + n']$, the helpers reveal the selected attribute $\boldsymbol{k}_i[\mathsf{m}]$ and bucketize the reports according to the bucket where the key attribute belongs to: $\{\mathcal{B}_1, \ldots, \mathcal{B}_{L_{\mathsf{m}}}\}$.

Finally, the buckets with size less than the pruning threshold t, as well as the dummy bucket $\mathcal{B}_\perp$, are discarded. A value for t is typically decided based

[5]This is also aligned with the Internet-Draft [13].

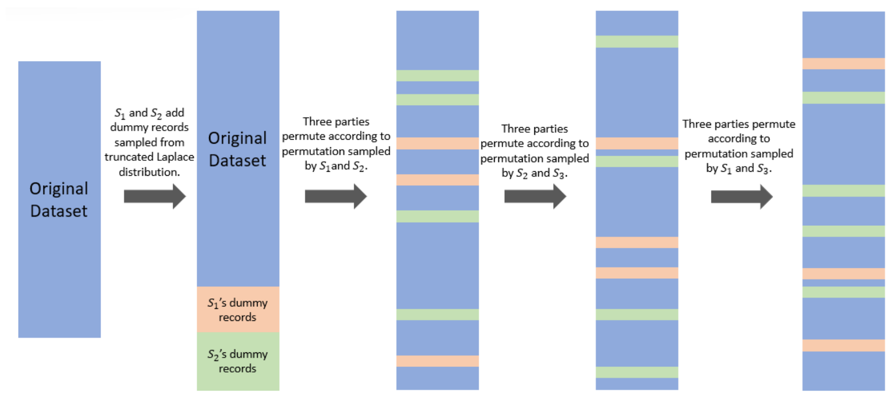
Original Dataset
S_1 and S_2 add dummy records sampled from truncated Laplace distribution.
Original Dataset
S_1's dummy records
S_2's dummy records
Three parties permute according to permutation sampled by S_1 and S_2.
Three parties permute according to permutation sampled by S_2 and S_3.
Three parties permute according to permutation sampled by S_1 and S_3.

Figure 9: The workflow of Bucketization starting with dummy record addition and oblivious shuffling. The dummy records are colored to show the shuffling, due to the XOR of random masks, they will not be traced. This flow will run with three servers, which carry the shares of the original records.

on the DP parameter $\bar{M}$, as $\bar{M}$ indicates the dummy records added on average to each bucket. Since the output shares remain with $\mathcal{S}_1$ and $\mathcal{S}_3$, we apply an organizing step to put the shares back in place (*i.e.*, reshare to $\mathcal{S}_1$ and $\mathcal{S}_2$) to run the protocol again if needed, as we will describe next. However, this resharing is not necessary in implementations: we include it in the protocol in order to be able to use $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ repeatedly.

4.2 Layered Bucketization

In this section we will describe how our method makes the aggregate histogram computations on various attributes very flexible. In short, our protocol can be run on a single or a few attributes, not necessarily on all attributes. We achieve this flexibility by recursively calling the $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ protocol.

Our method enables running queries such as

$$\begin{array}{c}\mathsf{SELECT}\ \ \mathsf{COUNT}(\text{Category})\ \ \ \mathsf{FROM}\ \ \text{D}\\ \mathsf{WHERE}\ \ \text{Gender} = \text{“}\mathsf{She}\text{”}\ \ \ \mathsf{GROUP\ BY}\ \ \text{Category}\end{array}$$

as long as there is enough data in the corresponding bucket. Moreover, such an approach enables a performance gain for attributes with large key sizes. For example, if one attribute has 32 bits, we can apply layered pruning to speed up the protocol. More importantly, if the domain of the attributes is sparse (say 2^{16} keys represented with 32 bits), layering the Bucketization of 32 bits keys into three (10, 10, and 12 bits correspondingly) and pruning the buckets after Bucketization of each layer would allow us to run the protocol efficiently even for relatively large key sizes, which appear with sparse domains. A formal description of the layered algorithm is given in Algorithm 1.

Concretely, when the first layer is done with m_1, only the corresponding bits

Protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$

Parameters. C as the total number of collected reports.

Input. A query index m to indicate which attribute to bucketize on; shares of $\mathcal{S}_1$ and $\mathcal{S}_2$ as $D^1 = \left(\boldsymbol{k}_i^1\right)_{i\in[C]}$ and $D^2 = \left(\boldsymbol{k}_i^2\right)_{i\in[C]}$, such that $\boldsymbol{k}_i = \boldsymbol{k}_i^1 + \boldsymbol{k}_i^2$; a threshold t.

Initialization. Each server receives and decrypts their share. They discard the shares if they are not in G.

Bucketization.

1. (**Differential privacy**) $\mathcal{S}_1$ and $\mathcal{S}_2$ invoke the protocol Π_{NoiseGen} from Figure 5 on input $(\mathsf{dp}, D^1, \mathsf{m})$ and $(\mathsf{dp}, D^2, \mathsf{m})$. For $\mathsf{b} \in \{1, 2\}$, $\mathcal{S}_{\mathsf{b}}$ sends $(\mathsf{dp}, D^{\mathsf{b}}, \mathsf{m})$ and updates D^{b} by the output of the protocol as $D^{\mathsf{b}}_{\mathsf{priv}} = \left(\boldsymbol{k}_i^{\mathsf{b}}\right)_{i\in[C+n']}$ with appended n' dummy records.
2. (**Random shuffling**) $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ invokes the protocol Π_{RandShuf} from Figure 7 on inputs $(\mathsf{shuffle}, D^1_{\mathsf{priv}})$, $(\mathsf{shuffle}, D^2_{\mathsf{priv}})$. It outputs $D^b_{\mathsf{priv\_perm}} = \left(\boldsymbol{k}'^{b}_i\right)_{i\in[C+n']}$ to $\mathcal{S}_{\mathsf{b}}$ for $b \in \{1, 3\}$.
3. (**Bucketizing**) For $i \in [C+n']$, $\mathcal{S}_1$ and $\mathcal{S}_3$ reveal their shares of $\boldsymbol{k}'_i[\mathsf{m}]$ and recover the bucket identifier id_i (decimal value of $\boldsymbol{k}'_i[\mathsf{m}]$). Allocate L_{m} empty buckets $\{\mathcal{B}_1, \ldots, \mathcal{B}_{L_{\mathsf{m}}}\}$. For $i \in [C+n']$, $\mathsf{b} \in \{1, 3\}$, $\mathcal{S}_b$ puts the record $\boldsymbol{k}'^{b}_i$ into the corresponding bucket $\mathcal{B}_{\mathsf{id}_i}$.
4. (**Pruning and organizing**) For each bucket revealed in previous step, discard the bucket $\mathcal{B}_j$ if $|\mathcal{B}_j| < t$ which has small number of records and the dummy buckets (indexed as $j = \perp$) along with all the records in it. For the shares of noisy database from $\mathcal{S}_1$ and $\mathcal{S}_3$, $\mathcal{S}_3$ generates two secret shares of $D^2_{\mathsf{priv\_perm}}$ and sends the shares to $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively; $\mathcal{S}_1$ combines the share he receives from $\mathcal{S}_3$ with $D^1_{\mathsf{priv\_perm}}$ so that $\mathcal{S}_1$ and $\mathcal{S}_2$ have the new shares of the $D_{\mathsf{priv\_perm}}$.

Output. Output buckets which remain after pruning, their counts, and the attribute value of index m for each bucket. Servers keep a private output which shares datasets of records for each bucket so that they can be used as input for further instance of the protocol.

Figure 10: Our Histogram Protocol based on Oblivious Shuffling

are revealed. Then, the procedure bucketizes on the next attribute index m_2 by generating noisy reports, setting bucket IDs with the bits of m_2 and sampling

Algorithm 1 Layered $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$

Input: A list of attribute indices $\mathcal{M} = \{\mathsf{m}_1, \ldots, \mathsf{m}_\lambda\}$; a shared dataset for $\mathcal{S}_1$ and $\mathcal{S}_2$.
Output: Histogram on attributes in $\mathcal{M}$.

```
1: procedure LAYERED($\mathcal{M}, \mathsf{label}, i, D^1, D^2$)
2:     if $(\mathsf{label}, i) = (\perp, \perp)$ then
3:         Set $\mathsf{label} = \text{null}, i = 1$.
4:     end if
5:     if $i \leq \lambda$ then
6:         Call $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ with inputs $(\mathsf{m}_i, D^1, D^2)$.
7:         for each produced bucket, the value $v$ of $\boldsymbol{k}[\mathsf{m}_i]$, $D^1, D^2$ do
8:             Call LAYERED($\mathcal{M}, \mathsf{label} + v, i + 1, D^1, D^2$).
9:         end for
10:    else
11:        Output $\mathsf{label}$ and $\mathsf{count}$(D).
12:    end if
13: end procedure
```

other attribute bits at random. Since the $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ call feeds other attributes into the random shuffling procedure, these attributes go through a "reshare" process. It means that each key a server received will be a permuted dataset masked with a random vector. So, the original revealed buckets for attribute m_1 as public information will not cause any information leakage.

4.2.1 Example

We continue with our toy example. The reports consist of 5-bit keys representing two attributes with 2 and 3 bits respectively. Our protocol allows to run the bucketization on the first attribute Gender (with 2 bits) into 3 buckets $B = \{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3\}$ after discarding the dummy bucket, prune the buckets which have number of reports below some threshold t, $B' = \{\mathcal{B}_i : |\mathcal{B}_i| \geq t\}$, and continue bucketization for the next attribute for each remaining bucket in B' in the next layer.

When we open the first attribute (*i.e.*, only the first two bits), the bucket $\mathcal{B}_1$ will contain $\{00\ 010, 00\ 000, 00\ 010, 00\ 010, 00\ 111, 00\ 111, 00\ 111\}$, yielding 7 records where the second attribute is still secret shared. Then, suppose the protocol goes to the second layer on $\mathcal{B}_1$ where the bucketization is run on the second attribute. It would mean that both $\mathcal{S}_1$ and $\mathcal{S}_2$ will create a noise vector for all possible buckets including the dummy bucket. Let's focus on bucket $\mathcal{B}'_3$, which counts the reports where $\boldsymbol{k}_i[1] = 00$ and $\boldsymbol{k}_i[2] = 010$. Suppose $\mathcal{S}_1$ sampled 2 and $\mathcal{S}_2$ sampled 1 to add in that bucket $\mathcal{B}'_3$. Then, the output of that bucket will have $3 + 2 + 1$ reports, where 3 comes from real reports (neither $\mathcal{S}_1$ nor $\mathcal{S}_2$ know these true report counts), and additional 2 and 1 come from the dummy reports. It means that the query that asks for counts, where $\boldsymbol{k}[1] = 00$

and $\boldsymbol{k}[2] = 010$, will output 6 (instead of 3). Finally, in the second layer, more dummy records are added with $\boldsymbol{k}_i[2] = 111$ and then the second attribute values are revealed. After revealing, the bucket corresponding to 111 and the buckets with less than t counts will be discarded.

In an extreme case we consider every single bit of the key as an attribute, in which case our layered protocol runs as a (pruned) binary tree descent. We analyse the complexity of the layered protocol, as well as this extreme case, next.

4.2.2 Complexity of Algorithm 1

For each layer $i = 1, \ldots, \lambda$, we consider every call to $\textsc{Layered}(\cdot, \cdot, i, \cdot, \cdot)$. The average complexity is the size of the input dataset to this call added with $L_i \bar{M}$ dummy records. The size of the input dataset is the size of a bucket at the upper $(i-1)$-th layer which is not pruned (larger than t) and not a dummy bucket. We analyse the complexity in two different cases.

In the first case, we consider t too low, *i.e.*, $t \leq M$ so that the number of selected buckets is exponential, or that there is no pruning, i.e. $t = 0$. In this case, the number of selected buckets at layer $i-1$ is upper bounded by $L_1 \cdot L_2 \cdot \ldots \cdot L_{i-1}$. The sum of the bucket sizes is $C + L_1 \cdot \ldots \cdot L_{i-1} \bar{M}$ on average. By summing over all layers, we obtain total complexity of $O(\lambda C + \bar{M} L_1 \ldots L_\lambda)$ (comparable to $O(\ell\, C + \bar{M}\, B)$). This is the complexity of the full histogram.

In the second case, we consider t large enough $(t > M)$ to prune effectively. We consider every possible bucket $\mathcal{B}_1, \ldots, B_{L_1 L_2 \ldots L_{i-1}}$ at layer $i-1$. We denote by a_j the number of true records in bucket $\mathcal{B}_j$ and by $X_j = \lfloor M + Z_j \rceil$, the number of added dummy records, where Z_j follows the truncated Laplace distribution. Finally, we let Y_j be the number of dummy buckets which are added in $\mathcal{B}_j$ at layer i. The complexity to treat $\mathcal{B}_j$ at layer i is bounded by $a_j + X_j + Y_j$ if $a_j + X_j \geq t$. Thus, the complexity to treat layer i is $\sum_j (a_j + X_j + Y_j) \cdot 1_{a_j + X_j \geq t}$. Because $\sum_j a_j = C$, this complexity is bounded by $C + \sum_j (X_j + Y_j) \cdot 1_{a_j + X_j \geq t}$. The coins for X_j and Y_j are independent. Since we want to compute the average complexity, we can directly average Y_j and get a complexity of $C + S$ with $S = \sum_j \mathbb{E}\left((X_j + L_i \bar{M}) \cdot 1_{a_j \geq t - X_j}\right)$. We can show that all buckets such that $a_j < t - M$ have little influence on the sum (either there are a few with high a_j or a_j is so low that X_j has too little chance to exceed $t - a_j$). The sum over buckets such that $a_j \geq t - M$ has a number of terms bounded by $\frac{C}{t-M}$ and is bounded by $\left(\bar{M} + L_i \bar{M}\right) \frac{C}{t-M}$. We sum over all layers and obtain $O(\lambda C + (L_1 + \cdots L_\lambda) \bar{M} \frac{C}{t-M})$. In the extreme case, with $\lambda = \ell$ and $L_{\mathsf{m}} = 2$, this is $O(\ell C + \bar{M} \frac{C}{t-M})$.

4.3 Privacy Analysis of $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ Protocol

4.3.1 Differential Privacy

The $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ protocol is (ϵ, δ)-differentially private with $\delta = \mathsf{p} \frac{\mathsf{e}^{\epsilon} - 1}{1 - \mathsf{p}}$ for a failure probability $\mathsf{p} = \frac{1}{2} \mathsf{e}^{-\epsilon\left(\mathsf{M} + \frac{1}{2}\right)}$. We prove the statement in Appendix A.3.

We can tune the parameters for our protocol quite nicely. If we want our protocol to achieve $(\epsilon = 1, \delta = 2^{-40})$-DP, then we take $\mathsf{p} \approx 2^{-41}$ which implies $M = 28$. So, for each bucket, we add 28 dummy records on average.

In our protocol, we provide privacy against a malicious server by making both servers to add noise. If one of them is malicious, even though the malicious server can subtract the dummy records he added from the final histogram, learn the histogram with the only dummy records from his counterpart.

Previously, we computed the average number of dummy records per bucket as 28 for $(\epsilon = 1, \delta = 2^{-40})$-DP. This becomes better if all the participants are honest, i.e. it is enough for each honest server to add 14 dummy records on average per bucket because if both are honest, the total 28 is preserved. These parameters change when we have layered Bucketization with λ layers. For example, for $\lambda = 16$, $(2\lambda - 1)\epsilon = 1$, and $(2\lambda - 1)\delta = 2^{-40}$, we take $\mathsf{p} \approx 2^{-40}$ which implies $M = 732$ and $\bar{M} = 732 + 2^{-30}$. We can keep the $(2\lambda - 1)\delta = 2^{-20}$, then we obtain $\bar{M} = 409 + 2^{-11}$.

4.3.2 (Informal) Security Analysis against Malicious Clients

In this section, we give the privacy bound against malicious clients. Note that the malicious behaviour of the client is to modify his reports in a way that it corresponds to a different bucket at the end of the protocol. This holds true because our protocol uses the length preserving secret sharing mechanism. A (malicious) client submits two shares of a report to two corresponding servers. Regardless of the authenticity of shares, the shares will belong to one bucket that an honest client could have submitted.

Informally, the $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ protocol protects against small subset of malicious clients. Since the shares of the attributes preserve the length of the original key size, a small subset of client can only share a wrong key to be counted in another bucket. Since the aggregated results are already noisy, removing the record from the original bucket and increasing the count on another bucket only gives the affect of noise as long as only small set of clients are allowed to do that.

Formally, we prove the following result in Appendix A.4. Let N be the number of malicious clients. The L_1 distance between true histogram and the incorrect histogram output from $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ protocol is bounded by $2\mathsf{N}$.

4.3.3 (Informal) Privacy Analysis with Semi-Honest Servers

The cryptographic protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ is built upon a generic honest-majority three-party computation protocol and a specific three-party oblivious permutation protocol. Overall, it is under honest-majority assumption, which means that the system does not tolerate any collusion. As long as there is no collusion between any pairs of servers, true counts and aggregate keys are hidden from each server as well as the reporting origin. We formally prove the privacy of random shuffling in Appendix A.2.

In the LAYERED protocol, the semi-honest server additionally learns the number of records in the dummy bucket before discarding it. This bucket includes dummy records from upper layers and newly added dummy records. We take it into account in the analysis of the LAYERED protocol.

4.3.4 (Informal) Privacy Analysis Against a Malicious Server

Privacy against malicious server is formally analyzed in Appendix A.5 under a leakage function. This leakage function allows a malicious server to choose an offset vector Δ and to learn the noisy histograms of $D + \Delta$ instead of D. The leakage function from the execution of protocol with semi-honest servers use $\Delta = 0$. Thus, the malicious server model is only introducing the ability to select a nonzero Δ in the leakage function. Thus, we provide the same guarantee as in [6]: allows only additive attacks where a malicious server chooses an offset vector Δ and computation is made on dataset $D + \Delta$ instead of Δ.

We analyse the implication for differential privacy for LAYERED protocol in Appendix A.6.

5 Existing Proposals

Even though there are many different proposals to solve secure aggregate problems, in this section, we focus on two specific proposals: Prio [10] and Distributed Point Functions (DPF) [6].

5.1 Prio

Prio [10] is the first existing protocol to solve privacy preserving aggregate systems which is robust against malicious clients. It does not rely on any general purpose MPC. The protocol can be used for many different aggregates such as histograms, sum, average, heavy-hitter, and others with different techniques and, as a result, with different costs.

Prio uses two-party computations in order to compute the aggregates. Each client secret shares (defined in $\mathbb{Z}_p$ for a prime p) their data to the servers. In order to provide robustness against malicious clients, Prio integrates a special range proof called SNIP and characterized by a Valid predicate. Each client gives each server a proof that the shared data satisfies this predicate. A data point x (shared by a client) is supposed to satisfy the Valid predicate in order to prove the validity of the data point. The predicate is defined by an arithmetic circuit with N multiplications. Even though constructing such proofs are efficient enough, the size of the proofs are $O(N)$ elements in $\mathbb{Z}_p$. This implies a very expensive communication complexity from clients to servers. When the servers receive the proofs, they run the Valid predicate which only requires 1 MPC multiplication per client no matter how large N is.

Prio encodes data x before sharing and this encoding depends on which aggregate function to compute and what type of proof is required. For example, to prove that x is made of ℓ bits, the client first encodes x as $\mathsf{Encode}(x) = (x, \beta_0, \ldots, \beta_{\ell-1})$ β_i represents bits. Then, it generates a proof that $x = \sum_i \beta_i 2^i$ and that every bit β_i is a root of a polynomial $P(z) = z^2 - z$. Thus, what is shared and proved is $\mathsf{Encode}(x)$.

If Prio is used to compute the histograms (or frequency counts as the paper names it), then the encoding becomes a lot larger. The encoding is defined as $\mathsf{Encode}(x) = (\beta_0, \ldots, \beta_{B-1})$ where B is the number of the buckets ($B = 2^\ell$ for full histograms) and $\beta_x = 1$ while $\beta_i = 0$ for $i \in \{0, \ldots, B-1\} \setminus \{x\}$. Valid predicate requires all β_i to be 0 or 1 as well as $\sum_i \beta_i = 1$. As it can be observed, such a method is inefficient for histogram computations. Therefore, we also omit its performance analysis in our comparisons.

Finally, Prio, as it is proposed, does not provide any differential privacy guarantees. However, as shown in some use-cases, it may be possible to add such guarantee under certain conditions [2]. For now, we are not aware of any effort put in that direction. Instead, another proposal to solve specifically the heavy-hitter problem with differential privacy guarantees is proposed. This new proposal specifically aims to reduce the client-side communication complexity of Prio for heavy-hitter problem, as well as introducing additional differential privacy guarantees. We will explain this new primitive next.

5.2 Distributed Point Functions

Recently, Google proposed an Attribute Reporting API with Aggregate Reports scheme, which strongly aligns with this problem [1]. A potential solution mentioned in their proposal relies on Distributed Point Functions (DPF): a two-party secure computation protocol [6, 15]. More precisely, DPF consists of two protocols: DPF.Gen and DPF.Eval. We pause here to explain the basic idea of DPF. Theoretically, the keys can be represented with a large vector of size of the key space. For an ℓ-bit key k, the key can be represented as a one-hot encoded vector of size 2^ℓ, with the k-th position set to 1 and other positions to 0. Then, this vector can be secret shared and sent to two servers to compute the aggregates. However, this naive approach requires too much communication. The beautiful idea DPF introduces is to generate the secret shares of this vector in a compact form and let the servers expand the keys to the full vectors by executing a series of cryptographic operations. The structure of this expansion is a tree structure, *i.e.*, the expansion happens level by level. Essentially, DPF takes these vectors and treats them as functions, which are equivalent when the representation is a point function.

At the beginning of the data collection clients generate their secret shared reports by DPF.Gen. Then, two servers jointly execute DPF.Eval to generate noisy aggregates. Data users (*e.g.*, advertisers) make queries to two servers and receive differentially private results. The aggregate queries DPF allows are histogram and sum on reported keys and values.

The most recent DPF construction is introduced as a solution to the *private heavy hitters problem* [6]. Particularly, Boneh *et al.* [6] describes three main protocols in their paper.

The first protocol is to build a private subset histogram from collected reports for a given set of keys. The set of keys may or may not be known to the servers. It requires $O(CB)$ $\mathsf{DPF.Eval}$ calls, where C is the number of reports and B is the set of keys to build the histogram on (without differential privacy).[6]

The second protocol is to find the most popular keys, which appears with a threshold t (without differential privacy); this is called the t-heavy hitters problem. Boneh *et al.* defines a new DPF called *incremental DPF* (iDPF) to solve this problem more efficiently than with standard DPF. The complexity of the proposed protocol is $O(\ell C^2/t)$ $\mathsf{DPF.Eval}$ calls, where ℓ is the (fixed) size of the keys collected from clients. The third protocol is simply to use the t-heavy hitters protocol with threshold $t = 1$. Then, the complexity becomes $O(\ell C^2)$ $\mathsf{DPF.Eval}$ calls.

These protocols can be made differentially private by applying the noise addition process at certain steps. The differential privacy parameters proposed in [6] use

$$\epsilon' = \epsilon\sqrt{2q \ln \frac{1}{\delta'}} + \epsilon q e^{\epsilon - 1} ,$$

with $q = \ell C/t$. They provide example parameters for an (ϵ', δ')-DP protocol, with $\ell = 256$, $\epsilon = 0.001$, $t = C/100$, and $\delta' = 2^{-40}$, resulting in $\epsilon' = 1.22$. However, the impact on the complexity and the accuracy is not analysed.

6 Performance Evaluation

We want to take advantage of performance benefit of layering for large attribute sizes) for histogram aggregation and report the performance. When we run Layered $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ on a long attribute size, we layered it with $\lambda \geq 2$ by dividing the attribute into equal sizes. The three helper servers $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ run on Azure Standard D8s v4 virtual machines with 8 virtual CPUs and 32 GiB RAM. $\mathcal{S}_1, \mathcal{S}_3$ are located in the West-US-2 region (Washington) and the round-trip latency between them is throttled to 60 ms. $\mathcal{S}_2$ is located in the East-US-2 region (Virginia). The network latency between the east and west regions is around 60 ms. Note that most of the network communication happens between $(\mathcal{S}_1, \mathcal{S}_2)$, and $(\mathcal{S}_2, \mathcal{S}_3)$.

Our implementation utilizes only a single thread and the code is far from fully optimized. For some experiments, we run all servers on the same machine and simulated the network latency with 60 ms; we clarify when we do this. We ignore the cost of clients in the experiments and only focus on the overhead of helper servers. Finally, all experiments are implemented with (ϵ, δ)-DP, with parameters $\epsilon = 1$ and $M = 28$. We obtain $\mathsf{p} = 2^{-42}$, $\delta = 2^{-41}$, $\bar{M} = M + 2^{-37}$ for one layer.

[6]Note that the complexity of $\mathsf{DPF.Eval}$ is exponential in the size of the keys.

6.1 Constructing a Full Histogram.

We benchmark the performance of our protocols for generating differentially private **full histograms** from **short keys**. $C = 10\,000\,000$ reports of clients are synthesized and generated from a uniform distribution. We choose the reported key lengths from the list $\ell \in \{16, 18, 20, 22, 24\}$. For each execution, a full histogram consisting of 2^ℓ buckets is generated by the helper servers. When testing our protocols, the one-time bucketization protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ is executed for $\ell \leq 22$. For $\ell = 24$, we instantiate Algorithm 1 to run a two-layer bucketization on $\ell_1 = 12$ bits and $\ell_2 = 12$ bits, where $\ell = \ell_1 + \ell_2$. This is because of memory constraints caused by large number of buckets and dummy reports from DP noise.

We compare the performance of our protocol with the **regular DPF**-based protocol.[7] In this scheme, the input to the helper servers are DPF keys generated from a domain of size 2^ℓ. Two helper servers evaluate the shared keys at all points of the DPF domain. Finally, they output a vector of size 2^ℓ representing the histogram.

To generate full histograms, the regular DPF-based protocol is used with the key evaluation phase. The performance measures the computational efficiency of helpers running in one server. We run the benchmarks with provided code, which does not take the number of reports as a parameter, instead, it benchmarks the regular DPF for one report. We first benchmark the average time used to evaluate a DPF tree at every point and interpolate it into the time usage for $C = 10\,000\,000$ reports by simply multiplying it. The results are shown in Table 3. Both DPF and Bucketization implementation will benefit from optimization. However, in general, our protocol is much more efficient than the regular DPF-based protocol for generating **full histograms for short keys**.

6.2 Constructing a Subset-Histogram via Pruning

In the real world practice, generating a histogram of the whole domain is not always meaningful. The domain size can be large and the reports are usually not uniformly distributed. As in the setting of [6], the reported keys come from a certain distribution and are encoded sparsely in a large domain. The data collector is only interested in popular reports, but tend to ignore rarely appeared outliers. In histogram aggregation, a large number of buckets may be empty or only contain a small number of reports, when the aggregation keys have a large domain size or the input reports have a highly concentrated distribution. We demonstrate the performance of our protocol LAYERED $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ with following experiments.

We synthesize a dataset of $C = 10\,000\,000$ input 32-bit keys from a domain size of 2^{16}, so the attributes are sparsely encoded in the keys. The keys follow a Zipf distribution, with Zipf parameter 1.03. We set the threshold to be $t = \frac{C}{10\,000} + \bar{M}$, where $\bar{M}$ is the average number of noise in each output bucket. We

[7]We used the implementation available at https://github.com/google/distributed_point_functions (commit 934011c).

ℓ	16	18	20	22	24
Ours	9.2 s	11.2 s	25.3 s	75 s	422 s
Regular DPF	1.15 days	4.75 days	19 days	76 days	305 days

Table 3: Performance of constructing full-domain histograms. The inputs from 10 000 000 clients are uniformly sampled with ℓ-bit keys. The output is a full histogram containing 2^ℓ buckets. The numbers in the table represent the running time of experiments. For our protocols, we set two locally separated servers with 110 ms network latency. We instantiate one-time bucketization protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ when $\ell \leq 22$ and two-layer bucketization Algorithm 1 for $\ell = 24$ with a split of 12 bits for each layer. Our protocol implements the DP with parameters set to $\epsilon = 1, \delta = 2^{-40}, \bar{M} = 28$. Regular DPF results are run on one server for one report and multiplied by C as the code provided in the repository allows us. Regular DPF results do not implement DP. We omit Prio due to the expensive client-to-server communication complexity.

run Algorithm 1 with a two-layer bucketization on higher and lower 16-bit keys, respectively. It takes around 139 seconds to generate the histogram with 945 output buckets.

We also compare the performance of our bucketization protocol with the subset-histogram appeared in the end-to-end performance evaluation of [6]. The two-party computation protocol in [6] is based on the incremental DPF (iDPF). For the performance evaluation, $C = 400\,000$ input keys are generated from a Zipf distribution with Zipf parameter 1.03 and support 10 000. The bit-length of input keys is 256. The threshold is $t = C/1000$. The experiment of [6] is done between 2 servers of 32 virtual CPUs and is equipped with a network of 61.9 ms round-trip latency. It takes around 53 minutes to generate a subset-histogram. For our protocol, 3 helper servers run the protocol LAYERED $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ (described in Algorithm 1) with $\lambda = 16$ and bucketize the keys into 16-bit sub-keys. It takes 122 seconds to generate the subset-histogram from the above input.

6.3 Micro-Benchmarks

The Bucketization protocol is efficient and flexible. The clients encode a number of attributes into a report and secretly share the whole report to helper servers. According to the query from the reporting origin, helper servers perform histogram aggregation on any set of attributes. For the DPF-based protocols, the queries are known ahead of time and the clients need to generate evaluation materials for certain queries.

We micro-benchmark the protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ with variable key length to demonstrate the performance of three main components: dummy records adding, random shuffling, and noisy label reveal. For each execution, $C = 10\,000\,000$ reports are sampled from a Zipf distribution with parameter 1.03. The pruning

Key size ℓ' (bits)	32	64	256	512
Add dummy records	26 ms	47 ms	176 ms	355 ms
Random shuffling	3.91 s	5.86 s	20.1 s	38.4 s
Noisy labels reveal	2.41 s	2.86 s	6.73 s	10.92 s

Table 4: The numbers are total running time of the one-time bucketization protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ with all servers running on the same machine with 60 ms latency. The bucketization is executed only on a 16-bit attribute.

threshold is set to $t = 10 + \bar{M}$. The length of the reported keys is from a list $\ell \in \{32, 64, 256, 512\}$ and the servers generate differentially private histograms from a domain size of 2^{16} on any 16 bits encoded as attributes in the keys. The results are shown in Table 4.

As an interactive MPC protocol, Bucketization is susceptible to the restrictions on network communication, especially the round-trip latency (60 ms). The overhead mainly lies in the three-party random shuffling protocol, which has a communication complexity of $\mathcal{O}((C + B\bar{M})\ell)$. Revealing the noisy labels has a communication complexity of $\mathcal{O}(C + B\bar{M})$. Overall, the drawback of our protocol is the communication complexity between servers.

7 Conclusions

We have presented an efficient three-party protocol, Bucketization, for computing privacy-preserving histogram queries. The basic protocol Figure 10 can be used iteratively to compute histograms for large keys with sparse distributions, as described in Algorithm 1.

We believe our approach can present a viable method for enabling web advertisers to obtain valuable information about their ad campaigns, while still preserving the privacy of individual users with state-of-the-art cryptographic techniques and differential privacy. Our protocol is simpler and works with linear communication/computational complexity in the number of clients. The performance analysis indicates that it can be very efficient for various different test vectors.

References

[1] Attribution Reporting API with Aggregate Reports. https://github.com/WICG/conversion-measurement-api/blob/main/AGGREGATE.md.

[2] Exposure Notification Privacy-preserving Analytics. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

[3] Library for Prio. https://github.com/abetterinternet/libprio-rs.

[4] Privacy Preserving Measurement Protocol. https://github.com/abetterinternet/ppm-specification.

[5] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares. https://eprint.iacr.org/2021/576.pdf, 2021.

[6] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight Techniques for Private Heavy Hitters, 2021.

[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing. In *Advances in Cryptology – EUROCRYPT 2015*. Springer International Publishing, 2015.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing: Improvements and Extensions. In *CCS 2016*, 2016.

[9] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-Shared Shuffle. In *Advances in Cryptology – ASIACRYPT 2020*, pages 342–372. Springer International Publishing, 2020.

[10] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 259–282. USENIX Association, 2017.

[11] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *3rd Theory of Cryptography Conference—TCC 2006*, volume 3876 of *LNCS*, pages 265–284. Springer, 2006.

[12] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407, August 2014.

[13] Tim Geoghegan, Christopher Patton, Eric Rescorla, and Christopher Wood. Privacy preserving measurement. https://datatracker.ietf.org/doc/draft-gpew-priv-ppm/.

[14] Niv Gilboa and Yuval Ishai. Distributed Point Functions and Their Applications. In *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658. Springer International Publishing, 2014.

[15] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Advances in Cryptology—Eurocrypt 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, 2014.

[16] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM Press, 1987.

[17] Sahar Mazloom and S. Dov Gordon. Secure computation with differentially private access patterns. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 490–507. Association for Computing Machinery, 2018.

[18] Andrés Muñoz Medina, Umar Syed, Sergei Vassilvitskii, and Ellen Vitercik. Private Optimization Without Constraint Violations. In *AISTATS*, 2021.

[19] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and PSI for secret shared data. In *ACM Conf. on Computer and Communications Security (CCS) 2020*, pages 1271–1287. ACM Press, 2020.

[20] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. FOCS'86, page 162–167. IEEE Computer Society, 1986.

A Appendix

A.1 Sampling Noise Securely

To sample the noise following the description given in Section 3.1, we have two options:

1. Naive rejection sampling:

```
1: Sample $n \leftarrow \mathsf{Lap}\left(X, \frac{1}{\epsilon}\right)$
2: while $n < -M - \frac{1}{2}$ do.
3:     Sample $n \leftarrow \mathsf{Lap}\left(X, \frac{1}{\epsilon}\right)$
4: end while
5: Output $n$
```

2. Directly inverting the CDF of truncated Laplace distribution:

```
1: Sample $u$ uniformly between 0 and 1.
2: Set $z = 2(1 - \mathsf{p})\left(u - \frac{\frac{1}{2} - \mathsf{p}}{1 - \mathsf{p}}\right)$.
3: Set $a = -\frac{\ln(1 - |z|)}{\epsilon}$.
4: Set $x = a\,\mathsf{sign}(z)$.
5: Output $x$.
```

This follows from

$$\begin{aligned}
\mathrm{CDF}(x) &= \int\limits_{-M}^{x} \frac{\mathsf{Lap}\left(t, \frac{1}{\epsilon}\right)}{1-\mathsf{p}}\, dt \\
&= \frac{\frac{1}{2}-\mathsf{p}}{1-\mathsf{p}} + \int\limits_{0}^{x} \frac{\mathsf{Lap}\left(t, \frac{1}{\epsilon}\right)}{1-\mathsf{p}}\, dt \\
&= \frac{\frac{1}{2}-\mathsf{p}}{1-\mathsf{p}} + \frac{\epsilon}{2(1-\mathsf{p})} \int\limits_{0}^{x} e^{-\epsilon.|t|}\, dt \\
&= \frac{\frac{1}{2}-\mathsf{p}}{1-\mathsf{p}} + \frac{1}{2(1-p)}(1-e^{-\epsilon|x|})\mathsf{sign}(x)
\end{aligned}$$

When the second method is followed and carefully implemented, assuming that inverting the CDF can be done in constant time, the method is **not** time-invariant. However, the number of rejections is independent from input u and output x. Thus, the number of rejection does not leak any information to the adversary.

A.2 Privacy of Oblivious Random Shuffling In Honest-but-Curious Model

We allow that a malicious participant U colludes with the Reporting Origin to learn the final histogram. In the worst case, we assume that U learns A' and B' produced by the shuffling protocol based on which the final histogram is computed.

Theorem 1. *Assume that all the participants follow the protocol and are non-colluding (honest but curious). For each participant of the protocol* Π_{RandShuf} *described in Figure 7, there exists an efficient simulator* Sim_U *such that the view of* U *in the protocol can be simulated from the final output* (A', B').

Proof. The view of $U = \mathcal{S}_1$ is that the received shares from each client D^1, the value π_{12}, π_{13}, R_{12} and R_{13}, the value A, and the value B':

$$(D^1, \pi_{12}, \pi_{13}, R_{12}, R_{13}, A, B')$$

$\mathcal{S}_1$ computes A' from (A, π_{13}, R_{13}). Then, its view is equivalent to

$$(D^1, \pi_{12}, \pi_{13}, R_{12}, R_{13}, A, A' + B')$$

where $A' + B' = \pi_{13}(\pi_{23}(\pi_{12}(D)))$. Since π_{13} is known, the view of $\mathcal{S}_1$ is equivalent to

$$(D^1, \pi_{12}, \pi_{13}, R_{12}, R_{13}, A, \pi_{23}(\pi_{12}(D)))$$

The first five terms: $D^1, \pi_{12}, \pi_{13}, R_{12}, R_{13}$ are independently sampled. The last term is a random permutation of D which is independent of the first five terms. The sixth term A is a function of D^2, R_{12}, R_{23} with an independent value R_{23}. Thus, the simulator for this view would independently sample the first six terms and would select an independent permutation of D which can be done from the output of the protocol.

The same procedure applies to $U = \mathcal{S}_2$ and $U = \mathcal{S}_3$ similarly. □

A.3 Differentially Private Protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$

Theorem 2. *The $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ protocol given in Figure 10 is (ϵ, δ)-differentially private with $\delta = \mathsf{p}\frac{\mathsf{e}^{\epsilon}-1}{1-\mathsf{p}}$ for a failure probability $\mathsf{p} = \frac{1}{2}\mathsf{e}^{-\epsilon\left(\mathsf{M}+\frac{1}{2}\right)}$.*

We consider what is learnt by $\mathcal{S}_1$ (or $\mathcal{S}_2$) as other participants will see noisy results. $\mathcal{S}_1$ learns the final noisy histogram minus the noise it added himself, i.e. the true histogram plus the noise of his counterpart. As other participants will see more noise in data, we focus on $\mathcal{S}_1$ and $\mathcal{S}_2$ and prove the theorem as follows:

Proof. Let d and d' be two neighboring databases defined on $\mathbb{N}^{\chi}$. In what follows, we let i be a fixed index such that $|d_i - d'_i| = 1$ and $d_j = d'_j$ for all $j \in \{1, \cdots, L\}$ except $j = i$. Let $f(\cdot)$ be the function with noisy output such that $f := \mathbb{N}^{\chi} \to \mathbb{R}^L$.

For each bucket, protocol Π_{NoiseGen} (given in Figure 5) iterates until the Laplace noise is larger than $-M$. Let P denote the PDF of the noise n as $P(n) = \frac{\mathsf{Lap}(n, \frac{1}{\epsilon})}{1-\mathsf{p}}$ if $n \geq -M$ and $P(n) = 0$ otherwise. Recall that $\mathsf{p} = \frac{1}{2}\mathsf{e}^{-\epsilon\left(\mathsf{M}+\frac{1}{2}\right)}$.

Let $S \in \mathbb{R}^L$ be an arbitrary set. We split S into two disjoint subsets S_{good} and S_{bad}. S_{good} has points $s \in S$ such that $(s_i - d_i) \geq -M - \frac{1}{2}$ **and** $(s_i - d'_i) \geq -M - \frac{1}{2}$. It means that we can obtain s from either d or d' (as far as the bucket i is concerned) because the noise to add is greater than or equal to $-M - \frac{1}{2}$. S_{bad} has points s such that $(s_i - d_i) < -M - \frac{1}{2}$ **or** $(s_i - d'_i) < -M - \frac{1}{2}$. It means that s is impossible to obtain from either d or d' (as far as the bucket i is concerned). Here, i is a specific index defined from d and d' by $|d_i - d'_i| = 1$. We consider noise vectors n defined as $(s - d)$ (or $(s - d')$).

The property for s to be in S_{good} or S_{bad} depends on the single bucket index i. We start computing the probability of $f(d)$ being in S_{good} for which the standard Laplace mechanism proof [12] works as it is:

$$
\begin{aligned}
\Pr(f(d) \in S_{\mathsf{good}}) &= \int_{s \in S_{\mathsf{good}}} \prod_j P(s_j - d_j) && (1)\\
&= \int_{s \in S_{\mathsf{good}}} \frac{\mathsf{Lap}(s_i - d_i)}{1 - \mathsf{p}} \prod_{j \neq i} P(s_j - d_j) && (2)\\
&= \int_{s \in S_{\mathsf{good}}} \frac{\mathsf{Lap}(s_i - d_i' + d_i' - d_i)}{1 - \mathsf{p}} \prod_{j \neq i} P(s_j - d_j') && (3)\\
&\leq e^{\epsilon} \int_{s \in S_{\mathsf{good}}} \frac{\mathsf{Lap}(s_i - d_i')}{1 - \mathsf{p}} \prod_{j \neq i} P(s_j - d_j') && (4)\\
&= e^{\epsilon} \int_{s \in S_{\mathsf{good}}} \prod_j \Pr(s_j - d_j') && (5)\\
&= e^{\epsilon} \Pr(f(d') \in S_{\mathsf{good}}) && (6)\\
&\leq e^{\epsilon} \Pr(f(d') \in S) && (7)
\end{aligned}
$$

where Equation (4) follows from

$$\mathsf{Lap}(s_i - d_i' + (d_i' - d_i)) = \mathsf{Lap}(s_i - d_i') e^{\epsilon(d_i - d_i')} \leq \mathsf{Lap}(s_i - d_i') e^{\epsilon}$$

Next, we compute the probability over S_{bad} which can be bounded as follows. Let the noise $n_i = s_i - d_i$. If $f(d)$ is in S_{bad}, it means that either $n_i < -M$ (which is not possible due to resampling) or $n_i + d_i - d_i' < -M + \frac{1}{2}$. To reach any $s \in S_{\mathsf{bad}}$ from f, we must add a noise between $-M - \frac{1}{2}$ and $-M + \frac{1}{2}$ (this is a necessary but not sufficient condition). Thus,

$$
\begin{aligned}
\Pr(f(d) \in S_{\mathsf{bad}}) &\leq \Pr\left(n_i \in \left[-M - \frac{1}{2}, -M + \frac{1}{2}\right]\right) && (8)\\
&= \int_{-M-\frac{1}{2}}^{-M+\frac{1}{2}} \frac{\mathsf{Lap}(n_i, \frac{1}{\epsilon})}{1 - \mathsf{p}} && (9)\\
&= \mathsf{p} \frac{\mathsf{e}^{\epsilon} - 1}{1 - \mathsf{p}} && (10)\\
&= \delta && (11)
\end{aligned}
$$

Equation (10) follows from $\int_{-M-\frac{1}{2}}^{-M+\frac{1}{2}} \mathsf{Lap}(n_i, \frac{1}{\epsilon}) = \frac{1}{2} e^{-\epsilon\left(M + \frac{1}{2}\right)} (e^{\epsilon} - 1)$ where $\frac{1}{2} e^{-\epsilon\left(M + \frac{1}{2}\right)} = \mathsf{p}$. Hence, we show that $\Pr(f(d) \in S_{\mathsf{bad}})$ is lower than the probability to sample a "bad" noise which is δ. $\Pr(f(d) \in S_{\mathsf{bad}}) \leq \delta$ for $\delta = \mathsf{p} \frac{\mathsf{e}^{\epsilon} - 1}{1 - \mathsf{p}}$.

Finally,

$$
\begin{aligned}
\Pr(f(d) \in S) &= \Pr(f(d) \in S_{\mathsf{good}}) + \Pr(f(d) \in S_{\mathsf{bad}}) && (12)\\
&\leq e^{\epsilon} \Pr(f(d') \in S) + \delta && (13)
\end{aligned}
$$

We conclude that the protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ is (ϵ, δ) differentially private for $\delta = \mathsf{p}\frac{\mathsf{e}^{\epsilon}-1}{1-\mathsf{p}}$.

□

A.4 Security Analysis for Malicious Clients

Theorem 3. *Let* N *be the number of malicious clients. The* L_1 *distance between the true histogram and the incorrect histogram output from* $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ *protocol described in Figure 10 is bounded by* $2\mathsf{N}$.

Proof. We start with one malicious client. Let $\mathsf{rec}_{\mathsf{auth}}$ be the true record of a malicious client. Let a (resp. b) be the number of records in the correct (resp. incorrect) bucket that $\mathsf{rec}_{\mathsf{auth}}$ belongs to. The consequence of the malicious behaviour is that true bucket will have $\mathsf{a} - 1$ records while incorrect bucket will have $\mathsf{b} + 1$. The L_1 distance between true histogram and the is defined as the sum (over all buckets) of the absolute values of the difference between two counts in both histograms. In the case of one malicious client, the L_1 distance is bounded by 2. By triangle inequality, the L_1 distance induced by N clients is bounded by $2\mathsf{N}$. □

A.5 Privacy Against a Malicious Server

We consider the following Leak game played by an adversary $\mathcal{A}$ with a dataset D as input.

```
Leak_A(D)
1 : Set $C$ to the size of $D$.
2 : Generate $n^2$ dummy records $D_{\mathsf{dum}}$.
3 : Run $\mathcal{A}(\mathsf{coins}, C, n^2) \rightarrow \Delta$.
4 : Compute the histogram $\mathsf{hist}$ of $D||D_{\mathsf{dum}} + \Delta$.
5 : Output $(\mathsf{coins}, C, n^2, \mathsf{hist})$.
```

We let $\mathsf{View}_{\mathcal{S}}(D)$ be the view of $\mathcal{S}_1$ when running the protocol with input dataset D.

Theorem 4. *For a malicious server* $\mathcal{S}_1$*, there exists an adversary* $\mathcal{A}$ *and a simulator* Sim *such that for any dataset* D $\mathsf{Sim}(\mathsf{Leak}_{\mathcal{A}}(D)) \sim \mathsf{View}_{\mathcal{S}_1}(D)$.

Thus, a malicious $\mathcal{S}_1$ does not learn more than what it would learn playing the Leak game.

Proof. The protocol from the viewpoint of $\mathcal{S}_1$ is defined as follows: $\mathcal{S}_1$ receives dataset share D^1 and the total number n^2 of dummy records $\mathcal{S}_2$ wants to add. $\mathcal{S}_1$ selects and outputs $n^1, \pi_{12}, \pi_{13}, R_{12}, R_{13}$. $\mathcal{S}_1$ gets a vector A defined by $A = \pi_{23}(\pi_{12}((D - D^1)||D^2_{\mathsf{dum}}) + R_{12}) + R_{23}$ with π_{23} and R_{23} uniform and D_{dum} the dummy shares by $\mathcal{S}_2$. $\mathcal{S}_1$ selects vectors A' and B. And, $\mathcal{S}_1$ learns B' defined by $B' = \pi_{13}(\pi_{23}(B) - R_{23}) + R_{13}$

Until B' is obtained, the partial view of $\mathcal{S}_1$ is $(\mathsf{coins}, D^1, n^2, A)$ which is perfectly simulatable as the tuple elements are independent and follow a known distribution. However, the final B' is dependent on the rest of the tuple which leaks.

Let $\mathsf{Out} = A + \pi_{13}^{-1}(B' - R_{13})$. There is a 1-to-1 correspondence between the final view $(\mathsf{coins}, D^1, n^2, A, B')$ and $(\mathsf{coins}, D^1, n^2, A, \mathsf{Out})$. Thus, instead of B', we consider the equivalent Out.

We have $\mathsf{Out} = \pi_{23}(\pi_{12}((D - D^1)||D^2_{\mathsf{dum}}) + B + R_{12})$. Let $\Delta' = \pi_{12}^{-1}(B + R_{12}) - D^1||0^{n^1+n^2}$ so that $\mathsf{Out} = \pi_{23}(\pi_{12}(D||D^2_{\mathsf{dum}} + \Delta'))$

We assume that D^2_{dum} is composed of the n^2 dummy records created by $\mathcal{S}_2$ and by n^1 slots of 0 shares (to be added to the dummy records created by $\mathcal{S}_1$). We accordingly split Δ' as $\Delta' = \Delta||\Delta_0$ with Δ_0 of size n^1 (to be added to the 0 slots). Hence, $\mathsf{Out} = \pi_{23}(\pi_{12}(((D||D_{\mathsf{dum}}) + \Delta)||\Delta_0))$ which is a random permutation of $(D||D_{\mathsf{dum}}) + \Delta)||\Delta_0$.

We construct $\mathcal{A}$ as follows:

```
A(coins, C, n^2)
1 : Generate $D^1$ uniformly of size $C$.
2 : Simulate $\mathcal{S}_1$ upon receiving $D^1$ and $n^2$.
3 : Simulate $\mathcal{S}_1$ selecting $n^1, \pi_{12}, \pi_{13}, R_{12}, R_{13}$.
4 : Generate $A$ uniformly of size $(C + n^1 + n^2)$.
5 : Simulate $\mathcal{S}_1$ receiving $A$.
6 : Simulate $\mathcal{S}_1$ selecting $A'$ and $B$.
7 : Compute $\Delta' = \pi_{12}^{-1}(B + R_{12}) - D^1||0^{n^1+n^2}$.
8 : Split $\Delta' = \Delta||\Delta_0$.
9 : Output $\Delta$.
```

All random processes are done using the random sequence of coins at the input of $\mathcal{A}$ so that they could be redone deterministically by Sim when needed.

In $\mathsf{Leak}_{\mathcal{A}}(D)$, we could create an arbitrary vector with the histogram hist, append it to Δ_0, apply a random permutation π. We would obtain a view with same distribution as the view of $\mathcal{S}_1$ in the protocol. We define Sim as follows:

```
Sim(coins, C, n^2, hist)
1 : Redo the computations of $\mathcal{A}$ to get the same variables.
2 : Create a vector $V$ with histogram $\mathsf{hist}$.
3 : Set $V' = V||\Delta_0$.
4 : Pick a random permutation $\pi$.
5 : Set $\mathsf{Out} = \pi(V')$.
6 : Compute $B' = \pi_{13}(\mathsf{Out} + A) + R_{13}$.
7 : Output $(\mathsf{coins}, D^1, n^2, A, B')$.
```

This produces a view with same distribution as $\mathsf{View}_{\mathcal{S}_1}(D)$ in the protocol for $\mathcal{S}_1$. □

A.5.1 Discussions

The proof for malicious server $\mathcal{S}_2$ is simpler. The view of the protocol for $\mathcal{S}_2$ is: it receives dataset shares D^2 and the number n^1 of dummy records that $\mathcal{S}_1$ wants to add; it selects and outputs $(n^2, \pi_{12}, \pi_{23}, R_{12}, R_{23}$; it selects a vector A and learns vectors A' and B' where $A' = \pi_{13}(A) - R_{13}$ and

$$B' = \pi_{13}(\pi_{23}(\pi_{12}(D - D^2) - R_{12}) - R_{23}) + R_{13}$$

with π_{13} and R_{13} random. Then, the view of $\mathcal{S}_2$ is

$$\mathsf{View}_{\mathcal{S}_2}(D) = (\mathsf{coins}, D^2, n^1, A', B')\,.$$

The (A', B') pair is equivalent to $(\mathsf{Unif}, A' + B')$ with $A' + B'$ being a random permutation of

$$(A + \pi_{23}(\pi_{12}(D - D^2) - R_{12}) - R_{23})\,.$$

With appropriate change of variables, we can construct $\mathcal{A}$ and Sim for $\mathcal{S}_2$ in a similar way.

The proof is even simpler for $\mathcal{S}_3$ as the protocol leaks less to him. $\mathcal{S}_3$ receives a total length $N = C + n^1 + n^2)$; it selects and outputs $\pi_{13}, R_{13}, \pi_{23}, R_{23}$. Then, it receives B and A' such that $B = \pi_{12}(D^1) - R_{12}$ and

$$A' = \pi_{13}(\pi_{23}(\pi_{12}(D^2) + R_{12}) + R_{23}) - R_{13}\,.$$

The view of $\mathcal{S}_3$ is

$$\mathsf{View}_{\mathcal{S}_3}(D) = (\mathsf{coins}, N, B, A')\,.$$

In A', $\mathcal{S}_3$ can peel off $R_{13}, \pi_{13}, R_{23}, \pi_{23}$ and get $\pi_{12}(D^2) + R_{12}$ which added to B gives a random permutation of $D||D_{\mathsf{dum}}$.

A.5.2 Collusion Between Reporting Origin and a Helper Server

In the security analysis, we assume that A' and B' leak. The view of Reporting Origin is a deterministic function of A' and B'. Therefore, the collusion between a malicious server and the Reporting Origin is already implicitly covered.

A.6 Differential Privacy of Layered Protocol with Honest-but-Curious and Malicious Servers

We first consider honest-but-curious server. Given two neighbouring datasets d and d' and a target s, the noise to select is either $s - d$ or $s - d'$. For every attribute of the unique record which was added or withdrawn between d and d', there is a corresponding noise which is changed by 1. Moreover, the change in this noise induces a change in the following dummy bucket noise, as well. Thus, the total number of affected sample noises is $2\lambda - 1$.

In the proof of Theorem 2, what changes is that we have one e^ϵ appearing for each of the affected noise. The consequence is that ϵ is multiplied by $2\lambda - 1$.

We also define S_{good} to hold for every affected noise. The consequence is that δ multiplied by $2\lambda - 1$. Hence, the Layered protocol is $((2\lambda - 1)\epsilon, (2\lambda - 1)\delta)$-DP.

With a malicious server, what changes is that the server can decide an offset Δ at every execution of the Bucketization protocol. This does not affect the proof.