Aggregate Measurement via Oblivious Shuffling

Erik Anderson
Microsoft
erikan@microsoft.com

Melissa Chase
Microsoft
melissac@microsoft.com

Wei Dai
Microsoft
wei.dai@microsoft.com

F. Betül Durak
Microsoft
betuldurak@microsoft.com

Kim Laine
Microsoft
kim.laine@microsoft.com

Siddharth Sharma
Microsoft
siddhash@microsoft.com

Chenkai Weng
Northwestern University
ckweng@u.northwestern.edu

ABSTRACT

We introduce a new secure aggregation method for computing aggregate statistics over secret shared data in a client-server setting. Our protocol is particularly suitable for ad conversion measurement computations, where online advertisers and ad networks want to measure the performance of ad campaigns without requiring privacy-invasive techniques, such as third-party cookies. Our protocol has linear complexity in the number of data points and guarantees differentially private outputs. We formally analyze the security and privacy of our protocol and present a performance evaluation with comparison to other approaches proposed for a similar task.

1 INTRODUCTION

Privacy-Preserving Aggregation. Many software applications and services today collect statistical data about their use, for example, for optimizing user-experiences and performance. Collecting more descriptive statistics may yield better results, but also raises more privacy and regulatory compliance concerns. This privacy-utility trade-off has been an active area of research for a long time, including for smart meters [15, 27], private networks [10, 19, 26], and other measurements [21].

In this work, we focus on the problem of privacy-preserving aggregation, where a large number of *clients* each submit a data *report*. The goal is to compute aggregate statistics over the reports in a way that does not compromise privacy and deliver the results to a *Reporting Origin*. More precisely, we follow the model proposed in Prio [14], where each client secret shares it's report and distributes the shares to a small number of servers. The servers then work together to compute aggregates over the reports in a way that guarantees differential privacy for each client report.

While similar systems have been used for other applications[1], we focus on web privacy related applications and present a solution which is both more efficient and more flexible than other proposals.

Web Privacy and Ad Conversion Measurement. In today's web, various entities perform cross-site tracking of people's activity for multiple reasons, one of which is to derive insights about online marketing campaigns. This cross-site tracking is often accomplished via third-party cookies, where website A includes content getting and setting a cookie tied to website F. When website F is also loaded across multiple other websites, F is able to track people's activities across these sites, raising significant privacy concerns [5].

Due to these concerns, browser vendors have started to greatly restrict when third-party cookies can be set. For instance, Apple Safari blocks all third-party cookies by default and Google Chrome's Privacy Sandbox effort proposes to phase out third-party cookies as they exist today. All major browser vendors are taking steps 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 valuable services free of charge to everyone.

In many measurement scenarios the most valuable information concerns trends rather than individual events, for example, how often an ad placement on an publisher's site results in the visitor completing a purchase of the advertised item or service. These insights are currently derived by collecting information about individuals (web browser clients) and then computing preferred aggregates, with no guarantee that someone's data would not be used for other, potentially more privacy-invasive, purposes. By instead switching to systems that provide only aggregate results with provable guarantees of user-level privacy, most measurement scenarios can still be enabled, while mitigating risks and concerns.

In this work, we propose a solution to this problem in the form of a privacy-preserving aggregate protocol based on secure multi-party computation and differential privacy.

1.1 Our Model

1.1.1 Parties and Trust assumptions. We consider a system as in Figure 1, which follows along the same basic structure as the Verifiable Distributed Aggregation Functions currently being formalized by the IETF [4]. The system includes the following parties:

(1) **Clients**, that each submit one data report. In the web setting, the reports would typically be submitted by web browser clients. As there is no way to limit how client software behaves, we need to ensure security against malicious clients.
(2) A **Reporting Origin**, that collects reports from the clients, sends them to the helper servers (see below), and chooses which aggregates to compute. In our motivating application, this could be an ad network wanting to measure the effectiveness of an ad campaign. Even a malicious Reporting

[1] See for example Mozilla's use of Prio for telemetry [20] and Google and Apple's use of Prio for contact tracing with analytics [2].

Origin should not learn anything about individual clients' reports.

(3) **Helper servers (aggregation servers)**, that work together to perform the aggregation. We want to provide privacy even when malicious helper servers collude with a Reporting Origin, as long as a majority of the helper servers is honest. One might also consider the stronger requirement, where correctness of the final result in the presence of adversarial helper servers is guaranteed. However, we leave this for future work.

1.1.2 Data reports. We assume that each report collected from a client is a bit string, encoding one or more predefined *attributes*, such as client information (age, gender, geographic region), device information (user-agent string, OS), or ad category. Each attribute is allocated a certain number of bits from the report. The bits reserved for an attribute thus encode the categorical *attribute values*.

We also consider numerical attributes over which one might want to compute sums, for example the dollar amount of ad conversion purchases. In a malicious client setting, this requires a non-trivial protocol. We propose a new method which is significantly better than Prio [14].

1.1.3 Flexible Aggregation. In non-private ad conversion measurement systems, each report contains many attributes about a client and analysts are able to view aggregate statistics about the ad campaign's success, subdivided by a variety of different attributes [31]. For example, an analyst might first view the number of conversions and value of conversions segmented by ad campaign, then filter for a particular campaign, and group the results by specific attributes, such as age, geographic region, or a combination of attributes.

Filtering makes it possible to explore a sparse space. For example, the analyst could first partition by geographic region, then consider only those regions for which there has been conversion activity, then further partition by age, etc. This allows the analyst to identify combinations of attributes with a non-trivial number of conversions, without having to explore exponentially many possibilities.

In a privacy-preserving solution some restriction on the queries is necessary in order to guarantee differential privacy, but we want to maintain enough flexibility to such a dashboard, where an analyst can decide on-the-fly how they want to view the data. However, existing proposals cannot do this. The two main ideas under consideration for ad conversion measurement are based on *Distributed Point Functions* (DPFs) [8, 9, 22], *incremental DPFs* (iDPFs) [7], and Prio [1, 14]. iDPF is very similar to DPF; it adds support for longer reports and differential privacy.

In the (i)DPF approach, attributes are encoded in a bit string, as they are in our approach. However, the only kinds of aggregation functions supported are to count reports with a given prefix, so if the attribute bits encode to first a geographic location, then an age bracket, and then a device type, one could obtain counts for "east coast", "east coast, 65+", and "east coast, 65+, mobile", but one could not obtain counts for "65+, mobile", except by separately querying with each possible location. As the number of attributes included in reports increases, this becomes increasingly restrictive.

In Prio, data is encoded as vectors of values and the only aggregation functions that can be computed are those that can be expressed as sums of these values. For example, if we wanted to be able to count the number of reports corresponding to a combination of location, age, and device type, the client would have to encode their report into a vector with an entry for every possible (location, age, device type) combination. Clearly, as the number of attributes grows, this becomes infeasible.

1.2 Our Results

We propose an efficient (linear time and communication complexity in the number of reports) privacy-preserving aggregation system based on secure 3-party computation, where the Reporting Origin learns only differentially private aggregates for reports and does not learn any other data associated to a specific client. The architecture of our system is described in Figure 1. We refer to our aggregate protocol as **Bucketization**, as it can be seen as putting reports in buckets representing different attributes. Finally, we show that Bucketization provides better flexibility and performance than prior work.

1.2.1 Flexibility. We consider a report encoding multiple attributes. For example, a 16-bit report could represent two attributes: a 10-bit Ads Category and a 6-bit Customer Region (geographic region). Our protocol allows running a histogram query on *any* selected attributes encoded in the report in *any* order. This enables us to compute more complicated histogram queries. For example, the servers can run a histogram query on Ads Category attribute first. They can deliver the output at this point, or continue with building a histogram based on Customer Region, when Ads Category is equal to a specific value. This behaviour is depicted in Figure 11. In general, **Bucketization allows the analyst to choose the aggregation attributes on the fly (from the collected list of attributes).**

1.2.2 Cheap Sum Computation. Prio [14] describes an approach to compute sum on numerical secret values in the presence of malicious client. Their protocol needs a range proof from the client to be convinced that the client is not "poisoning" the computations with some huge values -this is easy to do for a malicious client because the shares of secret value will be presented in groups with large primes. In Section 3.3, we describe how to convert a secret value from a group with small order to a group with large order so that client can secret share its value within a small range. This means that **we do not need the expensive range proofs from clients.** The client shares its values within the given range by using a small group order and the servers run a conversion protocol that lifts secret shared values to a larger group to allow finalize the sum computation. We harmonize two techniques to achieve it: Quotient Transfer [28] and Oblivious Transfer (OT).

1.2.3 Security guarantees. We formally analyze the robustness and privacy guarantees of our protocol in a few different threat models.

- Bucketization provides **robustness against malicious clients**. We can guarantee that a malicious client cannot cause a report to be counted more than once.
- Bucketization utilizes **differential privacy** (Laplace mechanism) with parameter ϵ. This achieves $(\bar{\epsilon}, \delta = \mathrm{p}\frac{e^{\epsilon}}{1-\mathrm{p}})$-DP, with $\bar{\epsilon} = 0$ and $\delta \in [0, 1]$ with a very small p. In other words, the reported histograms reveal only noisy aggregate counts.

- We first prove **privacy against semi-honest helper servers**. As usual, semi-honest adversaries execute the protocol honestly but try to learn additional information from the protocol transcript.
- Next, we extend the security to a stronger notion: **security against a single malicious server**, whose aim is to learn sensitive information about honest clients by deviating from the protocol execution. However, robustness (correctness of outputs) in the case of a malicious server is not guaranteed, which is also the case in [7, 14].
- Finally, we extend our guarantees to provide privacy against a Reporting Origin that colludes with one of the servers. Our proofs are based on a leakage model, as explained in Section 4.4 and Appendix C.4.

We compare our security and privacy guarantees to most relevant prior work in Table 1.

1.2.4 Complexity. The time complexity of our histogram and sum 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 ($\bar{M} = O(-(1/\epsilon)\ln\delta)$ to achieve $(0, \delta)$-DP for histograms, $(\epsilon, 0)$-DP for sum). The communication complexity is $O(\ell(C+B\bar{M}))$, where ℓ is the report length. The communication required from clients in their reports is only $O(\ell C)$, with some encryption overhead. We compare our time and communication complexity to most relevant prior work in Table 2.

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 protocol, server-to-server communication is also linear in the number of clients. However, our constants are small and the total communication ends up being smaller than for the other proposed protocols, as shown in Table 2.

1.2.5 Experiments. We run experiments to measure the performance of Bucketization on various sizes of reports **for histograms**. The results and analysis are shown in Section 5. We choose three test cases: (1) reports with 16-bit to 24-bit attributes, where we output full-domain histograms (*i.e.*, output 2^{16} to 2^{24} noisy buckets); (2) very large reports, from 32 bits up to 512 bits, encoding multiple 16-bit attributes, where we perform histogram aggregation on an arbitrarily chosen 16-bit attribute, and (3) 256-bit reports, encoding multiple 16-bit attributes, where we compute *heavy hitters* comparing to the iDPF-based scheme in [7].

In all test cases, our protocol shows high efficiency and outperforms the protocols based on (i)DPF. (Prio for all these test cases would be so prohibitively slow that we were not able to run it.) Additionally, the experiment (2) shows extra flexibility in the case of a sparse domain, where multiple attributes are encoded in large reports and the servers are able to aggregate on arbitrarily selected attributes. This cannot be done by (i)DPF-based schemes.

1.3 Our Techniques

1.3.1 Strawman 1: Simple secret sharing. As a first strawman solution, consider the following. The clients will each secret share their reports bit-wise into two shares and encrypt each share for one of two helper servers. The servers will decrypt the encrypted shares as soon as they receive them. In order to evaluate a histogram query requiring filtering by a given attribute, they will exchange the shares of the corresponding bits of each client's reports, which lets them compute the correct attribute for each client, and either reveal the resulting counts or subdivide further. Note that this satisfies the functionality we want: we can compactly represent a long string of attributes in the report and adaptively choose which of these attributes to filter by, and whether to further subdivide the results.

Unfortunately, this reveals the exact counts for each attribute value, so it cannot be differentially private. Additionally, if the helper servers know which client a report comes from, they will learn the attribute value for that client.

1.3.2 Strawman 2: Adding noise. The above issues make it clear that we need a way to add noise to the counts. Since the counts are computed by counting the number of reports with a given attribute, we need to add noise by somehow changing the number or reports with each attribute. It is easier to add reports with a given attribute than to remove them, we will do this by only adding additional reports. Specifically, for each possible value of the attribute we are partitioning by, each server will sample a non-zero random number and add as many *dummy reports* with the particular attribute set and all other attributes set to zero. We will then subtract an appropriate number from the final count so that the expected value remains correct. The server will secret share their augmented set of reports with the other server and perform the count as in Strawman 1, resulting now in noisy counts.

Unfortunately, the helper servers will know which report shares came from the initial set and which were added in the noise addition step. This means that once the attribute values are reconstructed, it will still be easy for each helper server to compute the exact counts.

1.3.3 Strawman 3: Mixnets. This brings us to the third component of our solution, where the helper servers will shuffle the reports so that neither of them knows which reports correspond to the initial data and which were added as noise. One way to do this would be to take a mixnet approach, where the reports are all shuffled first by one server and then by the other.

More precisely, the client would encrypt the shares in such a way that the first server could partially decrypt and shuffle the pairs of shares and then pass them to the next server, who would again partially decrypt and shuffle the pairs of shares. The result would be pairs of ciphertexts, the first being readable by the first server, the second being readable by the second server, but neither server knowing the permutation that was applied. This satisfies the privacy we want: the servers can encrypt their dummy reports just as the client would, and after both servers have shuffled the reports, neither server will be able to tell which reports are original client reports and which are dummy reports.

Protocol	Robustness against malicious clients	Semi-honest servers	Privacy against malicious server	Correctness against malicious server	DP	# of servers
DPF [8, 9, 22]	Requires consistency check	yes	yes	no	no	2
iDPF [7]	Requires sketching protocol	yes	yes	no	yes	2
Prio [1, 14]	Requires range proofs	yes	yes	no	no	2
Bucketization	No cost	yes	yes	no	yes	3

Table 1: Threat model and security guarantee comparison.

Protocol	Time complexity	Server-to-server comm.	Client-to-server comm.
DPF [8, 9, 22]	$(2^\ell C)$DPF.Eval calls	0	$\lambda\ell$
iDPF [7]	$(\ell C^2/t)$DPF.Eval calls	$\ell \log_2(p)$	$\lambda\ell$
Prio [1, 14]	$O(BC)$	$O(C \log_2(p))$	$B \log_2(p)$
Bucketization	$O(C + B\bar{M})$	$O(\ell(C + B\bar{M}))$	ℓ

Table 2: Complexity comparisons of private histogram computation. ℓ is the report 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, but in some other cases $B \ll 2^\ell$); $\bar{M}$ 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.**

However, getting the full flexibility we want would be expensive. Recall that we want to encode many attributes and allow the analyst/Reporting Origin to partition by one attribute and recursively subdivide the resulting partitions based on other attributes. Each such subdivision would require a public-key decryption to be applied to each report, with layers of encryption also increasing the size of the ciphertexts. As described above, this would also require the client to know how many times each attribute must be shuffled, which would prevent the desired flexibility.

1.3.4 Final solution: 3-party shuffle. To address these issues, we replace the mixnet in the above solution with a secure honest-majority three-party oblivious shuffle protocol based on the permutation protocol from [32]. In this protocol, initially two parties have shares of a list of items. With the help of a third party, they perform an MPC protocol, at the end of which two parties each hold new shares of the same list of items, but permuted according to a random permutation that none of the individual servers knows. We assume that each pair of the servers are connected by a secure channel. The protocol is secure against semi-honest adversaries and it also preserves privacy in presence of at most one malicious server.

Thus, our final protocol proceeds as follows. The clients generate secret shares of reports for two servers. The two servers add dummy reports for each attribute, where the number of dummy reports is randomly sampled from an appropriate distribution. The original reports and dummy reports are permuted with the help of third server through an oblivious shuffling subroutine. Finally, shares of the appropriate bits are revealed for each report, the relevant attribute of each report is reconstructed and used to partition the reports. The partitions can be used to compute the histograms or further operations can be applied within the partition.

1.4 Related Work

There exists a big line of prior academic work to solve privacy-preserving aggregate systems for the web, and specifically for conversion measurement. The most intuitive ones are based on general purpose MPC [23, 38]; some utilize a new primitive called Distributed Point Function (DPF) [7–9, 22]; the rest have different flavors [1, 14]. We group the DPF solutions as regular DPF [8, 9, 22] and incremental DPF (iDPF) [7]. In this work, we mainly focus on iDPF [7] and Prio [14], as these approaches are currently being proposed for standardization by IETF [21, 35, 37]. We compare their complexities with our solution shortly and provide detailed descriptions of these proposals in Appendix D. Our approach overall comes with a time complexity advantage compared to both DPF/iDPF and Prio.

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 Appendix D and will compare the Prio with our protocol in Section 5. **We propose a new method in order to avoid client-side proof computations in Section 3.3** .

DPF takes a completely orthogonal approach and introduces a high computational complexity linear to the size of the key space, while avoiding the linear communication complexity between servers. The iDPF-based protocol improves it to complexity quadratic to the number of clients but only for subset histograms. We give a detailed description of an existing proposal using DPF in Appendix D. We compare the performance of our protocol with iDPF [7] in Section 5.

Some prior works achieves secure aggregation based on multi-party shuffling protocols. Mazloom *et al.* propose a protocol for access pattern hiding secure graph computation, which can be applied for histogram aggregation [30]. The computation of histograms follow the same approach (adding dummies for DP, shuffling, and counting) as our work, but with less efficient building blocks, and without our layering techniques with optimizations. Note that layering is the main advantage of our proposal in histogram computations for large report sizes. More specifically, Mazloom *et al.* histogram protocol uses 2-server MPC with garbled circuits for shuffling and noise addition. Thus, the cost would be prohibitive when shuffling a large dataset. Instead, we use 3 servers and cheap operations to get a significant complexity advantage and lightweight MPC in the end. Moreover, our protocol enables computation of sum in addition to histograms (or with a combination of histograms in layered Bucketization) by using a new technique for modulo conversion in order to protect against malicious clients who may report large values to .

Similarly, 2-server mixnet-based protocols with verifiable shuffling incur prohibitive overhead for using public-key operations and zero-knowledge proofs [12, 34].

A private messaging scheme called Vuvuzela involving mixnets and differential privacy can potentially be used for simple histogram queries though it has not been explored [36]. However, this kind of shuffling protocol does not provide privacy guarantee against a malicious server. Its differential privacy is also undefined when used for secure aggregation.

There are also several works [6, 13] that propose differential privacy by shuffling. However, these works consider a slightly different trust model, namely, they assume each client will change their input with some small probability. These inputs are then shuffled with a permutation that the adversary does not know, which – combined with the noise added by all the clients – provides differential privacy with significantly less noise per client than simple local DP would require. However, this inherently requires that there are many honest clients: if the adversary can omit the noise in most of the inputs, then the shuffling cannot provide any additional privacy. This is a problem in our setting, where it may be that the Reporting Origin will be the one deciding which set of reports to submit for aggregation. The helper servers would have no way to verify that the reports came from legitimate clients.

Moreover, the histogram protocol from [13] requires that each client send 1 bit for each element in the domain. In our application the domain is the set of all length ℓ bit strings, so the protocol in [13] would require 2^ℓ bits of communication per client. When we start considering larger strings, this clearly becomes infeasible. A layering approach might be possible, but it would require significantly smaller and thus more layers with that approach. Our layering protocol also allows us to remove the dummies added in the previous layers each time we do a new bucketization; it's not clear how to correct errors in previous layers if using the approach from [13], so the errors would accumulate.

2 PRELIMINARIES

2.1 Notation

We consider a protocol where multiple clients (*e.g.*, web browsers) each input a single data report. The protocol will perform a secure 3-party computation using three helper servers and output results to a Reporting Origin.

We denote the Reporting Origin as $\mathcal{R}$ and the three helper servers as $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$. We use $\mathsf{b} \in \{1, 2, 3\}$ to denote a server index. Unless explicitly stated otherwise, all indices start from 1.

D denotes the dataset of all reports, one from each client; we denote its size by C. The report from client i is denoted by d_i. Each client report is formed with μ attributes. The value of the m-th attribute of d_i is $d_i[\mathsf{m}]$, where $\mathsf{m} \in [\mu]$. It is an element of a group G_m. We assume that G_m includes a "dummy value" $\perp$ and that $d_i[\mathsf{m}] \in G_\mathsf{m} \setminus \{\perp\}$. The $\perp$ value is reserved for internal use in our protocol.

There are several ways to implement the groups G_m. For example, if the value of the m-th attribute, $d_i[\mathsf{m}]$, is an ℓ_m-bit string and $1 \ldots 1$ is never used to report, we could set $G_\mathsf{m} = \mathbb{Z}_2^{\ell_\mathsf{m}}$ and $\perp = 1 \ldots 1$. For the rest of the paper, we will follow this notation, but our construction adapts to other groups as well.[2] We let L_m be the order of G_m and let $G = G_1 \times G_2 \times \cdots \times G_\mu$. Hence, d_i for $i \in [C]$ is an element of G. Finally, we allow some attributes to be "numerical" so that they can be summed. In that case, $G_\mathsf{m} = \mathbb{Z}_p$ for some odd prime p. [3]

In the case of binary representation, an ℓ-bit d_i will have μ different attributes, such that $\ell = \sum_{\mathsf{m}=1}^{\mu} \ell_\mathsf{m}$, meaning each attribute m is ℓ_m 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 $\{\mathcal{B}_j\}$, for $j \in G_\mathsf{m}$. Among these buckets, one bucket is reserved for the dummy value. We denote the bucket of dummies by $\mathcal{B}_\perp$. We note that number of output buckets is **not necessarily** the same as the size of the report space. This is due to the fact that in practice, even though the report is represented with 32 bits, the values can be limited to a much smaller space. We will clarify when this is the case.

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 protocol with a small example. We work with 10 client reports, where each report consists of three attributes: Gender (represented with 2 bits for "he/she/they/$\perp$"), Category (represented with 3 bits), and Age (represented within a range of $[0, 100]$). In this example, we take $G_1 = \mathbb{Z}_2^2$ and $G_2 = \mathbb{Z}_2^3$, with group operation bit-wise XOR, $G_3 = \mathbb{Z}_{101}$ with group operation modular addition. Since we have $C = 10$, D consists of 10 reports. Each report will have $\mu = 3$ attributes with 5 bits in total: $d_i[1] = \mathsf{x}_1\mathsf{x}_2$ and $d_i[2] = \mathsf{y}_1\mathsf{y}_2\mathsf{y}_3$ as well as a secret value $d_i[3]$ within a specified range. 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 reports with $d_i[1] = 11$. An example of a corresponding dataset D (without considering any secret sharing yet) is given

[2] If all ℓ_m-bit strings must be used, we let $G_\mathsf{m} = \mathbb{Z}_{2^{\ell_\mathsf{m}}+1}$ and $\perp = 2^{\ell_\mathsf{m}}$.

[3] This group represents the final group where the secret shares of client values are lifted. See Section 3.3 for the details.

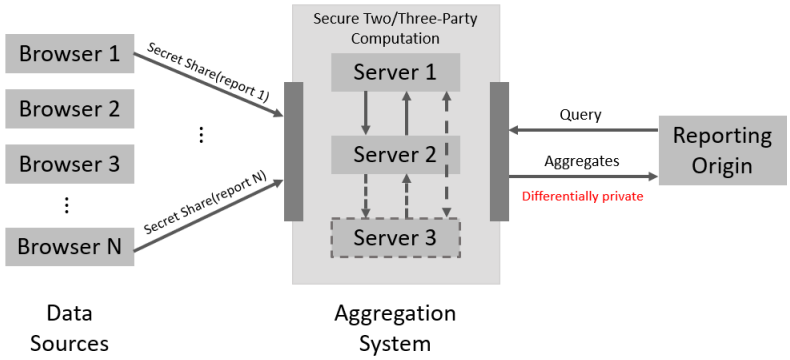

Figure 1: System Architecture.

$d_1 = 00\ 010\ 25$
$d_2 = 01\ 001\ 32$
$d_3 = 00\ 000\ 19$
$d_4 = 00\ 010\ 64$
$d_5 = 10\ 010\ 53$
$d_6 = 00\ 010\ 42$
$d_7 = 01\ 110\ 21$
$d_8 = 01\ 001\ 39$
$d_9 = 10\ 000\ 45$
$d_{10} = 10\ 010\ 76$

Figure 2: A visualization of our small example: the dataset D with $C = 10$ reports with $\mu = 3$ attributes: 2 of them bitstring with total of 5 bits and 1 of them is a numerical value.

$d_1^1 = 10\ 011\ 33$
$d_2^1 = 00\ 011\ 45$
$d_3^1 = 01\ 111\ 10$
$d_4^1 = 11\ 110\ 52$
$d_5^1 = 01\ 001\ 11$
$d_6^1 = 10\ 001\ 90$
$d_7^1 = 00\ 011\ 2$
$d_8^1 = 01\ 101\ 55$
$d_9^1 = 00\ 011\ 13$
$d_{10}^1 = 11\ 101\ 82$

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

$d_1^2 = 10\ 001\ 68$
$d_2^2 = 01\ 010\ 88$
$d_3^2 = 01\ 111\ 9$
$d_4^2 = 11\ 100\ 12$
$d_5^2 = 11\ 011\ 42$
$d_6^2 = 10\ 011\ 53$
$d_7^2 = 01\ 101\ 19$
$d_8^2 = 00\ 100\ 85$
$d_9^2 = 10\ 011\ 32$
$d_{10}^2 = 01\ 111\ 95$

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

Figure 3: Secret shares of D held by $\mathcal{S}_1$ and $\mathcal{S}_2$.

in Figure 2. Notice that there are no reports with $d_i[1] = 11$ or $d_i[2] = 111$, as these buckets are reserved for the dummy reports.

2.2 Secret Sharing

In the GMW protocol [23], a secret value is information-theoretically shared between multiple parties for secure multi-party computation. Let G denote a finite additive group. In the 2-party case, a client can share a secret value $k \in G$ to two servers by first uniformly sampling $r \leftarrow G$, sending r to one server, and $k - r$ to the other server. Neither share alone reveals any information about the value k. Such a scheme works for both Boolean circuits and arithmetic circuits; it requires no communication for the 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: $d_i[j] = d_i^1[j] \oplus d_i^2[j]$, for $i \in [10]$, $j \in \{1, 2\}$ $d_i[3] = d_i^1[3] + d_i^2[3] (\mod 101)$. We depict the shares in Figure 3.

2.3 Differential Privacy

Differential privacy [17, 18] protects the privacy of individual rows in a database, while still allowing meaningful aggregate queries to be made. For each (ordered) database $D \in \chi^C$, 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 two rows and have the same cardinality.[4]

We say that a mechanism $\mathcal{M}$ is $(\bar{\epsilon}, \delta)$-differentially private ($(\bar{\epsilon}, \delta)$-DP) for parameters $\bar{\epsilon} \geq 0$ and $\delta \in [0, 1]$, if for any $S \subseteq \text{Range}(\mathcal{M})$ and any neighboring $\bar{D}$ and $\bar{D}'$,

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

In our results, $\bar{\epsilon} = 0$.

The property of differential privacy is maintained through *post-processing*. Informally, this 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.

To achieve $(\bar{\epsilon}, \delta)$-DP for our protocol, we utilize the standard *Laplace mechanism*. In the Laplace mechanism, we add noise drawn from the Laplace distribution to the output of a statistical aggregate. The probability density function of Laplace distribution is $\text{Lap}(X, b) = (1/2b) \exp(-|X|/b)$; it has zero mean and standard deviation $\sigma = \sqrt{2b^2}$. The Laplace distribution we consider from now on is Lap $(X, 1/\epsilon)$. So, in our notation, ϵ **denotes a parameter**

[4] We have two different notions of neighboring: one for histograms and the other for sum. Formally, neighboring in histogram means L_1 norm of two database $\|\bar{D} - \bar{D}'\|_1 = 2$ and $\sum_i \bar{D}(i) = \sum_i \bar{D}'(i)$. For summing, we have another notion of neighboring: for mth attribute $\bar{D}$ and $\bar{D}'$ are neighbors if $\exists\ i$ such that $\forall\ j$ and v with $i \neq j$ or $v \neq \text{m}$, $\bar{D}_j[v] = \bar{D}'_j[v]$ and $\bar{D}_i[\text{m}] = \bar{D}'_i[\text{m}] \pm 1$.

of the Laplace distribution, whereas $\bar{\epsilon}$ **is a privacy parameter of our protocol**.

Finally, our sum protocol utilizes the standard Laplace mechanism where the noise is added by both servers and achieves $(\epsilon, 0)$-DP.

2.4 Oblivious Random Shuffling

We will make use of an oblivious shuffling protocol that runs between three servers. It inputs a dataset, initially secret shared between two servers, and outputs 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.

Multiple oblivious shuffling protocols have been presented in prior work. Chase *et al.* [11] uses the idea of an oblivious permutation for 2-party oblivious random shuffling. However, for performance reasons, the 2-party approach is insufficient for our protocol. Instead, Mohassel *et al.* [32] proposed an oblivious permutation protocol in the honest majority 3-party setting, with linear computation and communication cost. We will instantiate a modified version of [32] in Section 3.2.

3 SUBROUTINES

Before we describe the details of our subroutines, we depict the data and computation flow of our protocol in Figure 4. Section 3.1 will describe how step 2 in Figure 4 is computed. Section 3.2 will describe how step 3 in Figure 4 is computed. The full protocol is described in Section 4.

3.1 Differential Privacy with Constraints For Histograms

To achieve differential privacy for histograms, we add noise to the counts for each bucket. Ideally, we would like to use the Laplace mechanism, which samples both positive or negative noise from the Laplace distribution. However, we regard dummy reports added to buckets as noise, and in our protocol we can only *add* dummy reports – not remove any reports. One solution would be to follow the proposal from Medina *et al.* [33] as a general framework for private optimization with non-negative constraints. Instead, we hope to still align with the standard Laplace mechanism, as it provides a better balance between privacy and utility and is simpler to analyze.

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 leads us to generate discrete positive or negative noise.

First, we define the truncation of Laplace. We sample a noise n from $\mathsf{Lap}(X, 1/\epsilon)$: $n \leftarrow \lfloor \mathsf{Lap}(X, 1/\epsilon) \rceil$ (rounded to the closest integer[5]), which could be negative. If $n < -M$, we sample again until $n \geq -M$. We obtain a PDF for n, where $\Pr(X) = 0$ if $X < -M - 1/2$ and $\Pr(X) = \mathsf{Lap}(X, 1/\epsilon)/(1 - \mathsf{p})$ otherwise, with p defined as the resampling probability, which we will define shortly. Finally, we shift the output by adding M to n, which ensures that $n + M$ is always non-negative.

In our application, we can use $n + M$ as the number of dummy reports to be added to a particular bucket and later subtract the value M (a public parameter) from each count, leaving the original Laplace noise n added to the buckets.

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

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

Solving for M, we obtain $M = -(1/\epsilon)\ln(2\mathsf{p}) - 1/2$. As an example, if we want $\mathsf{p} \approx 10^{-6}$, M becomes $M = \lceil 13.12/\epsilon - 0.5 \rceil$.

3.1.2 Computation of average dummy reports. As described above, the noise added in each bucket consists of $n + M$ dummy reports, where n is sampled from the truncated Laplace distribution. Since

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

the expectation value of n+M becomes $\bar{M} = M + \frac{\mathsf{p}}{1-\mathsf{p}}\ (1/\epsilon + M + 1/2)$. As expected, this is very close to M.

We provide a suggested way to implement noise sampling in Appendix A. 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 of dummy reports for each of these four buckets, with $\mathcal{B}_4$ consisting entirely of dummy reports.

Suppose the noise vector $\mathcal{N}^1 = (2, 1, 0, 2)$ for $\mathcal{S}_1$.[6] This means that a total of $n^{(1)} = 5$ new reports will be added: two reports $d^1_{1,1}, d^1_{2,1}$ to $\mathcal{B}_1$, one report $d^1_{1,2}$ to $\mathcal{B}_2$, and two reports $d^1_{1,4}, d^1_{2,4}$ to $\mathcal{B}_4$.

In step **Generating dummy reports**, each dummy report (for $\mathcal{S}_1$) will have the form $d^1_{i,j} = [j|111]$, where j is represented in binary with 2 bits and other attribute is set to the reserved value 111. The same steps are repeated for $\mathcal{S}_2$, with different total noise vector $\mathcal{N}^2$ and dummy report count $n^{(2)}$.

Finally, in the step **Appending shares to dummy reports**, $\mathcal{S}_1$ will append the $n^{(1)} = 5$ dummy reports, as well as $n^{(2)}$ fake reports with all bits filled with 0, to its true reports from 10 clients. This way, the only communication needed between $\mathcal{S}_1$ and $\mathcal{S}_2$ is the exchange of the numbers $n^{(1)}$ and $n^{(2)}$.

Concretely, suppose the noise vector of $\mathcal{S}_2$ be $\mathcal{N}^2 = (1, 0, 0, 1)$. We depict the noise addition following our example in Figure 6.

If the servers now reveal the buckets for the first attribute, they can then organize the reports in buckets accordingly:

$$\mathcal{B}_1 = \{d^{\mathsf{b}}_1, d^{\mathsf{b}}_3, d^{\mathsf{b}}_4, d^{\mathsf{b}}_6, d^{\mathsf{b}}_{11}, d^{\mathsf{b}}_{12}, d^{\mathsf{b}}_{16}\}, \quad \mathcal{B}_2 = \{d^{\mathsf{b}}_2, d^{\mathsf{b}}_7, d^{\mathsf{b}}_8, d^{\mathsf{b}}_{13}\},$$

$$\mathcal{B}_3 = \{d^{\mathsf{b}}_5, d^{\mathsf{b}}_9, d^{\mathsf{b}}_{10}\}, \quad \mathcal{B}_4 = \{d^{\mathsf{b}}_{14}, d^{\mathsf{b}}_{15}, d^{\mathsf{b}}_{17}\}.$$

Finally, they reveal the requested histogram as the counts of the buckets $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$, ignoring the dummy bucket $\mathcal{B}_4$.

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

[6] Note that sampling such a noise vector is in practice unrealistic. Instead, we would expect to get noise values centered around our chosen shift parameter M. We use this small vector in this example for the sake of simplicity.

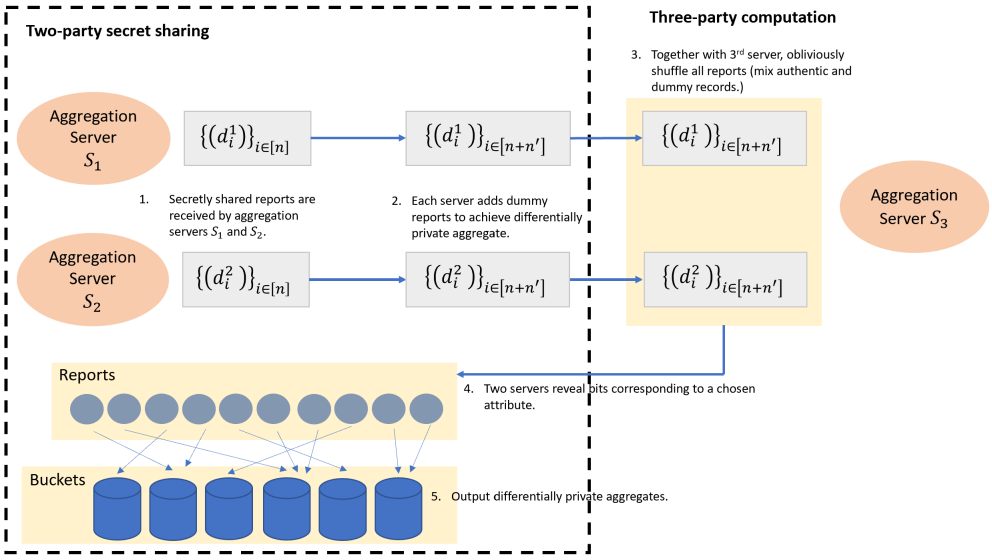

Figure 4: The full data and computations flow between three servers. The secret sharing and adding noise with dummy reports run with two servers. The third server is triggered when the oblivious shuffling is triggered. The buckets are output by revealing secrets from two servers again.

3.2 Oblivious Random Shuffling

In our work, we take the Mohassel *et al.* [32] protocol and modify it into an efficient honest-majority 3-party oblivious random shuffling protocol with linear complexity. The protocol is formally described in Figure 7 and illustrated in Figure 12 (in Appendix E).

In **Initialize** step, we allow each pair of servers to jointly sample a permutation and a vector of masks. In practice, the permutation and mask can both be sampled from random seeds so the communication cost is trivial. Compared to 2-party protocols, the presence of the third party leads to linear instead of linearithmic computational and communication overhead. During the **Shuffling** phase, each party needs to send only one message, resulting in a constant number of rounds of communication. We do not need commitments or interactions, because of the third server we introduced. More precisely, when one of the servers is malicious, the sampling from the (third) honest server is enough to create a uniform distribution for the permutation.

The formal privacy of the protocol in the semi-honest server model is proven in Theorem C.1 in Appendix C.1. Informally, what we prove when all three servers are semi-honest is that the views of any corrupt party can be perfectly simulated. At the initialization phase, each pair of parties jointly sample a random permutation and a random mask vector. These can be simulated by uniformly sampling and sending to the adversaries random 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 $A := \pi_{23}(\pi_{12}(D^2) + R_{12}) + R_{23}$ from $\mathcal{S}_2$, 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 so there is nothing 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. In the formal treatment we will assume collusion between Reporting Origin and one of the servers. Reporting Origin learns the final output and we assume the leakage of A' and B', as well.

3.3 Secure Modulo Conversion for Sum

So far, we only worked with additively secret shared bitstring keys to build histograms. However, in our system, the reports are formed with real number values. It is more difficult to work with real numbers to compute the sum as opposed to the histograms: the clients may cheat during secret sharing and send some very large values to the reporting origin. Since the computations are done in MPC settings by secret sharing the reported values, it will not be caught and the end result will return as a meaningless sum.

Malicious Clients: There are two ways to enforce clients to secret share a value within a range: boolean secret share within a range and then lift it to a larger field to accommodate sums or arithmetic secret share with small modulo and lift it to a larger field. They are detailed below. Let the client value be d_i which is an ℓ-bit string. Let p be an ℓ-bit odd integer which determines the range of d_i.

(1) **Boolean-to-Arithmetic(B2A) Share Conversions:** This technique is used in [1] and stated that it is more efficient than the Arithmetic-to-Arithmetic(A2A) conversion. Each client secret shares its value such that $d_i = d_i^1 \oplus d_i^2$ where d_i^{b}'s are Boolean shares. Since the length of the shares are

Protocol Π_{NoiseGen}

Parameters. An attribute index m. L_{m} buckets. There are two parameters: ϵ and M. We use truncated Laplace distribution $\frac{\mathrm{Lap}(X,1/\epsilon)}{1-\mathsf{p}}$ for $X \geq -M - \frac{1}{2}$ with $\mathsf{p} = \frac{1}{2}\mathsf{e}^{-\epsilon(\mathsf{M}+\frac{1}{2})}$.

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.

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

Generating dummy reports. For each $\mathsf{b} \in \{1,2\}$ and each bucket $j \in G_{\mathsf{m}}$, $\mathcal{S}_{\mathsf{b}}$ creates $n^{\mathsf{b}}_j + M$ dummy reports as follows: for $i \in [n^{\mathsf{b}}_j + M]$, set $d^{\mathsf{b}}_{i,j}[\mathsf{m}] = j$ and $d^{\mathsf{b}}_{i,j}[v] \leftarrow \perp (\in G_v)$ (in the case of a numerical attribute for summing, $d^{\mathsf{b}}_{i,j}[v] \leftarrow 0$ $(\in G_v)$ if v is the attribute to be summed) for $v \in [\mu] \setminus \{\mathsf{m}\}$ (which form one dummy report $d^{\mathsf{b}}_{i,j}$). $\mathcal{S}_{\mathsf{b}}$ forms all of $n^{(\mathsf{b})} = \sum_{j \in G_{\mathsf{m}}} (n^{\mathsf{b}}_j + M)$ dummy reports $d^{\mathsf{b}}_{i,j}$ as $D^{\mathsf{b}}_{\mathsf{dum}}$.

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

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

Figure 5: The protocol of DP noise generation.

D^1_{priv}
d^1_1 = 10 011 33
d^1_2 = 00 011 45
d^1_3 = 01 111 10
d^1_4 = 11 110 52
d^1_5 = 01 001 11
d^1_6 = 10 001 90
d^1_7 = 00 011 2
d^1_8 = 01 101 55
d^1_9 = 00 011 13
d^1_{10} = 11 101 82
d^1_{11} = 00 111 0
d^1_{12} = 00 111 0
d^1_{13} = 01 111 0
d^1_{14} = 11 111 0
d^1_{15} = 11 111 0
d^1_{16} = 00 000 0
d^1_{17} = 00 000 0

(a) D^1_{priv}

D^2_{priv}
d^2_1 = 10 001 68
d^2_2 = 01 010 88
d^2_3 = 01 111 9
d^2_4 = 11 100 12
d^2_5 = 11 011 42
d^2_6 = 10 011 53
d^2_7 = 01 101 19
d^2_8 = 00 100 85
d^2_9 = 10 011 32
d^2_{10} = 01 111 95
d^2_{11} = 00 000
d^2_{12} = 00 000 0
d^2_{13} = 00 000 0
d^2_{14} = 00 000 0
d^2_{15} = 00 000 0
d^2_{16} = 00 111 0
d^2_{17} = 11 111 0

(b) D^2_{priv}

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

Protocol Π_{RandShuf}

Notation. When the operators $\{+,-\}$ are applied to vectors, they mean element-wise addition and subtraction. Permutations π are on the index set $[C]$ and $\pi(D)$ is defined by $\pi(D)_{\pi(i)} = D_i$

Input. For $\mathsf{b} \in \{1,2\}$, S_{b} inputs $(\mathsf{shuffle}, D^{\mathsf{b}})$ where $D^{\mathsf{b}} := (d^{\mathsf{b}}_i)_{i \in [C]}$, with $d^{\mathsf{b}}_i \in G$ (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 $A' := \pi_{13}(A) - R_{13}$.
(3) $\mathcal{S}_3$ computes $B' := \pi_{13}(\pi_{23}(B) - R_{23}) + R_{13}$.

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

Figure 7: The protocol of oblivious random shuffling.

retained, clients cannot report very large shares. When they send more than ℓ-bit shares, they are discarded. Computing sum with Boolean additive shares are easy. However, to sum the Boolean shares are too costly. Therefore, we need to switch from Boolean representation to algebraic representation with large enough modulus to compute the sum.

As suggested by [1], B2A share conversion has been well-studied technique and the current most efficient technique in two-party semi-honest setting is due to [16]. It is reported that to convert a pair of ℓ-bit Boolean shares, Prio+ uses ℓ independent instances of oblivious transfer (OT) with each of them communicating $\frac{\ell+1}{2}$ bits on average. Hence, the communication complexity for ℓ bit Boolean shares to ℓ-bit arithmetic shares is $O(\ell^2)$. The simple scheme to sum that uses B2A conversion is described between page 14 and 18 in [1].

The OT in ABY construction requires pre-computations in an offline phase. Instead of OT, we could use bit-wise multiplication which also require pre-computations. However, generating the pre-computations can be omitted by

introducing a (honest and non-colluding) 3rd server which would generate the randomness.

(2) **Arithmetic-to-Arithmetic (A2A) Share Conversion:** Each client secret shares its value such that $d = (d^1 + d^2) \bmod p$ where the d^b's are arithmetic shares modulo p. This way, the servers can detect if a client sends a too large value which is bigger than p. However, to compute the sum, we need to lift the shares to a larger domain and it is known as quite expensive. **However, we argue that it may not be the case**. We will use techniques from Quotient Transfer (QT) which is first introduced in [28] and leveraged in other works [29].

Let p' be an odd prime modulus that we like to convert the shares of d from modulo p. Briefly, we will describe how to convert secret shares of a value (modulo a small odd integer p) into secret shares of the same value in a large odd prime p'. We start with the following observation: $d = (d^1 + d^2) \bmod p$ implies that

$$d = d^1 + d^2 - q * p \tag{1}$$

where $q \in \{0, 1\}$ *and if* d is an even value, then we know that q is equal to the XOR of the least significant bit (lsb) of d^1 and d^2 as shown below.

$$\begin{aligned} q &= lsb(d^1) \oplus lsb(d^2) \\ &= lsb(d^1) + lsb(d^2) - 2 * lsb(d^1) * lsb(d^2) \end{aligned} \tag{2}$$

This observation suggests that if we know how to secret share q from the lsb of d^b, then we can compute the modulo conversion easily. In other words, sharing the secret bit q into shares of p' is enough to lift the shares of d from modulus p into large odd prime p'. This technique is called *quotient transfer* and first used in [28]. Let q^1 and q^2 be the shares of q modulo p'. We want to obtain two shares $(d^1)' = (d^1 - q^1 * p) \bmod p'$ and $(d^2)' = (d^2 - q^2 * p) \bmod p'$ such that $d = (d^1)' + (d^2)' \bmod p'$. If we share the q between the servers, then the conversion of d to a larger field would be done locally. For that, we need a 2-party protocol which takes $a = \text{lsb}(d^1)$ from $\mathcal{S}_1$ and $b = \text{lsb}(d^2)$ from $\mathcal{S}_2$ and outputs the m^1 to $\mathcal{S}_1$ and m^2 to $\mathcal{S}_2$ such that $m^1 + m^2 \bmod p' = \text{lsb}(d^1) * \text{lsb}(d^2)$: i.e. an OT protocol to multiply the least significant bits. Finally, each server b computes first $q^b = \text{lsb}(d^b) - 2 * m^b$ and then $(d^b)' = d^b - q^b * p \bmod p'$. The total cost of this protocol is 1 OT per client input. It is a factor of ℓ faster than the Prio+ proposal because it requires only one OT per record instead of ℓ OTs where ℓ is typically bigger than 64.

To address the requirement on d to be even, the servers can multiply d^b by 2 to make the reported values even after secret sharing. Then, the result of the conversion will be divided by 2 which is possible because p' is odd, i.e. 2 is invertible.

4 REPORT BUCKETIZATION

The goal of report bucketization is to arrange a set of reports D collected from C clients into buckets according to a subset of encoded attributes, while preserving privacy of the individual reports. For attribute m of ℓ_m bits, there are $L_m = 2^{\ell_m}$ buckets in total. Our full protocol involves four parties: a Reporting Origin $\mathcal{R}$ and three helper servers $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$.

Protocol $\Pi_{A2AConvert}$

Participants: Two helper servers $\mathcal{S}_1$ and $\mathcal{S}_2$. Note that this protocol utilizes an oblivious transfer (OT) protocol as a subroutine and our instantiations of such OT runs with **three** servers for better performance though it could run with two servers.

Initialize. $\mathcal{S}_1$ and $\mathcal{S}_2$ receive shares of records $D^1 := \{d_i^1\}_{i\in[n]}$ and $D^2 := \{d_i^2\}_{i\in[n]}$ respectively.

(1) Each server b checks if d_i^j is in modulo p. If not, they communicate to the counterpart to discard the reported value. The valid shares are multiplied by two so that it satisfies $2 * d_i = (d_i^1 + d_i^2) \bmod p$.
(2) Each server b computes $\text{lsb}(d_i^b)$. Then, they run an OT protocol (as described below in Figure 9) with their least significant bits to compute the shares of $\text{lsb}(d_i^1) * \text{lsb}(d_i^2)$. At the end, $\mathcal{S}_1$ obtains m_i^1 and $\mathcal{S}_2$ obtains m_i^2 such that $m_i^1 + m_i^2 \bmod p' = \text{lsb}(d_i^1) * \text{lsb}(d_i^2)$.
(3) Each server b computes $q_i^b = \text{lsb}(d_i^b) - 2 * m_i^b \bmod p'$.
(4) Each server b computes $(d_i^b)' = d_i^b - q_i^b * p \bmod p'$.

Output: $\mathcal{S}_1$ and $\mathcal{S}_2$ outputs shares of records $D^1 := \{(d_i^1)'\}_{i\in[n]}$ and $D^2 := \{(d_i^2)'\}_{i\in[n]}$ respectively, which are lifted shares of $2 * d_i$ modulo p'. Thus, it satisfies that $d_i = \frac{(d_i^1)' + (d_i^2)'}{2} \bmod p'$.

Figure 8: The pseudocode for secure modulo conversion which uses Quotient Transfer.

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. Internally, two of the servers run DP noise generation and all three run the oblivious random shuffling protocol.

The input to our full protocol is a dataset of reports $D = (d_i)_{i\in[C]}$, initially secret shared between two non-colluding servers. To ensure that the servers 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 reports.[7] At the end of the protocol, two servers obtain a new secret shared vector $D_{\text{priv}} = (d_i)_{i\in[C+n']}$, where all reports with the same attribute values ("buckets") are stored consecutively in D_{priv}. Note that the size of D_{priv} becomes $C + n'$, where n' represents appended dummy reports, ensuring that the outputs revealed to the Reporting Origin and helper servers are differentially private.

[7] This is aligned with the Internet-Draft [21].

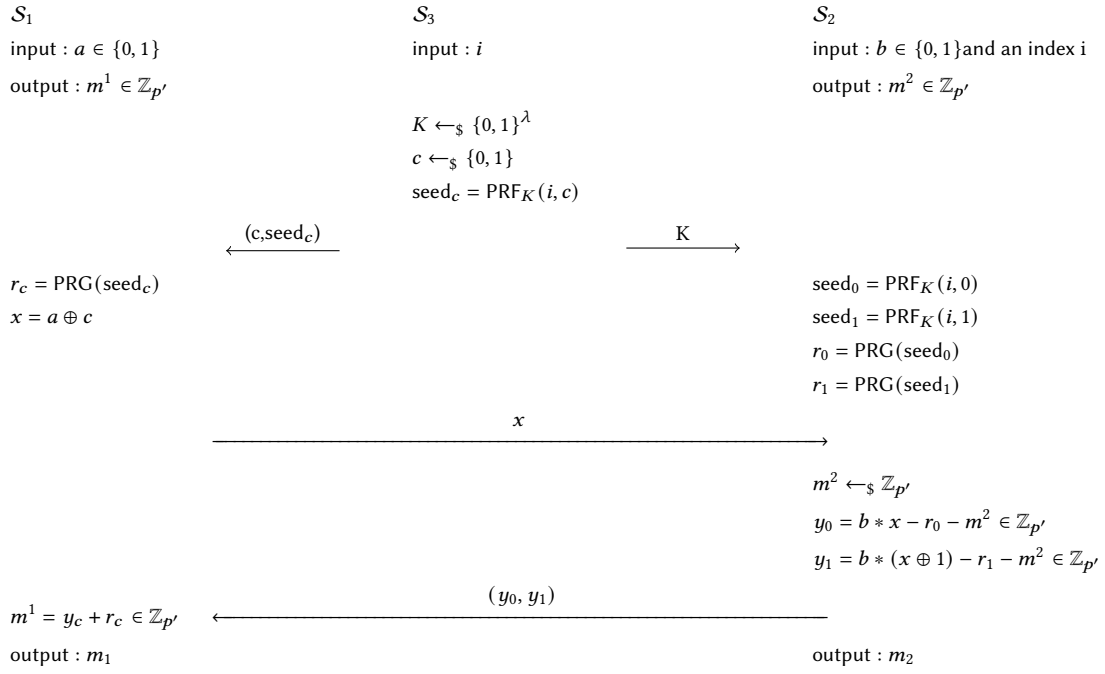

Figure 9: 3-party Oblivious Transfer to compute the arithmetic shares of multiplication of two bits a and b. It is adapted from [3]. $\mathcal{S}_3$ inputs only the index number i for the i^{th} iteration. This protocol is run with the same K for each record in parallel. The communication complexity is $(\lambda + 2)$ bits (λ bits for seed$_c$ and 2 bits for c and x) and 2 group elements in $\mathbb{Z}_{p'}$, y_0 and y_1.

4.1 Private Histogram Protocol Description

We describe our Bucketization protocol as a full procedure $\Pi^{\text{Hist}}_{\text{Bucketize}}$ in Figure 10. Let $D = (d_i \in G)_{i\in[C]}$ be the dataset of reports collected from clients. $\Pi^{\text{Hist}}_{\text{Bucketize}}$ 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 $D^1 = \left(d_i^1\right)_{i\in[C]}$ and $D^2 = \left(d_i^2\right)_{i\in[C]}$, such that $d_i = d_i^1 + d_i^2$ 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 reports $D_{\text{priv}} = (d_i)_{i\in[C+n']}$.

After generating D_{priv}, the servers $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ execute a 3-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 servers can trace back any report to the original report, nor can they distinguish between the authentic reports and dummy reports. The output will be a permuted dataset $D_{\text{priv\_perm}} = \left(d_i'\right)_{i\in[C+n']}$, secret shared between the helper servers $\mathcal{S}_1$ and $\mathcal{S}_3$.

For each $i \in [C + n']$, the servers reveal the selected attribute $d_i[\text{m}]$ and bucketize the reports into buckets $\{\mathcal{B}_j\}$ for $j \in G_{\text{m}}$ according to the attribute value.

Finally, the buckets with size less than a pruning threshold t, as well as the dummy bucket $\mathcal{B}_\perp$, are discarded. A value for t is typically decided based on the DP parameter $\bar{M}$, as $\bar{M}$ indicates the dummy reports 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 them to $\mathcal{S}_1$ and $\mathcal{S}_2$, to run the protocol again if needed (see below). While this resharing is not really necessary in implementations, we include it in the protocol description in order to be able to use $\Pi^{\text{Hist}}_{\text{Bucketize}}$ repeatedly. **The protocol outputs the counts for each bucket, shifted by the public parameter M described in Section 3.1.**

We illustrate the procedures of dummy reports addition and oblivious shuffling in Figure 13 (in Appendix E).

4.2 Private Sum Computation

We described how to lift a secret shared value from a small modulus to larger one in Section 3.3. Servers run the modulus conversion for each record using modulus p' which is large enough so that the final sum added a Laplace noise is always between $\frac{-p'}{2}$ and $\frac{p'}{2}$. Then, each server locally adds the shares of every value to be summed to obtain sum of all shares. Each server adds a rounded Laplace noise with parameter ϵ (rounding done to the nearest integer). The local sums are revealed to compute the noise sum of numerical attribute values. This sum modulo p' gives a number between $\frac{-p'}{2}$ and $\frac{p'}{2}$ as the noisy aggregated sum. Using the same analysis as

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

Parameters. C as the total number of collected reports. Parameters ϵ and M for Π_{NoiseGen}.

Input. A query index m to indicate which attribute to bucketize on; shares of $\mathcal{S}_1$ and $\mathcal{S}_2$ as $D^1 = \left(d_i^1\right)_{i\in[C]}$ and $D^2 = \left(d_i^2\right)_{i\in[C]}$, such that $d_i = d_i^1 + d_i^2$; 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 $D^\mathsf{b}_{\mathsf{priv}} = \left(d_i^\mathsf{b}\right)_{i\in[C+n']}$ with appended n' dummy reports.

(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 $D^b_{\mathsf{priv\_perm}} = \left(d_i'^b\right)_{i\in[C+n']}$ 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 $d'_i[\mathsf{m}]$. They allocate L_m empty buckets $\{\mathcal{B}_j\}$, one for each $j \in G_\mathsf{m}$. For $i \in [C + n']$, $\mathsf{b} \in \{1, 3\}$, $\mathcal{S}_b$ puts the report $d_i'^b$ into the corresponding bucket according to the value of $d'_i[\mathsf{m}]$.

(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 reports and the dummy buckets (indexed as $j = \perp$) along with all the reports in it. For the shares of noisy database from $\mathcal{S}_1$ and $\mathcal{S}_3$, $\mathcal{S}_3$ generates two secret shares of $D^3_{\mathsf{priv\_perm}}$ 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 reports for each bucket so that they can be used as input for further instance of the protocol.

Figure 10: Our Histogram Protocol from Oblivious Shuffling.

for histogram, from the viewpoint of a malicious server, the sum protocol is $(\epsilon, 0)$-DP.

4.3 Layered Bucketization

In this section we will describe how our method provides flexible histogram aggregations. For reports with multiple attributes, our protocol supports aggregation on an arbitrary set of attributes by recursively invoking $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$.

For example, our method enables running queries such as

SELECT COUNT(Category) FROM D
WHERE Gender = "She" GROUP BY Category

as long as there is enough data in the corresponding bucket. Moreover, such an approach enables a performance gain for attributes with large report sizes. For example, if an 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} reports represented with 32 bits, layering Bucketization of the 32 bits reports into three (10, 10, and 12 bits) and pruning the buckets after each layer, allows us to run the protocol efficiently even for relatively large reports with sparse domains. A formal description of the layered algorithm is given in Algorithm 1.

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 D^1 and D^2 for $\mathcal{S}_1$ and $\mathcal{S}_2$. label and i are reserved for the recursion and set to $\perp$ by default.
Output: Histogram on attributes in $\mathcal{M}$.

procedure LAYERED($\mathcal{M}$, label , i, D^1, D^2)
  if (label, i) = ($\perp$, $\perp$) **then**
    Set label = null, $i = 1$.
  end if
  if $i \le \lambda$ **then**
    Call $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ with inputs (m_i, D^1, D^2).
    for each produced bucket B_j **do**
      Set v to the attribute value $d[m_i]$.
      Set D^1 and D^2 to the shares of the bucket.
      Call LAYERED($\mathcal{M}$, label $\mid v, i + 1, D^1, D^2$).
    end for
  else
    Output label.
    Subtract the parameter M (used in $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$) from each value in count(D) and output the result.
  end if
end procedure

Concretely, when the first layer is done with m_1, only the corresponding bits are revealed. Then, the procedure bucketizes on the next attribute index m_2 by generating noisy reports, creating buckets for each possible value of m_2, and setting the values for other attributes to their dummy values.

We discuss the complexity of the layered protocol and further optimizations in Appendix B.

4.3.1 Example. We continue with our toy example. The reports consist of 5-bit attribute 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

$$\{00\ 010\ 25, 00\ 000\ 19, 00\ 010\ 42, 00\ 010\ 64, 00\ 111\ 0, 00\ 111\ 0, 00\ 111\ 0\},$$

yielding 7 reports where the second and third attributes are still secret shared. Then, suppose the protocol goes to the second layer on $\mathcal{B}_1$, where bucketization is run on the second attribute. For this example, we focus on the bucket $\mathcal{B}'_3$, which counts the reports where $d_i[1] = 00$ and $d_i[2] = 010$. Suppose $\mathcal{S}_1$ sampled 2 and $\mathcal{S}_2$ sampled 1 to add in $\mathcal{B}'_3$, 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 first attribute is 00 and second attribute is 010, will output 6, instead of 3. Finally, in the second layer, more dummy reports are added with $d_i[2] = 111$ 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.[8]

In an extreme case we consider every single bit of the report 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.

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

4.4.1 Differential privacy. The $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ protocol is $(\bar{\epsilon}, \delta)$-DP, with $\delta = 2\mathsf{p}\frac{e^{\epsilon}-1}{1-\mathsf{p}}$ for a failure probability $\mathsf{p} = \frac{1}{2}e^{-\epsilon\left(M+\frac{1}{2}\right)}$, where ϵ is the standard Laplace parameter. We prove the statement in Appendix C.2.

We can tune the parameters for our protocol quite nicely. If we want our protocol to achieve $(\bar{\epsilon} = 0, \delta = 2^{-40})$-DP, then we take $\epsilon = 1$ in Laplace distribution and $M = 28$, which implies $\mathsf{p} \approx 2^{-42}$. So, for each bucket, we add 28 dummy reports on average.

In our protocol, we provide privacy against a malicious server by having both servers add noise. A malicious server can subtract the dummy reports it added from the final histogram and learn the histogram with dummy reports only from the other server. Earlier, we computed the average number of dummy reports per bucket as 28 for $(\bar{\epsilon} = 0, \delta = 2^{-40})$-DP. This becomes better if all the participants are honest: it is enough for each honest server to add 14 dummy reports on average per bucket, because if both are honest, the total 28 is preserved.

These parameters change when we use layered Bucketization with λ layers. For example, for $\epsilon = 1, \lambda = 16, (2\lambda - 1)\delta = 2^{-35}$, *i.e.*, $(\bar{\epsilon}, (2\lambda - 1)\delta)$-DP, we take $\mathsf{p} \approx 2^{-42}$, which implies $M = 31$ and $\bar{M} = M + 2^{-37}$. The analysis is in Appendix C.5.

4.4.2 (Informal) Security analysis with 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 attribute size, a small subset of client can only secret-share a wrong report to be counted in another bucket. Since the aggregated results are already noisy, removing the report 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 C.3. 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 $2\mathsf{N}$.

Other than the above MPC protocol, more measures can be taken to bound user contributions, e.g. browsers only upload one record per time period; records must be aggregated then dumped per time period; reports contain random nonce under CCA secure encryption so they can't be duplicated.

4.4.3 (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 are hidden from each server as well as the reporting origin. We formally prove the privacy of random shuffling in Appendix C.1.

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

4.4.4 (Informal) Privacy analysis with a malicious server. Privacy against malicious server is formally analyzed in Appendix C.4 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 [7]: 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 C.5.

5 PERFORMANCE EVALUATION

We implement our protocol and show its performance with several experiments. First, we demonstrate the efficiency of generating both full-domain and subset histograms. Meanwhile, their performance will also be compared with prior works. Second, we run micro-benchmarks to demonstrate the bottlenecks of our implementation.

[8] Note that in practice we would need to subtract the shift parameter M from the bucket counts, but we omit it here for the sake of simplicity. For details, refer back to Section 3.1.

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 $(\mathcal{S}_2, \mathcal{S}_3)$.

5.1 Existing Proposals

The existing proposals for web conversion measurements include Prio [1, 14] and (i)DPF-based protocols [7, 8], which are further discussed in Appendix D.

We will compare our work with the DPF-based protocol for full-domain histograms and the iDPF-based protocol for subset histograms. Note that full-domain histograms get not benefit from iDPF over DPF.

The Prio system is still under rapid development and not ready for benchmarking [25]. For histogram generation, it encodes each attribute in a one-hot vector and generates the histogram from the vectors using generic MPC protocols. It has a larger communication overhead and higher number of round-trips of communication, which may result in a longer running time compared to our protocol. The main difference is that it does not provide formal differential privacy guarantees.

It is important to note that our protocol differs from the other two on the trust assumption. Namely, Bucketization assumes an honest-majority among three servers, while the others require only two non-colluding servers.

5.2 Constructing a Full Histogram

We benchmark the performance of our protocols for generating differentially private **full histograms** from **short reports**. The length of reports ℓ are chosen as $\ell \in \{16, 18, 20, 22, 24\}$. For each execution, $C = 10\,000\,000$ reports are synthesized and generated from a uniform distribution. The helper servers output a full histogram consisting of 2^ℓ buckets.

The one-time bucketization protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ is executed for $\ell \leq 22$ with $t = 0$ to deliver the full histogram. The experiments are implemented with $(\bar{\epsilon} = 0, \delta)$-DP, with parameters $\epsilon = 1$ for Laplace distribution and $M = 28$. We obtain $\mathrm{p} = 2^{-42}, \delta = 2^{-40}, \bar{M} = M + 2^{-37}$ for the single layer.

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. In this case we achieve $(\bar{\epsilon} = 0, (2\lambda - 1)\delta)$-DP, with parameters $\epsilon = 1$ for Laplace distribution and $M = 28$. We set $t = 0$ again and obtain $\mathrm{p} = 2^{-42}$, $(2\lambda - 1)\delta = 2^{-39}$ for $\lambda = 2$.

We compare the performance of our protocol with the DPF implementation in [24] that outputs a full histogram. This is a non-interactive protocol and the computational cost for the two helper servers are equal. The benchmark of [24] runs the DPF algorithm for one server and records the average time for processing one report. Multiplying this average time with $C = 10\,000\,000$ reports results in the total running time.

The results are shown in Table 3. For $\ell \in \{16, 18, 20, 22, 24\}$, our protocol is at least four orders of magnitude more efficient than the DPF-based protocol for generating **full histograms from short reports**.

ℓ	16	18	20	22	24
Bucketization (seconds)	9.2	11.2	25.3	75	422
DPF [7, 8, 24] (days)	1.15	4.75	19	76	305

Table 3: Performance of constructing full-domain histograms.

5.3 Micro-Benchmarks

We report the micro-benchmarks of our protocol in Table 4. As an interactive 3-party protocol, Bucketization is susceptible to the restrictions of bandwidth and is slightly affected by high round-trip latency (60 ms) due to its constant number of rounds of communication. As we see in Table 4, performance is mainly limited by the 3-party random shuffling protocol, which has a communication complexity of $O((C + B\bar{M})\ell)$. Revealing the noisy labels has a communication complexity of $O(C + B\bar{M})$.

In order to understand the effect of report sizes, we micro-benchmark the protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ with variable report length to demonstrate the performance of the three main components: adding dummy reports, 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 length of the reports is from a list $\ell \in \{32, 64, 256, 512\}$ with $\lambda = 1$, meaning that we only run one layer and bucketize on only 16-bit attribute. The servers generate differentially private histograms from a domain size of $B = 2^{16}$ on any 16 bits encoded as attributes in the reports. The parameters for $(\bar{\epsilon}, \delta)$-DP is $\bar{\epsilon} = 0$, and $\delta = 2^{-40}$ and $M = 28$. The results are shown in Table 4.

Report length ℓ (bits)	32	64	256	512
Add dummy reports	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: Micro-benchmark results for Bucketization.

5.4 Constructing a Subset-Histogram via Pruning

We conclude our experiments with benchmark of subset histogram aggregation. 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 [7], the reports are sampled from a certain distribution and are encoded sparsely in a large domain. The data collector is only interested in popular reports and tends 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 reports have a large domain size or the input reports have a highly concentrated distribution. We demonstrate the performance of our protocol Algorithm 1 with following experiments. Note that the

parameters for $(\bar{\epsilon}, \delta)$-DP are chosen according to the analysis of Appendix C.5, while we keep $(2\lambda - 1)\delta = 2^{-40}$.

We first synthesize a dataset of $C = 10\,000\,000$ input 32-bit reports from a domain size of 2^{16}, so the attributes are sparsely encoded in the reports. The reports follow a Zipf distribution, with Zipf parameter 1.03. We run Algorithm 1 with two-layer Bucketization ($\lambda = 2$) on higher and lower 16-bit attributes. In order to obtain $(\bar{\epsilon}, (2\lambda - 1)\delta) = (0, 2^{-40})$-DP, we set the standard Laplace parameter $\epsilon = 1$ and $M = 29$. It follows that $p = 2^{-44}$. We set the threshold to be $t = \bar{M} + \frac{C}{10\,000}$; for $\bar{M} = M + 2^{-39}$, $t = 1029$. **It takes 74 seconds for our protocol to generate the histogram with 971 output buckets.**

We also compare the performance of our protocol with the subset-histogram appearing in the end-to-end performance evaluation of [7]. In the experiment, $C = 400\,000$ input reports are sampled from a Zipf distribution, with Zipf parameter 1.03 and support 10 000. The bit-length of input reports is 256. The prune threshold is set to $t = \frac{C}{1000}$. The experiment of [7] is done between 2 servers with 32 virtual CPUs and utilizes a network with 61.9 ms round-trip latency. **Their protocol takes around 53 minutes to generate a subset-histogram.**

For our protocol, 3 helper servers run Algorithm 1 with $\lambda = 16$ and bucketize the reports on 16-bit attributes at each layer. **It takes 122 seconds to generate the subset-histogram with 1635 output buckets, when we keep the privacy parameters** $(\bar{\epsilon}, \delta) = (0, 2^{-40})$. In order to obtain $(\bar{\epsilon}, (2\lambda - 1)\delta) = (0, 2^{-40})$-DP, we set the standard Laplace parameter $\epsilon = 1$ and $M = 31$, with $p = 2^{-47}$.

6 CONCLUSIONS

We have presented an efficient 3-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 domains, as described in Algorithm 1.

In addition to other use-cases, we believe our approach presents a viable method for enabling web advertisers to obtain valuable information about their ad campaigns, while still preserving people's privacy with state-of-the-art cryptography and differential privacy. Our protocol is simpler than prior work and, due to its linear communication/computation complexity in the number of clients, it outperforms prior work in many practical scenarios, as demonstrated by our experiments.

REFERENCES

[1] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. *Cryptology ePrint Archive*, 2021.

[2] Apple and Google. Exposure Notification Privacy-preserving Analytics (ENPA) White Paper, apr 2021.

[3] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 535–548, 2013.

[4] Richard L. Barnes, Christopher Patton, and Phillipp Schoppmann. Verifiable Distributed Aggregation Functions, apr 2022.

[5] Muhammad Ahmad Bashir and Christo Wilson. Diffusion of user tracking data in the online advertising ecosystem. *Proc. Priv. Enhancing Technol.*, 2018(4):85–103, 2018.

[6] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 441–459, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776. IEEE, 2021.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.

[10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. Sepia: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10. USENIX Association, 2010.

[11] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 342–372. Springer, 2020.

[12] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[13] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 375–403. Springer, 2019.

[14] 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.

[15] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proceedings of the first ACM workshop on Smart energy grid security*, pages 75–80, 2013.

[16] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[17] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[18] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[19] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private Collection of Traffic Statistics for Anonymous Communication Networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14. Association for Computing Machinery, 2014.

[20] Steven Englehardt. Next steps in privacy-preserving telemetry with Prio, jun 2019.

[21] Tim Geoghegan, Christopher Patton, Eric Rescorla, and Christopher Wood. Privacy Preserving Measurement, mar 2022.

[22] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 640–658. Springer, 2014.

[23] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. 2019.

[24] Google. An Implementation of Incremental Distributed Point Functions in C++, feb 2022. commit 88c73a78cd61dacba6d8258f13d0f5dc5f1fb0d2.

[25] Divvi Up (ISRG). An Implementation of Prio in C++, apr 2022. commit 1bd54185d01110ff4486b8d71ed8f17f1ea77b78.

[26] Rob Jansen and Aaron Johnson. Safely Measuring Tor. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16. Association for Computing Machinery, 2016.

[27] Marek Jawurek and Florian Kerschbaum. Fault-tolerant privacy-preserving statistics. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 221–238. Springer, 2012.

[28] Ryo Kikuchi, Dai Ikarashi, Takahiro Matsuda, Koki Hamada, and Koji Chida. Efficient bit-decomposition and modulus-conversion protocols with an honest majority. In *Australasian Conference on Information Security and Privacy*, pages 64–82. Springer, 2018.

[29] Yi Lu, Keisuke Hara, Kazuma Ohara, Jacob Schuldt, and Keisuke Tanaka. Efficient Two-Party Exponentiation from Quotient Transfer. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, page 643–662. Springer-Verlag, 2022.

[30] 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*, pages 490–507, 2018.

[31] Meta. Make smarter business decisions with actionable insights, may 2022.

[32] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and psi for secret shared data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1271–1287, 2020.

[33] Andrés Muñoz Medina, Umar Syed, Sergei Vassilvtiskii, and Ellen Vitercik. Private optimization without constraint violations. In *International Conference on Artificial Intelligence and Statistics*, pages 2557–2565. PMLR, 2021.
[34] C Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 116–125, 2001.
[35] Web Platform Incubator Community Group. Attribution Reporting API with Aggregatable Reports, May 2022. commit fd75741c4e5c047de7536c02c20cbc903645aa17.
[36] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152, 2015.
[37] IETF working group on Privacy Preserving Measurement. Privacy Preserving Measurement Protocol, may 2022. commit 3aa0e86a4261cd749f5fa0b2569f5a44f482f042.
[38] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. FOCS'86, page 162–167. IEEE Computer Society, 1986.

A 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

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

The first 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.

B FURTHER DETAILS ON LAYERED BUCKETIZATION

B.1 Complexity of Algorithm 1

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

First, consider t too low, *i.e.*, $t \leq M$ so that the number of non-pruned buckets is exponential. In this case, the number of selected buckets at layer $i-1$ is upper-bounded by $L_1L_2 \ldots L_{i-1}$. The sum of the bucket sizes is $C + L_1 \ldots L_{i-1}\bar{M}$ 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.

Next, consider t large enough ($t > M$) to prune effectively. We consider every possible bucket $\mathcal{B}_1, \ldots, \mathcal{B}_{L_1L_2\ldots L_{i-1}}$ at layer $i-1$. Denote by a_j the number of true reports in bucket $\mathcal{B}_j$ and by $X_j = \lfloor M + Z_j \rceil$, the number of added dummy reports, where Z_j follows the truncated Laplace distribution. Finally, let Y_j be the number of dummy buckets 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 $O(\ell C + \bar{M}\frac{C}{t-M})$.

B.2 Reducing the Cost of Layered Bucketization

There are two potential problems with layered Bucketization: 1) the number of buckets can be exponential in the number of layers. This incurs an exponential number of round-trips of communication; 2) The amount of noise added to each layer is too large so it causes a memory blowup when bucketing multiple buckets in a batch. To address these issues, we make slight change to the layered Bucketization. The trick is to prune on each attribute independently before the layered Bucketization. This also provides the targeted output buckets of each layer so that the servers would know they do not need to add dummy reports to certain buckets because no real reports belong to those buckets. The steps are as follows:

(1) For a list of attributes indexed by $\{m_1, \ldots, m_\lambda\}$, the servers invoke $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ to bucketize on the attribute m_i for $i \in [\lambda]$. For each invocation of the protocol, they prune the output buckets with pre-determined threshold and records the attribute values associated with output buckets. At last they merge all buckets into a new dataset that will be used for the Bucketization of the next attribute.
(2) The servers invoke Algorithm 1 to perform layered Bucketization on attributes indexed by $\{m_1, \ldots, m_\lambda\}$. The operations are as described except that when adding the dummy reports, the servers only add dummy reports belonging

to the output buckets of current layer. The list of output buckets is recorded at the last step.

This method would maintain the constant round-trips of communication for the layered Bucketization. For the privacy, the new algorithm does not reveal more information to servers than Algorithm 1.

C ANALYSIS OF SECURITY AND PRIVACY

C.1 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. □

C.2 Differentially Private Protocol $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$

Theorem 2. *The $\Pi^{\mathsf{Hist}}_{\mathsf{Bucketize}}$ protocol given in Figure 10 is $(0, \delta)$-differentially private with $\delta = 2\mathsf{p}\frac{e^{\epsilon}-1}{1-\mathsf{p}}$ for a failure probability $\mathsf{p} = \frac{1}{2}e^{-\epsilon\left(\mathsf{M}+\frac{1}{2}\right)}$.*

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_+ and i_- be the two indices such that $d_{i_+} - d'_{i_+} = 1, d_{i_-} - d'_{i_-} = -1$, and $d_j = d'_j$ for all $j \in \{1, \cdots, L\}$ except $j \in \{i_+, i_-\}$. Let $\mathcal{M}(\cdot)$ be the function with noisy output such that $\mathcal{M} := \mathbb{N}^{\chi} \to \mathbb{R}^L$.

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

Let $S \subseteq \mathbb{R}^L$ 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 buckets i_+ and i_- are concerned). 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 two bucket indices i_+ and i_-. We start computing the probability of $\mathcal{M}(d)$ being in S_{good} for which the standard Laplace mechanism proof [18] works as it is:

$$\Pr(\mathcal{M}(d) \in S_{\mathsf{good}}) = \int_{s \in S_{\mathsf{good}}} \prod_j P(s_j - d_j) \tag{3}$$

$$= \int_{s \in S_{\mathsf{good}}} \prod_{j \in \{i_+, i_-\}} \frac{\mathsf{Lap}(s_i - d_i)}{1-\mathsf{p}} \prod_{j \notin \{i_+, i_-\}} P(s_j - d_j) \tag{4}$$

$$= \int_{s \in S_{\mathsf{good}}} \prod_{j \in \{i_+, i_-\}} \frac{\mathsf{Lap}(s_i - d'_i + d'_i - d_i)}{1-\mathsf{p}} \prod_{j \notin \{i_+, i_-\}} P(s_j - d'_j) \tag{5}$$

$$= \int_{s \in S_{\mathsf{good}}} \prod_{j \in \{i_+, i_-\}} \frac{\mathsf{Lap}(s_i - d'_i)}{1-\mathsf{p}} \prod_{j \notin \{i_+, i_-\}} P(s_j - d'_j) \tag{6}$$

$$= \int_{s \in S_{\mathsf{good}}} \prod_j \Pr(s_j - d'_j) \tag{7}$$

$$= \Pr(\mathcal{M}(d') \in S_{\mathsf{good}}) \tag{8}$$

$$\leq \Pr(\mathcal{M}(d') \in S) \tag{9}$$

where Equation (6) follows from

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

and the two $e^{\pm\epsilon}$ cancel.

Next, we compute the probability over S_{bad} which can be bounded as follows. Let the noise $n_i - \frac{1}{2} = s_i - d_i$. If $\mathcal{M}(d)$ is in S_{bad}, it means that either $n_{i_+} < -M$ (which is not possible due to resampling) or $n_{i_+} + d_{i_+} - d'_{i_+} < -M + \frac{1}{2}$ or similarly that $n_{i_-} < -M + \frac{1}{2}$. 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,

$$\Pr(\mathcal{M}(d) \in S_{\text{bad}}) \leq 2\Pr\left(n_i \in \left[-M-\frac{1}{2}, -M+\frac{1}{2}\right]\right) \tag{10}$$

$$= 2\int_{-M-\frac{1}{2}}^{-M+\frac{1}{2}} \frac{\text{Lap}(n_i, \frac{1}{\epsilon})}{1-\text{p}} \tag{11}$$

$$= 2\text{p}\frac{\text{e}^{\epsilon}-1}{1-\text{p}} \tag{12}$$

$$= \delta\,, \tag{13}$$

where (12) follows from

$$\int_{-M-\frac{1}{2}}^{-M+\frac{1}{2}} \text{Lap}\left(n_i, \frac{1}{\epsilon}\right) = \frac{1}{2}e^{-\epsilon\left(M+\frac{1}{2}\right)}(e^{\epsilon}-1)\,,$$

where $(1/2)\exp\left[-\epsilon\,(M+1/2)\right] = \text{p}$. Hence, we show that $\Pr(\mathcal{M}(d) \in S_{\text{bad}})$ is lower than the probability to sample a "bad" noise which is δ. $\Pr(\mathcal{M}(d) \in S_{\text{bad}}) \leq \delta$ for $\delta = 2\text{p}\frac{\text{e}^{\epsilon}-1}{1-\text{p}}$. Finally,

$$\Pr(\mathcal{M}(d) \in S) = \Pr(\mathcal{M}(d) \in S_{\text{good}}) + \Pr(\mathcal{M}(d) \in S_{\text{bad}}) \tag{14}$$

$$\leq \Pr(\mathcal{M}(d') \in S) + \delta \tag{15}$$

We conclude that the protocol $\Pi^{\text{Hist}}_{\text{Bucketize}}$ is $(\bar{\epsilon} = 0, \delta)$ differentially private for $\delta = 2\text{p}\frac{\text{e}^{\epsilon}-1}{1-\text{p}}$.

□

C.3 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* $\Pi^{\text{Hist}}_{\text{Bucketize}}$ *protocol without noise described in Figure 10 is bounded by* 2N.

Proof. We start with one malicious client. Let rec_{auth} be the true report of a malicious client. Let a (resp. b) be the number of reports in the correct (resp. incorrect) bucket that rec_{auth} belongs to. The consequence of the malicious behaviour is that true bucket will have $\text{a}-1$ reports while incorrect bucket will have $\text{b}+1$. 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 2N. □

C.4 Privacy Against a Malicious Server

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

$\text{Leak}_{\mathcal{A}}(\text{D})$

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

We let $\text{View}_{\mathcal{S}}(D)$ 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, $\text{Sim}(\text{Leak}_{\mathcal{A}}(D)) \sim \text{View}_{\mathcal{S}_1}(D)$.

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 reports $\mathcal{S}_2$ wants to add. $\mathcal{S}_1$ selects and outputs $n^{(1)}, \pi_{12}, \pi_{13}, R_{12}, R_{13}$. S_1 gets a vector A defined by $A = \pi_{23}(\pi_{12}((D-D^1)||D^2_{\text{dum}}) + R_{12}) + R_{23}$ 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 $(\text{coins}, D^1, n^{(2)}, A)$ 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 $\text{Out} = A + \pi_{13}^{-1}(B' - R_{13})$. There is a 1-to-1 correspondence between the final view $(\text{coins}, D^1, n^{(2)}, A, B')$ and $(\text{coins}, D^1, n^{(2)}, A, \text{Out})$. Thus, instead of B', we consider the equivalent Out.

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

We assume that D^2_{dum} is composed of the $n^{(2)}$ dummy reports created by $\mathcal{S}_2$ and by $n^{(1)}$ slots of 0 shares (to be added to the dummy reports 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, $\text{Out} = \pi_{23}(\pi_{12}(((D||D_{\text{dum}}) + \Delta)||\Delta_0))$ which is a random permutation of $(D||D_{\text{dum}}) + \Delta)||\Delta_0$.

We construct $\mathcal{A}$ who plays the $\text{Leak}_{\mathcal{A}}(\text{D})$ game by using $\mathcal{S}_1$ as follows:

$\mathcal{A}(\text{coins}, C, n^{(2)})$

1: Generate D^1 uniformly of size C.
2: Run $\mathcal{S}_1$ with input $(D^1, n^{(2)})$. Output $(n^{(1)}, \pi_{12}, \pi_{13}, R_{12}, R_{13})$
3: Generate A uniformly of size $(C + n^{(1)} + n^{(2)})$.
4: Continue to run $\mathcal{S}_1$ with input A and output A' and B.
5: Compute $\Delta' = \pi_{12}^{-1}(B + R_{12}) - D^1||0^{n^{(1)}+n^{(2)}}$.
6: Split $\Delta' = \Delta||\Delta_0$.
7: Output Δ.

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 $\text{Leak}_{\mathcal{A}}(D)$, 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:

$\mathsf{Sim}(\mathsf{coins}, C, n^{(2)}, \mathsf{hist})$

1: Redo the computations of $\mathcal{A}$ to get the same variables.
2: Create a vector V with histogram hist.
3: Set $V' = V||\Delta_0$.
4: Pick a random permutation π.
5: Set Out $= \pi(V')$.
6: Compute $B' = \pi_{13}(\text{Out} + A) + R_{13}$.
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 $\mathcal{S}_1$. □

C.4.1 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 reports 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 $D||D_{\mathsf{dum}}$.

C.4.2 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.

C.5 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 report 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 $2(2\lambda - 1)$.

In the proof of Theorem 2, what changes is that we have one e^{ϵ} appearing for each of the affected noise, but they eventually cancel. We also define S_{good} to hold for every affected noise. The consequence is that δ is multiplied by $(2\lambda - 1)$. Hence, the Layered protocol is $(0, (2\lambda - 1)\delta)$-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.

D EXISTING PROPOSALS

Even though there are many different proposals to solve secure aggregate problems, in this section, we focus on two specific proposals: Prio [14] and Distributed Point Functions (DPF) [7].

D.1 Prio

Prio [14] 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.

D.2 Distributed Point Functions

Recently, Google proposed an Attribute Reporting API with Aggregate Reports scheme, which strongly aligns with this problem [35]. A potential solution mentioned in their proposal relies on Distributed Point Functions (DPF): a two-party secure computation protocol [7, 22]. 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* [7]. Particularly, Boneh *et al.* [7] 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 $O(CB)$ 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).[9]

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 $O(\ell C^2/t)$ 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 $O(\ell C^2)$ 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 [7] 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.

[9] Note that the complexity of DPF.Eval is exponential in the size of the keys.

E DESCRIPTION OF FIGURES

We include some figures in this section to support the descriptions of the system design and protocols. The Figure 11 is a visualization of the layered bucketization process. The Figure 12 shows the data flows between three helper servers during the oblivious shuffling. The Figure 13 depicts the main subroutines of the bucketization. Specifically, how three helper servers add dummy reports to a secretly shared dataset and then randomly shuffle it.

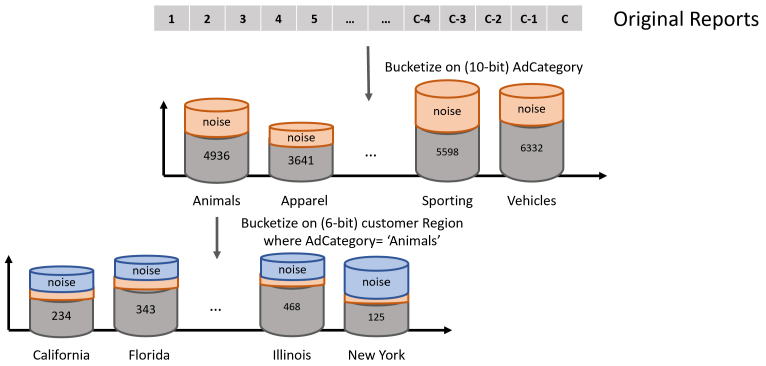

Figure 11: An example demonstrating the flexible functionality of our attribute aggregation protocol. When the computations run with Bucketization, the outputs will be differentially private histogram counts, *i.e.*, the size of each bucket plus a noise term.

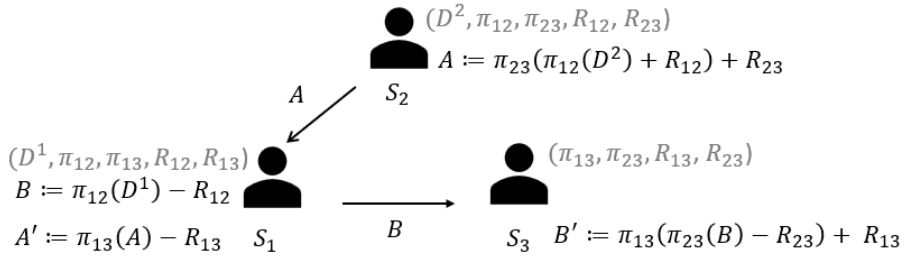

Figure 12: Illustration of Protocol Π_{RandShuf}. Gray tuples indicates the inputs known to each server.

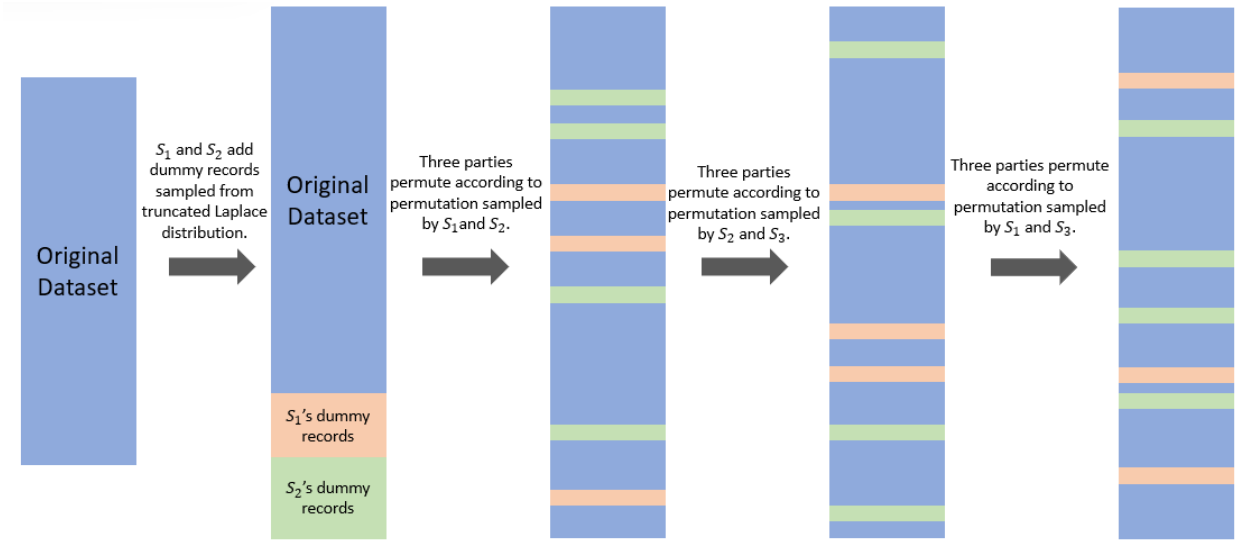

Figure 13: The workflow of sub-routines in Bucketization starting with dummy report addition and oblivious shuffling. It omits the secret sharing and the generation of output buckets. The dummy reports are colored to show the shuffling, due to the XOR of random masks, they will not be traced. This flow will run with three servers, which carry the shares of the original reports.