Proceedings on Privacy Enhancing Technologies ; 2022 (1):373–395

Badih Ghazi*, Ben Kreuter, Ravi Kumar, Pasin Manurangsi, Jiayu Peng, Evgeny Skvortsov, Yao Wang, and Craig Wright

Multiparty Reach and Frequency Histogram: Private, Secure, and Practical

Abstract: Consider the setting where multiple parties each hold a multiset of users and the task is to estimate the *reach* (i.e., the number of distinct users appearing across all parties) and the *frequency histogram* (i.e., fraction of users appearing a given number of times across all parties). In this work we introduce a new sketch for this task, based on an exponentially distributed counting Bloom filter. We combine this sketch with a communication-efficient multi-party protocol to solve the task in the multi-worker setting. Our protocol exhibits both differential privacy and security guarantees in the honest-but-curious model and in the presence of large subsets of colluding workers; furthermore, its reach and frequency histogram estimates have a provably small error. Finally, we show the practicality of the protocol by evaluating it on internet-scale audiences.

Keywords: reach, frequency histogram, sketching, differential privacy, multiparty computation

DOI 10.2478/popets-2022-0019
Received 2021-05-31; revised 2021-09-15; accepted 2021-09-16.

1 Introduction

A core problem in online advertising is to estimate the reach and frequency histogram of cross-publisher advertising campaigns. In this setting, an advertiser conducts a campaign across several publishers (defined as entities that host content and show advertisements on behalf of advertisers), with the potential of reaching overlapping sets of individuals (or households). The *reach* (aka cardinality) is the number of distinct users exposed to the campaign by at least one publisher. The *frequency histogram* is the fraction of users that are reached any particular number of times (e.g., once, twice, thrice...) by the campaign across all the publishers.[1]

Reach is an important metric for evaluating the efficacy of a campaign, and it is often used as the basis for billing agreements between advertisers and publishers. The frequency histogram is also crucial as it helps advertisers determine if individuals are being underexposed or overexposed to particular advertisements.[2] Advertisers are often interested in learning the frequency histogram in the range of $\{1, \ldots, 14, 15+\}$, where "15+" denotes a bucket containing the fraction of individuals exposed to the advertisement 15 or more times.

The desire to protect user privacy coupled with the lack of trust between the publishers (who usually compete for advertising revenue and are protective of their business intelligence) prevents them from sharing their raw reach data. Moreover, concerns about data breaches along with the absence of a trusted authority rule out any centralized solutions. This naturally leads to considering a distributed solution. The distributed setting we study consists of a set of publishers, a set of workers, and a single aggregator. Each publisher transmits information to one worker, and the workers and aggregators communicate amongst themselves. We augment this setting with formal *security* guarantees, which preclude eavesdroppers (along with a number of colluding publishers and workers) from learning substantial new information that is not supposed to be revealed by the output of the protocol. Moreover, the widespread concerns around user privacy lead us to seek a formal *privacy* guarantee, ensuring that the messages received by any entity during the protocol do not leak substantial new information about *any individual user*.

***Corresponding Author: Badih Ghazi:** Google Research, E-mail: badihghazi@gmail.com
Ben Kreuter: Google, E-mail: benkreuter@google.com
Ravi Kumar: Google Research, E-mail: ravi.k53@gmail.com
Pasin Manurangsi: Google Research, E-mail: pasin@google.com
Jiayu Peng: Google, E-mail: jiayupeng@google.com
Evgeny Skvortsov: Google, E-mail: evgenys@google.com
Yao Wang: Google, E-mail: wangyaopw@google.com
Craig Wright: Google, E-mail: wrightcw@google.com

1 We note that in the literature on differential privacy, frequency estimation/oracle is also used to refer to the task where given a universe element, the goal is to estimate the number of users holding it, see, e.g., [60].

2 www.sublime.xyz/en/blog/sublime-research-campaign-finds-overexposure-negative-impact

The security notion we consider is the *statistical indistinguishability* under computational assumptions, which is standard in cryptography in general, and in *secure multi-party computation* (SMPC) in particular [36]. Furthermore, we consider the multi-party honest-but-curious setting, where any proper subset of the workers could be colluding (allowing for a dishonest/corrupt majority). For privacy, we use *differential privacy* (DP) [28, 29], which has recently emerged as the gold standard for quantifying the privacy properties of algorithms, and has been deployed by government agencies [1] and tech [6, 26, 33, 41, 55]. For formal definitions, see Section 2.

The massive datasets typically generated in online advertising as well as the common desired goal of computing the reach and frequency histogram across multiple publishers and for several hundreds of thousands of campaign slices per day dictate the use of *scalable* algorithms. Specifically, we seek algorithms with small computational and communication footprints, and tractable dependence on the number of publishers. Accuracy is also critical, though we allow high probability bounds and approximate solutions, e.g., aiming for at least 95% of all estimates to be within 5% error.

In this work, we provide a novel sketching method for the cardinality and frequency histogram problems along with a distributed protocol for combining such sketches while satisfying all of the above properties, namely, security, privacy, scalability, and accuracy.

1.1 Technical Overview

Our solution combines Bloom filter-based [12] sketching with homomorphic encryption [2], SMPC [36], and DP [30].

It turns out that a conservatively-updated [34] counting Bloom filter (CBF) [58] could be used to accurately compute both reach and frequency histogram. Indeed, since the union operation only requires computing the entry-wise sum of the vector representing the data structure, an efficient multi-party protocol using homomorphic encryption could be used to estimate the frequency histogram, which is just the histogram of the CBF's counters, normalized. The main limitation of this approach is that a standard CBF requires space proportional to the number of items to be counted, which prevents it from scaling to internet-size audience estimation. This limitation led us to the development of a method for allocating items to registers exponentially, which allows audiences to be measured in space logarithmic in their size. Unfortunately, distributing the allocation this way causes a large number of collisions in the registers with the largest mass, which means that a naive estimation of the frequency histogram from the histogram of counters will result in a distribution that is incorrectly left-skewed. For this reason, we develop the *Same Key Aggregator* (SKA), which can be viewed as a cryptographic implementation of the *not equal* function, and which further emphasizes the need to sparsely represent the CBF when running the MPC protocol.

As part of the protocol execution we add appropriately calibrated noise to ensure our protocol is DP. In fact, not only is the final output private, but so are all intermediates, even if all but a single party collude. Achieving this presents us with multiple challenges. This is because the view of each party in each step of the protocol is distinct, requiring us to add multiple types of noise to hide different types of revealed information. Such noise must be crafted to make the protocol both private and accurate; e.g., some noise is added in an earlier step but must be filtered before the reach and frequency histogram estimation stages for accuracy.

1.2 Related Work and Alternatives

HyperLogLog. Computing reach for a single party or publisher is equivalent to the ubiquitous cardinality estimation problem. Numerous methods have been invented to solve this problem [42], and HyperLogLog [37] and its variants [44] are the de facto standard, with implementations of these algorithms available in many leading database and data warehousing solutions such as Redis, Google BigQuery, and Amazon Redshift. The multi-party extension of cardinality estimation could be easily dealt with if each party shared its HLL with all other parties or with some central authority; however it is known that HLLs are subject to leaking the presence of individual records and have the potential to leak an unbounded amount of negative information (i.e., records that are not in the set) [25]. Given these privacy concerns, new approaches to estimating multi-party cardinality are needed, for which ads measurement is just one of many possible use cases that include healthcare [62] and WiFi analytics [3]. Frequency, while being economically useful to the advertising ecosystem, also poses new and interesting technical challenges.

BLIP. Before landing on the approach outlined above, we explored several alternatives. One of these was BLoom-and-flIP (BLIP), which is a DP mechanism for Bloom filters that entails randomly flipping each bit

of the Bloom filter with some probability [4]. Unfortunately, this technique did not scale beyond a small handful of parties, and in particular we found that this number was inversely proportional to the probability of flipping each bit, which means that for a DP parameter ϵ of $\ln(3)$, which has a corresponding flip probability of 0.25, approximately 4 Bloom filters can be unioned before the error explodes. We also found that for a fixed Bloom filter size, the standard error of the cardinality estimate was sensitive to the size of the set being estimated. Frequency estimation posed additional problems. Specifically, defining a privacy mechanism for a CBF requires that the sensitivity be defined based upon the number of impressions associated with the user who has the largest number of impressions, and since we are dealing with counts not bits, the analogue to bit flipping is to perturb the count, for which we used the geometric mechanism [39]. The result of this, even in the single party case, and despite attempts to correct for the noise, is a highly inaccurate estimate.

Private Set Union. A related method for securely unioning Bloom filters and deriving a cardinality estimate was given by [31]. Their method starts with each input party splitting its Bloom filter into shares and transmitting them, one share to each node. Then each node combines the shares it receives, resulting in a combined share. Next, the nodes agree upon a permutation and each permutes its combined share. Finally, each node sends its permuted combined share to another independent node that does not know the permutation, and which combines the permuted shares, thereby giving a permuted Bloom filter. The resulting permuted Bloom filter is used to compute a cardinality estimate. This protocol has several shortcomings, the first being that the topology of the protocol could be simplified while simultaneously keeping the permutation hidden using techniques for distributed permutations presented herein. A second problem is that the output is not DP. A third issue is that the protocol relies on uniform Bloom filters and will therefore not scale to internet-sized audience estimation, nor is it adapted to computing frequency. We contend that either one of these last two issues could be easily addressed in the context of that protocol, but that the left-skew problem associated with the frequency estimation discussed above makes simultaneously addressing these problems quite difficult.

We have also considered combining HLLs via MPC, but it is unclear how to obtain a suitable method. The one devised by [62] for combining HLLs does not produce a DP result and also relies on shared secrets, which do not meet our bar for privacy and security respectively. We note that HLLs have been previously adapted for frequency estimation at Google using a similar technique to the one described herein [56].

Generic MPC. While generic SMPC results (e.g., [36]) could be applied to computing reach and frequency histogram, the resulting protocol would incur sizeable polynomial blow-ups, and thus would not scale. Specialized protocols for computing set intersection have been previously reported in real-world applications, for example the two-party protocol used by Google [45] which is similar to the protocol we present below. An important difference in our setting is the requirement that the outputs be DP, which is not met by that protocol.

In addition to its prevalence in database management systems, cardinality estimation has been studied in sketching, streaming, and communication complexity (e.g., [5, 8, 10, 15, 21, 23, 27, 35, 38, 47, 61] and the references therein).

In the DP literature, the Element Distinctness problem has been well-studied, see, e.g., [7, 18, 20, 25, 49, 53]. Moreover, estimating the frequency histogram is related to the anonymized histogram problem [43, 57].

Multi-Party DP. In the distributed setting, there has been some work at the intersection of privacy and security, e.g., [9]. A widely studied distributed model of privacy is *local* DP [48]. A drawback of this model is the large error that has to be incurred even for very simple tasks such as binary summation, where the error is known to grow asymptotically with the square root of the number of users [9, 16]. In addition to the shuffle setting [11, 19, 32, 46], other distributed models can be combined with DP including Secure Aggregation [13] and PRIO [24]; like ours, the latter operates in the multi-worker setting. As far as we know, a DP protocol computing reach and frequency histogram for internet-scale audiences is known neither in the Secure Aggregation model nor in the multi-worker setting.

Count(-Min) Sketch. The count sketch [17] and count-min sketch [22], while useful for determining the items in a set with the highest frequencies (aka heavy-hitters), are not well-suited for the frequency histogram problem. However, these structures do have a striking resemblance to CBFs.

1.3 Organization

We start by providing some background material in Section 2. Our sketching technique is given in Section 3, and our MPC protocol is in Section 4. We state its security guarantee in Section 5 and its privacy properties in Sec-

tion 6. We discuss its efficiency in Section 7 and perform an empirical evaluation in Section 8. We conclude with some future directions in Section 9. Additional details on the privacy proof are in Appendix A.

2 Preliminaries

For any positive integer k, let $[k]$ denote the set $\{1, \ldots, k\}$. Let $\mathbb{R}_{\geq 0}$ (respectively, $\mathbb{R}_{>0}$) be the set of all non-negative (respectively, positive) reals. Similarly, $\mathbb{Z}_{\geq 0}$ denotes the set of all non-negative integers.

We next recall the definition and basic properties of DP. Datasets $\mathbf{X}$, $\mathbf{X}'$ are said to be *neighboring* if $\mathbf{X}'$ results from adding or removing[3] a single user from $\mathbf{X}$.

Definition 1 (Differential Privacy (DP) [28, 29])**.** *Let $\epsilon, \delta \in \mathbb{R}_{\geq 0}$. A randomized algorithm* Alg *taking as input a dataset is said to be* (ϵ, δ)-differentially private *if for any two neighboring datasets $\mathbf{X}$ and $\mathbf{X}'$, and for any subset S of outputs of* Alg*, it holds that* $\Pr[\textsc{Alg}(\mathbf{X}) \in S] \leq e^{\epsilon} \cdot \Pr[\textsc{Alg}(\mathbf{X}') \in S] + \delta$.

For an extensive introduction to DP, we refer the reader to [30, 59]. Since we will combine the security of MPC protocols with privacy, we require a definition of *computational* DP [50]. Below we state a simplified version from [30, Definition 9.4].

Definition 2 (Computational DP)**.** *Let $\epsilon, \delta \in \mathbb{R}_{\geq 0}$ and $n \in \mathbb{N}$. A randomized algorithm* Alg *taking as input a dataset is said to be* (ϵ, δ)-computational DP *if for any two neighboring datasets $\mathbf{X}$ and $\mathbf{X}'$, and for any polynomial (in $|\mathbf{X}|$) time algorithm* Dist*, it holds that* $\Pr[\textsc{Dist}(\textsc{Alg}(\mathbf{X})) = 1] \leq e^{\epsilon} \cdot \Pr[\textsc{Dist}(\textsc{Alg}(\mathbf{X}')) = 1] + \delta + \mathrm{negl}(|\mathbf{X}|)$*, where $|\mathbf{X}|$ denotes the size of input dataset $\mathbf{X}$ and* $\mathrm{negl}(\cdot)$ *is a function that grows slower than the inverse of any polynomial.*

Table 1 contains notation that we use in the paper. Each term will be described in detail when it is introduced.

Notation	Meaning
$n; \hat{n}$	cardinality; estimate of n
$f_{\max}$	maximum frequency level of interest
$r_i; \hat{r}_i$	ith entry of the frequency histogram, i.e., cardinality at frequency level i divided by n; estimate of r_i
$\mathcal{L}$	LiquidLegions sketch
m	total number of registers in a LiquidLegions sketch
a	decay rate of the LiquidLegions sketch
p	number of publishers
w	number of non-aggregator MPC workers
T	number of non-colluding MPC nodes
υ	noise for cardinality estimation
η	noise for frequency estimation
κ	noise for blinded histogram
λ	publisher noise
χ	noise for publisher noise added by protocol
(ϵ_*, δ_*)	privacy parameters for noise type $*$
μ_*	mean of the noise type $*$

Table 1. Notation used in the paper. To highlight that the noise comes from a certain party, we write it in the superscript; e.g., $\eta^{\mathsf{worker}(j)}$ denotes the frequency estimation noise from worker j.

3 LiquidLegions Sketch

In this section we describe *LiquidLegions*, an exponentially distributed variant of a counting Bloom filter (CBF) sketch, which we use to estimate reach and frequency at Internet scale. LiquidLegions is parameterized by a decay rate a and is an array of m registers, each of which is a $\langle$count, key$\rangle$ pair as described below.

Basic Sketch Operations. Items are assigned to registers according to a *truncated exponential distribution* (tExp) with parameter a, which has a probability density function (PDF) of

$$f(x; a) = \frac{ae^{-ax}}{1 - e^{-a}}, \; x \in (0, 1). \tag{1}$$

To assign an item to a register, we first split the interval $[0, 1]$ into m equal segments, where the ith segment corresponds to the ith register, and sample a real number from tExp and assign the item to the register corresponding to the interval in which the number fell. Thus an item is assigned to the ith register with probability

$$p_i = \frac{a}{m(1 - e^{-a})} \exp\left(-\frac{ai}{m}\right). \tag{2}$$

Note that sampling from tExp can be done by sampling a number in $[0, 1]$ uniformly (i.e., by hashing the item to be inserted by applying a function $\textsc{Hash}(\cdot)$, assumed henceforth in the analysis to be a source of uniform shared randomness), and then using the inverse CDF of

3 We note that while we define neighboring datasets in terms of *addition/removal* of elements, our results immediately imply bounds for the *substitution* notion of DP (where two datasets are considered neighboring if they can be obtained by *changing* a single element) albeit with a factor of 2 increase in ϵ and δ.

tExp, given by:

$$F^{-1}(u;a) = 1 - \frac{\log(e^a + u(1-e^a))}{a}.$$

AssignRegister in Algorithm 1 describes this.

Algorithm 1 Basic LiquidLegions Operations.

procedure InitializeSketch
Input: number m of registers
Output: empty sketch $\mathcal{L}$
  $\mathcal{L} \leftarrow$ array of $\langle 0, \texttt{null}\rangle$ of size m
procedure AssignRegister
Input: item x to assign, number m of registers
Output: LiquidLegions register index
  $f \leftarrow \text{Hash}(x)$
  $u \leftarrow (f \mod 2^{64})/2^{64}$
  $z \leftarrow 1 - \log(\exp(a) + u * (1 - \exp(a)))/a$
  return $\lfloor z * m \rfloor + 1$
procedure InsertItem
Input: item x, sketch $\mathcal{L}$
Output: updated sketch $\mathcal{L}$
  $j \leftarrow \text{AssignRegister}(x)$
  $k \leftarrow \text{Hash}(x)$
  $\mathcal{L}_j.\text{count} \leftarrow \mathcal{L}_j.\text{count} + 1$
  if $\mathcal{L}_j.\text{key} = \texttt{null}$ **or** $\mathcal{L}_j.\text{key} = k$ **then**
    $\mathcal{L}_j.\text{key} \leftarrow k$
  else
    $\mathcal{L}_j.\text{key} \leftarrow \texttt{destroyed}$
procedure MergeSketch
Input: sketches $\mathcal{L}, \mathcal{L}'$ with same number of registers
Output: merged sketch $\mathcal{L}$
  for $j = 1, \ldots, m$ **do**
    $\mathcal{L}_j.\text{count} \leftarrow \mathcal{L}_j.\text{count} + \mathcal{L}'_j.\text{count}$
    if $\mathcal{L}_j.\text{key} \neq \texttt{null}$ **and** $\mathcal{L}'_j.\text{key} \neq \texttt{null}$ **and** $\mathcal{L}_j.\text{key} \neq \mathcal{L}'_j.\text{key}$ **then**
      $\mathcal{L}_j.\text{key} \leftarrow \texttt{destroyed}$
    else if $\mathcal{L}_j.\text{key} = \texttt{null}$ **then**
      $\mathcal{L}_j.\text{key} \leftarrow \mathcal{L}'_j.\text{key}$

We next describe how an item is inserted into the sketch (InsertItem in Algorithm 1). Initially the key is null, and this holds until an item is assigned to the register, at which time a fingerprint (chosen as the Hash(·) digest in our protocol) of the assigned item is stored as the key. Then as a subsequent item is assigned to the register, the count is incremented by 1. Meanwhile, the key of the register and the fingerprint of the newly assigned item are compared. If they disagree, then the key of the register is marked with a sentinel value, destroyed, indicating that the register has been destroyed; see Figure 1 for an example. An undestroyed register is termed *active*. A destroyed register's count will not be used for estimating frequency, however a destroyed register will still contribute to the count of non-zero registers, which is required for estimating cardinality.

index	0	1	2	...	$m-1$
count	0	0	0	...	0
key	∅	∅	∅	...	∅

(a) A LiquidLegions sketch is an exponentially distributed CBFs that tracks register collisions using the fingerprint of the first item to be assigned to that register. It is represented as an array of (count, key) pairs. For brevity, ∅ denotes null.

index	0	1	2	...	$m-1$
count	1	0	1	...	0
key	k_1	∅	k_2	...	∅

(b) A LiquidLegions sketch with two items inserted. When an item is inserted into the sketch it is assigned to a register according to the AssignRegister procedure in Algorithm 1. In the case where the register is empty its count is set to 1 and its key is set to that of the item's fingerprint.

index	0	1	2	...	$m-1$
count	2	0	1	...	0
key	k_1	∅	k_2	...	∅

(c) A third item is inserted and assigned to register 0. Since its fingerprint is the same as that of the register key, the key is unchanged. The count is incremented.

index	0	1	2	...	$m-1$
count	2	0	2	...	0
key	k_1	∅	⊥	...	∅

(d) A fourth item is inserted and assigned to register 2. Its fingerprint is different than that of the register's key and therefore the key is replaced with destroyed (for brevity, denoted ⊥) indicating that the register is destroyed. The count is still incremented for convenience.

Fig. 1. Examples of basic operations or LiquidLegions.

Merging two sketches, which is lossless, is done register-wise, by summing the counts and ensuring the register is marked destroyed if the keys differ and are non-null; see MergeSketch in Algorithm 1.

Estimation Using LiquidLegions. Cardinality estimation is based on all the non-empty registers in the sketch, i.e., registers that contain at least one item. Note that each item is assigned to the ith register ($0 \le i \le m-1$) with probability $p_i = (1/m)f(i/m;a)$, where f is the pdf of tExp (1). Then, a mapping from the cardinality n to the expected fraction of non-empty registers can be obtained as follows:

$$\begin{aligned}
\mathcal{E}(n) &= \frac{1}{m}\sum_{i=0}^{m-1}\mathbb{P}(\text{the } i\text{th register is non-empty}) \\
&= \frac{1}{m}\sum_{i=0}^{m-1}(1-(1-p_i)^n) \approx 1 - \frac{1}{m}\sum_{i=0}^{m-1}\exp(-np_i) \\
&= 1 - \frac{1}{m}\sum_{i=0}^{m-1}\exp\left(-f\left(\frac{i}{m};a\right)\cdot\frac{n}{m}\right) \\
&\approx 1 - \int_0^1 \exp\left(-f\left(x;a\right)\frac{n}{m}\right)\mathrm{d}x = 1 - \int_0^1 \exp\left(-\frac{ae^{-ax}}{1-e^{-a}}\frac{n}{m}\right)\mathrm{d}x.
\end{aligned}$$

We then arrive at

$$\mathcal{E}(n) \approx 1 - \frac{1}{a}\left(\mathrm{Ei}\left(-\frac{an}{(1-e^{-a})m}\right) - \mathrm{Ei}\left(-\frac{ane^{-a}}{(1-e^{-a})m}\right)\right), \quad (3)$$

where $\mathrm{Ei}(t) = \int_{-\infty}^{t}(e^u/u)\mathrm{d}u$ is the exponential integral (Algorithm 2, EstimateCardinality). As $\mathcal{E}(\cdot)$ is monotone, a binary search can find its root.

Algorithm 2 Estimation with LiquidLegions.

procedure EstimateCardinality
Input: LiquidLegions sketch $\mathcal{L}$
Output: Estimated cardinality $\hat{n}$ of the set stored in $\mathcal{L}$
  $x \leftarrow \#\{j \mid \mathcal{L}_j.\mathtt{key} \neq \mathtt{null}\}$
  $\hat{n} \leftarrow$ root of $\mathcal{E}(n) = x/m$, as defined in (3)
  return $\hat{n}$
procedure EstimateFrequency
Input: LiquidLegions sketch $\mathcal{L}$
Output: Estimates $\hat{r}_1, \ldots, \hat{r}_{f_{\max}}$ such that $\sum_{i\in[f_{\max}]} \hat{r}_i = 1$
  $H \leftarrow [0, \ldots, 0] \in \mathbb{Z}^{f_{\max}}$
  for $j = 1, \ldots, m$ **do**
    if $\mathcal{L}_j.\mathtt{key} \neq \mathtt{null}$ **and** $\mathcal{L}_j.\mathtt{key} \neq \mathtt{destroyed}$ **then**
      $i \leftarrow \min(\mathcal{L}_j.\mathtt{count}, f_{\max})$
      $H[i] \leftarrow H[i] + 1$
  $S \leftarrow \sum_{i=1}^{f_{\max}} H[i]$
  for $i = 1, \ldots, f_{\max}$ **do**
    $\hat{r}_i \leftarrow H[i]/S$
  return $\hat{r}_1, \ldots, \hat{r}_{f_{\max}}$

Frequency estimation is based on all the active registers in the sketch, i.e., registers that contain exactly one item. Assuming Hash$(\cdot)$ is truly random, the register index of each item is *independent* of its frequency. Therefore, the items that fall in the active registers form an unbiased sample of all the items. The frequency histogram of these active registers is thus an unbiased estimate of the frequency histogram of all the items. See EstimateFrequency in Algorithm 2.

Theorems 1 and 2 below describe the accuracy of the proposed cardinality and frequency estimators; the proofs are in Appendices B and C respectively.

Theorem 1. *Fix $n/m = z$ to be any positive ratio, and let $m \to \infty$. Then,*

$$\frac{\mathbb{E}[\hat{n}] - n}{n} \to 0 \quad \textit{and} \quad \frac{m}{n^2} \cdot \mathrm{var}(\hat{n}) \to f(a, z),$$

where for any $a > 0$,

$$f(a,z) = \frac{a\left(\mathrm{Ei}(-c) - \mathrm{Ei}(-2c) - \mathrm{Ei}(-e^{-a}c) + \mathrm{Ei}(-2e^{-a}c)\right)}{(\exp(-e^{-a}c) - \exp(-c))^2} - z^{-1}, \tag{4}$$

with

$$c = \frac{az}{1 - e^{-a}}, \tag{5}$$

and

$$f(0,z) = \lim_{a\to 0+} f(a,z) = \frac{e^z - 1}{z^2} - \frac{1}{z}.$$

Theorem 2. *Fix $n/m = z$ as any positive ratio, fix each i, and let $m \to \infty$. Then,*

(a) $\mathbb{E}[\hat{r}_i] = r_i$.

(b) Let A be the number of active registers and let

$$\gamma = \frac{\exp(-e^{-a}c) - \exp(-c)}{a},$$

with c defined in (5). Then

$$\frac{\mathbb{E}[A]}{m} \to \gamma \quad \textit{and} \quad m \cdot \mathrm{var}(\hat{r}_i) \to \frac{z - \gamma}{z\gamma} \cdot r_i(1 - r_i).$$

LiquidLegions for Internet Scale. We next discuss the parameter choices for LiquidLegions to work at internet-scale. Let m be large enough, say, $m > 1000$. For cardinality estimation, from Theorem 1, the estimate has a relative standard deviation[4] of

$$\mathrm{rstd}(\hat{n} \mid a, m, n) \approx \sqrt{\frac{f(a, n/m)}{m}}. \tag{6}$$

From (6) we can approximately derive

$$m_{\min}(a, n, \alpha) := \min\{m \mid \mathrm{rstd}(\hat{n} \mid a, m, n) \leq \alpha\},$$

for any a, n, and a threshold α on the relative standard deviation. Figure 2 shows how $m_{\min}$ depends on a and $\log_{10}(n)$, for $\alpha = 2.5\%$. The upper bound of internet population in all different countries is about 10^9, so the plot is up to $\log_{10}(n) = 9$. When $a = 12$, $m_{\min}$ is within 10^5. For $a \leq 8$, however, $m_{\min}$ grows exponentially from some point, and exceeds 10^5 (in fact, exceeds even 10^6) when $\log_{10}(n) = 9$—such sketch length is computationally infeasible! We thus conclude that

(i) LiquidLegions with $a \geq 12$ can scale to internet-size audience estimation. (In addition, since the curve for $a = 16$ is above that for $a = 12$ for most n, $a = 12$ is more favorable than $a = 16$.)

(ii) LiquidLegions with $a \leq 8$, in particular, the uniform Bloom Filter ($a = 0$) cannot scale.

The above theoretical conclusions have been empirically validated; see Table 2. The empirical relative biases[5] are close to 0 as expected, and the empirical standard deviations align well with theoretical standard deviations. As can be seen from Table 2(b), for a uniform Bloom filter with $m = 10^5$ registers, the estimate quickly becomes more dispersed when $n > 5 \cdot 10^5$. And when $n > 10^6$, one has $> 1\%$ chance of encountering a saturated Bloom filter and thus failing to estimate the cardinality. On the other hand, a LiquidLegions sketch with $a = 12$ gives an estimate of $\sim 1\%$ rstd, for $n \leq 10^9$.

4 Standard deviation of the estimate divided by the truth.
5 Mean of (estimate − truth)/truth.

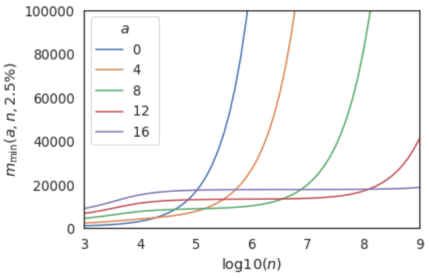


Fig. 2. The minimum m to achieve $\alpha = 2.5\%$ relative standard deviation of cardinality estimation, for different a and $\log_{10}(n)$.

true cardinality	empirical rel. bias	empirical rel. std	theoretical rel. std
10^2	-0.00009	0.00561	0.00549
10^3	-0.00004	0.00587	0.00555
10^4	-0.00018	0.00615	0.00620
10^5	0.00003	0.00839	0.00855
10^6	-0.00009	0.00953	0.00907
10^7	0.00069	0.00960	0.00913
10^8	-0.00001	0.00938	0.00931
10^9	-0.00033	0.01142	0.01132

(a) $a = 12$.

true cardinality	empirical rel. bias	empirical rel. std	theoretical rel. std
10	-0.00005	0.00316	0.00224
10^3	0.00018	0.00214	0.00224
10^5	0.00000	0.00282	0.00268
$5 \cdot 10^5$	0.00009	0.00746	0.00755
$8 \cdot 10^5$	0.00251	0.02159	0.02155
$9 \cdot 10^5$	0.00608	0.03450	0.03161
When $n \geq 10^6$, saturation occurs with $> 1\%$ probability, and the empirical bias and std are no longer measurable.			

(b) $a = 0$, i.e., uniform Bloom filter.

Table 2. Empirical results for LiquidLegions with $m = 1e5$; empirical bias and std obtained from 1000 replicates.

At this point, we stress that LiquidLegions is unsuitable for set membership, for which the standard Bloom filter is used. Indeed, because of the fairly large decay rate a we use, our technique encourages collisions in order to reduce the sketch size; thus, in that regime LiquidLegions would perform poorly for set membership.

For frequency estimation, Theorem 2 says $\hat{r}_i$ approximately has standard deviation[6]

$$\mathrm{std}(\hat{r}_i \mid a, m, n, r_i) = \sqrt{\frac{z-\gamma}{mz\gamma} \cdot r_i(1-r_i)}.$$

This attains the maximum when $r_i = 0.5$, so given a, m, n, $\mathrm{std}(\hat{r}_i \mid a, m, n, r_i)$ has a supremum of

$$\sup \mathrm{std}(a, m, n) = 0.5\sqrt{\frac{z-\gamma}{mz\gamma}}.$$

As for cardinality, we define

$$m^{\mathrm{freq}}_{\min}(a, n, \alpha_{\mathrm{freq}}) := \min\{m \mid \sup \mathrm{std}(a, m, n) \leq \alpha_{\mathrm{freq}}\}.$$

For $\alpha_{\mathrm{freq}} = 1\%$, the dependence of $m^{\mathrm{freq}}_{\min}(a, n, \alpha_{\mathrm{freq}})$ on a and $\log_{10}(a)$ is shown in Figure 3. (Note that α_{freq} is not necessarily equal to α since they are about std and rstd respectively.) It is similar to Figure 2, in spite of slight differences in the magnitude. We reach the same conclusion as before about LiquidLegions's scalability.

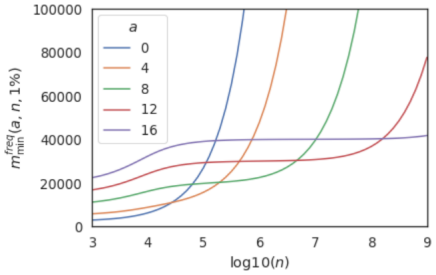


Fig. 3. The minimum m to achieve $\alpha_{\mathrm{freq}} = 1\%$ standard deviation of frequency estimation, for different a and $\log_{10}(n)$.

4 MPC Protocol

The protocol described below first extracts a differentially private, henceforth "private", count of distinct LiquidLegions register IDs (i.e., the index of the registers) in the union of several sketches. This count is then used as the input to LiquidLegions, which gives an estimate of the cardinality of the cross-publisher union. We also extract a private frequency histogram in the range $[1, f_{\max}+]$, where the last bucket of the histogram has a count of values $\geq f_{\max}$. Throughout, noise is injected to ensure that all intermediate outputs are private.

There are three types of entities involved in the protocol: p publishers, w workers, and a single aggregator. The term *node* is used to refer to either a worker or the aggregator. The total communication and computation overheads grow linearly with the number w of workers. For any particular execution of the protocol, the number p of publishers is fixed. Each publisher sends its sketch to exactly one of the workers at the beginning of the computation (Figure 4). We next introduce a few primitives before describing the protocol.

6 Since $\sum_{i=1}^{f_{\max}} r_i = 1$, we consider std instead of rstd.

4.1 Basic Primitives

Our protocol requires four core cryptographic ingredients, which are supplied by the Elliptic Curve (EC) ElGamal cryptosystem and the Pohlig–Hellman cipher [54]. The first property is an N-of-N threshold variant of EC ElGamal that is constructed by multiplying any number of public keys together. The second property, which is also supplied by the EC ElGamal cryptosystem, is additive homomorphism, which allows the counts of the input sketches to be summed. A consequence of this property is that a ciperhtext can be rerandomized by adding a new encryption of zero to it. We assume the existence of a function called ReRandomize that applies this operation to a ciphertext. The next primitive is supplied by the Pohlig–Hellman cipher, which commutes with EC ElGamal, and allows for the creation of a distributed pseudorandom function. Finally we require a source of cryptographic random numbers, for which we assume the existence of a function UnifInt that returns a random number of the same size as that of the elliptic curve points we use. Throughout we also use Enc and Dec to refer to encryption and decryption operations respectively.

In addition to these core primitives we define an additional function called the *same-key aggregator* (SKA). The purpose of this function is to ensure that the key of a register, to which multiple publishers contributed, is identical. Or in other words, it is a cryptographic mechanism for determining whether a register is destroyed, and ensures that only non-destroyed registers are used for computing the frequency histogram. To achieve this, we track a flag that when decrypted reveals only whether the count is valid.

We begin with encrypted pairs $\langle C_1, K_1\rangle, \langle C_2, K_2\rangle, \ldots$ of count and key that we wish to combine. This list can be thought of as the values for a single register, where each tuple is the value contributed by a specific publisher. Algorithm 3 shows how to construct the encrypted sum of the counts and an encrypted flag indicating whether all of the keys are identical. The value of the flag is a well-known constant `same_key`.

To ensure a private output, the protocol generates noise registers using the shifted and truncated geometric mechanism, where each node contributes an equal fraction of the noise distribution. This is achieved by having each worker generate noise that is distributed according to a difference of two truncated Polya random variables (Algorithm 4). The noise distribution is selected so that if T non-colluding parties added the noise, then we achieve (ϵ, δ)-DP if the measured quantity has sensitivity Λ (see Appendix A.1). We will invoke the noise sampling multiple times in the protocol with varying sensitivities depending on the quantity we are noising, e.g., the noise for the frequency histogram (i.e., η) has Λ set to 2 since adding/removing a single user can change 2 entries of the frequency histogram. Note that the output noise has mean μ and lies in $[0, 2\mu]$.

Algorithm 3 Same-key aggregator.

procedure SKA
Input: a list $\langle C_1, K_1\rangle, \ldots, \langle C_\ell, K_\ell\rangle$ of pairs of encrypted count and encrypted key; `pk` is an ElGamal public key
Output: if all the keys are the same, then an encryption s of `same_key` and encryption C' of the sum of the counts, where `same_key` is a well-known constant; otherwise, an encryption of two random numbers
  $s \leftarrow \text{Enc}(\texttt{same\_key}, \texttt{pk})$
  $C' \leftarrow 0$
  for $i = 1, \ldots, \ell$ **do**
    $s \leftarrow s + \text{UnifInt}() * (K_i - K_1)$
    $C' \leftarrow C' + C_i$
  $C' \leftarrow C' + \text{UnifInt}() * (s + \text{Enc}(-\texttt{same\_key}, \texttt{pk}))$
  return (s, C')

Algorithm 4 Noise sampling.

procedure SampleNoise
Input: privacy parameters (ϵ, δ), sensitivity Λ, number T of non-colluding nodes
Output: a random variable that is the difference of two truncated Polya random variables shifted by μ
  $r \leftarrow 1/T$
  $s \leftarrow \epsilon/\Lambda$
  $\mu \leftarrow \lceil \ln(2T\Lambda(1 + e^{\epsilon})/\delta)/s \rceil$
  do ▷ Sample X_1 and X_2 until both are $\leq \mu$
    $X_1 \leftarrow \text{Polya}(r, e^{-s})$; $X_2 \leftarrow \text{Polya}(r, e^{-s})$
  while $X_1 > \mu$ or $X_2 > \mu$
  $X = X_1 - X_2$
  return $\mu + X$

4.2 Protocol Description

The operation of the protocol is divided into five phases: *Creation*, *Setup*, and three execution phases: *Aggregation*, *ReachEstimation*, and *FreqEstimation*. The details of each of these phases are quite complicated and hence we first describe the flow of the protocol at a high level while intentionally ignoring privacy (Figure 4).

In the *Creation* phase we assume that each publisher has already created a sketch that represents the audience to be reported on. From here, the publisher transforms each non-zero register into a three-tuple, which

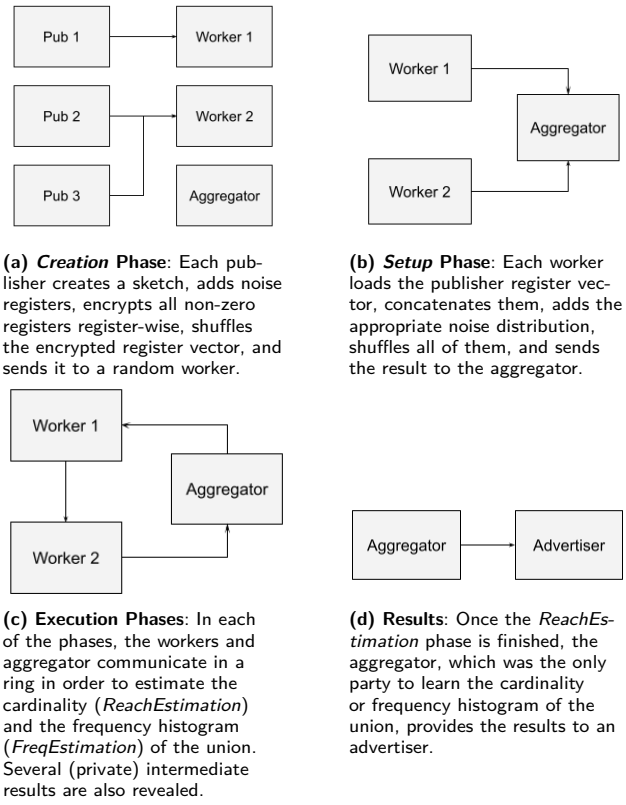
consists of the register ID, the count, and the key, or if the register has been destroyed, the key is replaced with `destroyed` indicating this. Each element of each tuple is then encrypted with the EC ElGamal cipher and transmitted to a random MPC worker.

After each publisher has transmitted its sketch to exactly one worker, the protocol commences. The first phase of the protocol is *Setup*, wherein each worker loads the arrays of three-tuples that were sent to it and concatenates them. This array is then shuffled tuple-wise and sent to the designated aggregator.

From here the workers and the aggregator communicate in a ring topology to remove the ElGamal encryption from the register IDs, while applying a Pohlig–Hellman key, which results in all of the register IDs being deterministically encrypted. Once this has happened, the three-tuples are joined on the register ID by the aggregator. Then the joined counters are simply summed, while the keys are combined using Algorithm 3; this is the *Aggregation* phase. From here it is possible to estimate reach by counting the number of distinct register IDs and applying the LiquidLegions estimator (EstimateCardinality in Algorithm 2) to this count. Note that due to the intricacies of maintaining privacy, the version of the protocol described below provides a reach estimate only after the second round is completed; this is the *ReachEstimation* phase.

After these phases, we are left with a set of two-tuples, where the first element of the tuple is a SKA ciphertext and the second is a count. The next round of communication decrypts the SKA ciphertext. All of tuples that are determined to be destroyed are discarded, leaving an array of counters for the undestroyed registers. Finally, in the third round of communication, the counters are decrypted and used to estimate the frequency histogram; this is the *FreqEstimation* phase.

The following sections describe all of this is increased detail and add several mechanisms for ensuring that all intermediate and final outputs are private.



(a) *Creation* Phase: Each publisher creates a sketch, adds noise registers, encrypts all non-zero registers register-wise, shuffles the encrypted register vector, and sends it to a random worker.

(b) *Setup* Phase: Each worker loads the publisher register vector, concatenates them, adds the appropriate noise distribution, shuffles all of them, and sends the result to the aggregator.

(c) Execution Phases: In each of the phases, the workers and aggregator communicate in a ring in order to estimate the cardinality (*ReachEstimation*) and the frequency histogram (*FreqEstimation*) of the union. Several (private) intermediate results are also revealed.

(d) Results: Once the *ReachEstimation* phase is finished, the aggregator, which was the only party to learn the cardinality or frequency histogram of the union, provides the results to an advertiser.

Fig. 4. The MPC protocol.

4.2.1 *Creation* Phase

In this phase, each publisher begins by creating a LiquidLegions sketch (Section 3). It then retrieves the public ElGamal key from each worker and combines these into a single "full public key". Next, the publisher runs the PrepareSketch procedure in Algorithm 5, which begins by generating a three-tuple for each non-empty register. Each active register is represented by a three-tuple of register ID, count, and key. Each destroyed register is represented by a three-tuple of the register ID, count, and `key_destroyed`, where `key_destroyed` is a well-known constant that will subsequently ensure that these registers are not used for frequency histogram estimation. After the register vector has been created, tuples of the form $\langle \texttt{reg\_pub\_noise}, 0, \textsc{UnifInt}() \rangle$ are appended, where `reg_pub_noise` is a well-known constant (these are *fake* registers added for privacy purposes). Next, the register vector is randomly shuffled, each element of each tuple is encrypted with the combined ElGamal key, and is then sent to a random worker.

4.2.2 *Setup* Phase

This phase begins when all publisher register vectors have been received by one of the workers. Each worker loads and concatenates the register vectors before appending four different noise types (see Table 3 and SetupWorker procedure in Algorithm 6).

Each of the four noise types is associated with a particular form of register tuple and each is intended to ensure either an intermediate result or the final output is private. The first type of noise applies to

Noise	Added by	Added in (phase)	Removed in (phase)	Purpose
λ	Publisher	*Creation*	*ReachEstimation*	Noise cardinality of publisher's # of non-empty registers
κ	Worker & Aggregator	*Setup*	*ReachEstimation*	Noise the blind histogram
χ				Noise publisher's noises
υ				Noise cardinality of non-empty registers (affecting reach estimate)
η	Worker & Aggregator	*ReachEstimation*	-	Noise frequency histogram (affecting frequency histogram estimate)
$\hat{\eta}$			*FreqEstimation*	Noise frequency histogram (with "flags" as in AggregationAggregator)

Table 3. In addition to the noise terms listed in this table, the *Setup* and *ReachEstimation* phases feature "padding noise" that ensures that the number of noise registers output by each party is fixed (in order to avoid leaking information through this count).

Algorithm 5 *Creation* Phase.

procedure EncTuple
Inputs: register ID r, count c, key k, public key pk
Output: encrypted three-tuple of the inputs
  return $\langle \text{Enc}(r, \text{pk}), \text{Enc}(c, \text{pk}), \text{Enc}(k, \text{pk}) \rangle$

procedure PrepareSketch
Inputs: LiquidLegions $\mathcal{L}$, combined ElGamal public key pk
Output: a register vector of three-tuples, where each element is encrypted using pk
  $rv \leftarrow [\,]$
  for $j = 1, \ldots, m$ **do**
    $c \leftarrow \mathcal{L}_j$.count
    $k \leftarrow \mathcal{L}_j$.key
    if $c = 0$ **then continue**
    if $k =$ `destroyed` **then**
      $k \leftarrow$ `key_destroyed`
    $r \leftarrow \text{Hash}(j)$
    rv.append(EncTuple(r, c, k, pk))
  $\lambda \leftarrow \text{SampleNoise}(\epsilon_\lambda, \delta_\lambda, p, 1)$
  for $i = 1, \ldots, \lambda$ **do**
    $r \leftarrow \text{Hash}(\texttt{reg\_pub\_noise})$
    rv.append(EncTuple$(r, 0, \text{UnifInt}(), \text{pk})$)
  rv.shuffle() ▷ Randomly permute
  return rv

the count of distinct register IDs and has the form $\langle *, *, \texttt{key\_destroyed} \rangle$. This noise, along with all other destroyed registers, will be filtered before estimating the frequency histogram.

The second type of noise applies to the blinded histogram, $h \in \mathbb{Z}^p_{\geq 0}$, where h_a is the number of register IDs that are non-empty in a of the publisher sketches, which is revealed in the *Aggregation* phase. These noise registers take on the form $\langle *, *, \texttt{key\_bh\_noise} \rangle$ and will be filtered in the *ReachEstimation* phase before calculating the count of distinct register IDs.

The third type of noise register is intended to "hide" the publisher noise that was added in the *Creation* phase and takes the form $\langle \texttt{reg\_pub\_noise}, *, * \rangle$. This ensures that individual publisher register counts cannot be determined when the publisher noise is filtered at the end of the *Aggregation* phase.

The final type of noise is padding for the rest of the noise, and takes on the form $\langle \texttt{reg\_pad\_noise}, *, * \rangle$. A number of noise registers equal to a well-known constant, B minus the number of previously added noise registers is added, thus ensuring that the worker adds exactly B to its register vector. This noise is also filtered, and does not impact any estimates.

Once all noise registers are appended to the register vector, it is shuffled and sent to the aggregator. After the aggregator receives the register vectors from all the workers, it adds its own noise registers as above and shuffles, forming the *combined register vector* (CRV); see SetupAggregator procedure in Algorithm 6.

4.2.3 *Aggregation* Phase

This phase starts when the aggregator passes the CRV to the first worker. Then for each register ID, the worker uses its secret key, sk_{eg}, to remove its ElGamal encryption before applying a newly generated Pohlig–Hellman key, sk_{ec}. The combined effect of this, once all workers have performed these two operations, is to apply a distributed pseudorandom function to the register IDs, allowing them to be later joined by the aggregator.

After operating on the register IDs, the worker then re-randomizes all of the keys and counts. The CRV is then shuffled and passed to the next worker, which repeats the same process (see AggregationWorker procedure in Algorithm 7). Once all of the workers have processed the CRV, the aggregator removes its ElGamal key from the register IDs, which then allows the register IDs to be joined. The result of the join is the noisy blinded histogram of register IDs.

Algorithm 6 *Setup* Phase.

procedure SETUPWORKER
Inputs: concatenated publisher register vectors rv, full public key pk
Output: modified rv
▷ **Step 1:** noise for distinct count of register IDs
  $\upsilon \leftarrow \text{SAMPLENOISE}(\epsilon_\upsilon, \delta_\upsilon, 1, T)$
  for $i = 1, \ldots, \upsilon$ **do**
    $r \leftarrow \text{UNIFINT}() > m$
    rv.append(ENCTUPLE(r, 0,key_destroyed, pk))
▷ **Step 2:** blinded histogram noise
  $N \leftarrow 0$
  for $k = 1, \ldots, p$ **do**
    $\kappa_k \leftarrow \text{SAMPLENOISE}(\epsilon_\kappa, \delta_\kappa, 2, T)$
    $N \leftarrow N + k \cdot \kappa_k$
    for $i = 1, \ldots, \kappa_k$ **do**
      $r \leftarrow \text{UNIFINT}() > m$
      for $j = 1, \ldots, k$ **do**
        rv.append(ENCTUPLE(r, 0, key_bh_noise, pk))
▷ **Step 3:** noise for publisher noise
  $\chi \leftarrow \text{SAMPLENOISE}(\epsilon_\chi, \delta_\chi, p, T)$
  for $i = 1, \ldots, \chi$ **do**
    rv.append(ENCTUPLE(
      reg_pub_noise, UNIFINT(), UNIFINT(), pk))
▷ **Step 4:** padding noise
  for $i = N + \upsilon + \chi + 1, \ldots, B$ **do**
    rv.append(ENCTUPLE(
      reg_pad_noise, UNIFINT(), UNIFINT(), pk))
▷ **Step 5:** randomly permute
  rv.shuffle()
procedure SETUPAGGREGATOR
Input: concatenated worker register vectors, crv
Output: modified crv
▷ **Step 1:** add noise and shuffle
  $crv \leftarrow \text{SETUPWORKER}(crv)$

The aggregator then runs the SKA on the counts and keys associated with each register ID yielding an encrypted count and an encrypted flag that is zero if all keys are the same; we denote this $flag_1$. Two other flags, $flag_2$ and $flag_3$, which indicate whether all keys for the register are equal to key_destroyed or key_bh_noise respectively (Table 4), are also computed. Finally, the register IDs are discarded, leaving a set of four-tuples of the form $\langle count, flag_1, flag_2, flag_3 \rangle$; see AGGREGATIONAGGREGATOR in Algorithm 7.

4.2.4 *ReachEstimation* Phase

The purpose of this phase is two-fold. First, a private estimation of the cardinality of the union of publisher sketches is achieved by decrypting the flags, and second, noise is added to ensure that the frequency histogram

Algorithm 7 *Aggregation* Phase.

procedure AGGREGATIONWORKER
Input: combined register vector crv, where each element of the tuple is encrypted
Output: modified crv
  for $\langle reg, count, key \rangle \in crv$ **do**
    $reg \leftarrow \text{ENC}(\text{DEC}(reg, \text{sk}_{\text{eg}}), \text{sk}_{\text{ec}})$
    $count \leftarrow \text{RERANDOMIZE}(count)$
    $key \leftarrow \text{RERANDOMIZE}(key)$
  crv.shuffle()
procedure AGGREGATIONAGGREGATOR
Input: combined register vector crv
Output: a list of $\langle count, flag_1, flag_2, flag_3 \rangle$
  for $(reg, \_, \_) \in crv$ **do**
    $reg \leftarrow \text{DEC}(reg, \text{sk}_{\text{eg}})$
  ▷ Join on the register IDs of crv to obtain a map of
  ▷ register ID to a list of count, key pairs
  $map = crv$.joinOnRegID()
  $L \leftarrow [\cdot]$
  for $(\_, pairs) \in map$ **do**
    $(count, flag_1) \leftarrow \text{SKA}(pairs)$
    $flag_2 \leftarrow \text{UNIFINT}() *$
      $(pairs[1].\text{key} - \text{key\_destroyed} + flag_1)$
    $flag_3 \leftarrow \text{UNIFINT}() *$
      $(pairs[1].\text{key} - \text{key\_bh\_noise} + flag_1)$
    L.append($\langle count, flag_1, flag_2, flag_3 \rangle$)
  return L

revealed in the *FreqEstimation* phase is private. This noise has the form $\langle f, 0, *, * \rangle$ for each $f \leq f_{\max}$. Next, noise of the form $\langle *, *, *, * \rangle$ is added in order to hide the count of destroyed registers. Finally, similar to the *Setup* phase, we pad the above noise to a well-known value, D for each worker, with registers of the form $\langle *, 0, 0, * \rangle$.

The phase begins with the aggregator, which adds the requisite noise before passing the shuffled set of four tuples to the first worker. Then once each worker has added the noise tuples and has decrypted the flags, the aggregator regains control and decrypts the set of flags, revealing their values (see REACHESTIMATIONWORKER

$flag_1$	$flag_2$	$flag_3$	semantics
0	R	R	register is not destroyed and not blinded histogram noise
0	R	0	register is blinded histogram noise
0	0	R	destroyed by all publishers or Step 1 noise added in Algorithm 6
R	R	R	destroyed in join

Table 4. Meaning of the various valid combinations of flag values; the remaining combinations are impossible. "R" indicates a random number. If all keys were the same, then $flag_1$ is zero. If all keys are equal to key_destroyed, then $flag_2$ is zero. Finally, if all keys are equal to key_bh_noise, then $flag_3$ is zero.

and ReachEstimationAggregator procedures in Algorithm 8).

The input to the LiquidLegions cardinality estimator is determined by counting the number of non-zero $flag_3$ values and subtracting from this the value $w*D-2$. Thus the blinded histogram noise is removed by dropping registers with $flag_3 = 0$. The publisher and padding noise from the *Setup* phase are removed by observing that by this time they contribute to just a single register each, and the frequency histogram noise is removed by observing that each worker contributed exactly D noise registers during this phase. The input to the next phase, which consists only of the non-destroyed counts, is determined by keeping only those counts that have $flag_1 = 0$ and $flag_2 \neq 0$ and $flag_3 \neq 0$.

4.2.5 *FreqEstimation* Phase

In this phase, the private frequency histogram of the combined publisher sketches is revealed. This begins with the aggregator creating an SKA-matrix of counters, where the column indices of the matrix correspond to the counter indices and the row indices correspond to the range $[f_{\max}-1]$. Each cell of the matrix is populated by a same-key aggregator, where the key is equal to the row index, and upon decryption will indicate whether the counter has the value denoted by the row index. Note that there will be at most one cell per column that is zero, thus indicating the value of the counter.

Columns with no zero value are put in the $f_{\max}$ bin (Table 5). Once constructed by the aggregator the SKA-matrix is circulated to each worker and decrypted, after which the aggregator finally removes its decryption, revealing the frequency histogram; see Algorithm 9 for more details.

The primary outputs of our protocol are the noisy reach and the noisy frequency histogram. Additionally, the outputs also include certain intermediate values revealed during the different phases, which have to be taken into account for security and privacy proofs. More details are in Lemma 2 and Lemma 3.

We note that the protocol described above can be used to generate an estimate of any real or categorical distribution, e.g., the total duration the audience was exposed to an advertisement or its demographics. This is done by introducing additional "counters" and using an additional SKA-matrix to reveal the distribution.

Algorithm 8 *ReachEstimation* Phase.

procedure AddFrequencyNoise
Inputs: list L of tuples of the form $\langle count, flag_1, flag_2, flag_3\rangle$, ElGamal public key pk_{eg} for encrypting tuples
Output: modified L
  ▷ items in below tuples are encrypted with pk_{eg}
  $\eta_{\text{total}} \leftarrow 0$
  for $f = 1, \ldots, f_{\max}$ **do**
    $\eta_f \leftarrow \text{SampleNoise}(\epsilon_\eta, \delta_\eta, 2, T)$
    $\eta_{\text{total}} \leftarrow \eta_{\text{total}} + \eta_f$
    for $i = 1, \ldots, \eta_f$ **do**
      L.append($\langle f, 0, \text{UnifInt}(), \text{UnifInt}()\rangle$)
  $\hat{\eta} \leftarrow \text{SampleNoise}(\epsilon_\eta, \delta_\eta, 2, T)$
  $\eta_{\text{total}} \leftarrow \eta_{\text{total}} + \hat{\eta}$
  for $i = 1, \ldots, \hat{\eta}$ **do**
    L.append(
      $\langle \text{UnifInt}(), \text{UnifInt}(), \text{UnifInt}(), \text{UnifInt}()\rangle$)
  for $i = \eta_{\text{total}} + 1, \ldots, D$ **do**
    L.append($\langle \text{UnifInt}(), 0, 0, \text{UnifInt}()\rangle$)
  L.shuffle()

procedure ReachEstimationWorker$_j$
Input: list L of tuples of the form $\langle count, flag_1, flag_2, flag_3\rangle$
Output: modified L
  for $\langle \_, flag_1, flag_2, flag_3\rangle \in L$ **do**
    $\text{Dec}(flag_1, \text{sk}_{\text{eg}})$
    $\text{Dec}(flag_2, \text{sk}_{\text{eg}})$
    $\text{Dec}(flag_3, \text{sk}_{\text{eg}})$
  ▷ pk is combined key of downstream nodes
  AddFrequencyNoise(L, pk)

procedure ReachEstimationAggregator
Input: list L of tuples of the form $\langle count, flag_1, flag_2, flag_3\rangle$
Outputs: unique register count, list of undestroyed counts
  AddFrequencyNoise(L, pk_{full})
  yield to worker$_1$
  ▷ control returns after last worker sends L
  $B \leftarrow 0$
  active $= [\cdot]$
  for $(count, flag_1, flag_2, flag_3) \in L$ **do**
    $flag_1 \leftarrow \text{Dec}(flag_1, \text{sk}_{\text{eg}})$
    $flag_2 \leftarrow \text{Dec}(flag_2, \text{sk}_{\text{eg}})$
    $flag_3 \leftarrow \text{Dec}(flag_3, \text{sk}_{\text{eg}})$
    if $flag_3 \neq 0$ **then**
      $B \leftarrow B + 1$
    if $flag_1 = 0$ and $flag_2 \neq 0$ and $flag_3 \neq 0$ **then**
      active.append($count$)
  ▷ X is the input to the LiquidLegions estimator
  $X \leftarrow B - w * D - 2$
  return (X, active)

	Enc(1)	Enc(1)	Enc(2)	Enc(3)
f_1	$\eta(1,1)$	$\eta(1,1)$	$\eta(1,2)$	$\eta(1,3)$
f_2	$\eta(2,1)$	$\eta(2,1)$	$\eta(2,2)$	$\eta(2,3)$

Table 5. Example of a same-key aggregator matrix for four active registers with counts of $\{1,1,2,3\}$ and $f_{\max}=3$, where $\eta(i,j) = R\cdot(\textsc{Enc}(i,\mathrm{pk}) - \textsc{Enc}(j,\mathrm{pk}))$ and R is a distinct random number. Each column has at most one zero value which determines the frequency of that counter, while columns with no zero values are assigned to $f_{\max}$.

Algorithm 9 *FreqEstimation* Phase.

procedure FreqEstimationWorker$_j$
Input: SKA-matrix M_s
Output: decrypts entries of M_s using $\mathrm{sk}_{\mathrm{eg}}$
  for $x \in M_s$ **do**
    $x \leftarrow \textsc{Dec}(x, \mathrm{sk}_{\mathrm{eg}})$
procedure FreqEstimationAggregator
Input: list L_a of undestroyed (i.e., active) counters
Output: decrypted SKA-matrix M
  $M \leftarrow [\cdot,\cdot]$
  col $\leftarrow 0$
  for $c \in L_a$ **do**
    for $f = 1,\ldots,f_{\max}-1$ **do**
      $M[\mathrm{col}, f] \leftarrow (c - \textsc{Enc}(f,\mathrm{pk})) * \textsc{UnifInt}()$
    col $\leftarrow$ col $+ 1$
  yield M to worker$_1$
  ▷ control returns here after all workers have processed M
  FreqEstimationWorker$_{\mathrm{agg}}(M)$

5 Security Property

Definition 3 (Honest-but-Curious Security)**.** *Let $f = (f_1,\ldots,f_W)$ be a function over W inputs $\mathbf{x} = (x_1,\ldots,x_W)$, where each f_i is an output for party i and let π be a W-party protocol for computing f. The protocol π is* secure in the honest-but-curious model with static corruptions *if for any subset $\mathcal{I}$ consisting of $T \le W$ parties there exists a simulator $\mathcal{S}_{\mathcal{I}}$ such that $\{(\mathcal{S}_{\mathcal{I}}(1^k,\{x_i\}_{i\in\mathcal{I}},\{f_i(\mathbf{x})\}_{i\in\mathcal{I}}), f(\mathbf{x}))\}_{\mathbf{x},k} \approx \{(\mathtt{view}^{\pi}_{\mathcal{I}}(\mathbf{x},k), \mathtt{output}^{\pi}(\mathbf{x},k))\}_{\mathbf{x},k}$.*

By the *view* we mean the inputs, randomness, and transcript of the messages received by all parties in $\mathcal{I}$, i.e., $\mathtt{view}_i = (x_i, r_i, \mathtt{transcript}_i)$. Moreover, $\mathtt{output} = \{\mathtt{output}_i\}_{1\le i\le W}$ is the result computed by each party following an execution of π. Note that in the honest-but-curious model the parties are assumed to output a specific function, and that security is defined in terms of the joint distribution of outputs of *all* parties. The simulator $\mathcal{S}_{\mathcal{I}}$ must produce $\{\mathtt{transcript}_i\}_{i\in\mathcal{I}}$ given only the inputs and outputs of each party in $\mathcal{I}$.

Theorem 3. *Under the Decisional Diffie–Hellman Assumption [14], the protocol described in Section 4 (Algorithms 6-9) satisfies Definition 3.*

A proof sketch of Theorem 3 is in Appendix D

6 Privacy Guarantee

The privacy guarantee of our protocol is encapsulated in the following theorem.

Theorem 4. *Let $\epsilon_\lambda, \delta_\lambda, \epsilon_\upsilon, \delta_\upsilon, \epsilon_\kappa, \delta_\kappa, \epsilon_\eta, \delta_\eta, \epsilon_\chi, \delta_\chi$ be as in Algorithms 5, 6, and 8. The view of any set of all but T nodes (i.e., workers or aggregator) in the protocol is $(\epsilon_\lambda+\epsilon_\upsilon+\epsilon_\kappa+\epsilon_\eta+\epsilon_\chi, \delta_\lambda+\delta_\upsilon+\delta_\kappa+\delta_\eta+\delta_\chi)$-computational DP.*

The proof of the above theorem is discussed in more detail in Appendix A.

Expected Total Number of Noise Registers. The numbers of noise registers, which contribute to the protocol communication overhead, are as follows:

- **Publisher Noise.** Each publisher adds μ_λ noise registers to their sketch in expectation.
- ***Setup* Phase.** Each worker/aggregator sends exactly B noise registers. The smallest possible B we can set so that the protocol is well-defined is $2\cdot\mu_\chi + 2\cdot\mu_\upsilon + \mu_\kappa\cdot p\cdot(p+1)$; we will set B to be this number to minimize the number of noise registers.
- ***ReachEstimation* Phase.** Each worker/aggregator sends exactly D noise registers. The smallest possible D we can set so that the protocol is well-defined is $2\cdot\mu_\eta\cdot(f_{\max}+1)$; we will set D to be this number to minimize the number of noise registers.

In total, the expected number of noise registers added during the entire protocol is

$$\mu_\lambda\cdot p + (w+1)\cdot(2\mu_\chi + 2\mu_\upsilon + \mu_\kappa\cdot p\cdot(p+1) + 2\mu_\eta\cdot(f_{\max}+1)). \tag{7}$$

Variance of the Noise in Reach Estimation. For parameters a, b, let $\mathrm{Polya}_{a,b}$ denote the Polya distribution and let $\mathrm{tPolyaDiff}_{a,b}$ denote the difference of two truncated Polya distributions (see Definition 6). The

number of non-empty registers used in reach estimation is the true number of non-empty registers plus the noise $\upsilon^{\text{aggregator}} + \left(\sum_{j\in[w]} \upsilon^{\text{worker}(j)}\right)$. When setting the parameters in our protocol, this noise has variance

$$\begin{aligned}\sigma_\upsilon^2 &:= (w+1)\cdot \text{var}(\text{tPolyaDiff}_{\mu_\upsilon, 1/T, e^{-\epsilon_\upsilon}}) \\ &\le (w+1)\cdot 2 \cdot \text{var}(\text{Polya}_{1/T, e^{-\epsilon_\upsilon}}) \\ &= (w+1)\cdot \frac{2e^{-\epsilon_\upsilon}}{T(1-e^{-\epsilon_\upsilon})^2}.\end{aligned} \tag{8}$$

Note that ϵ_υ is a parameter that can be set based on the ϵ of the entire protocol (see the example below).

Variance of the Noise in Frequency Estimation. The frequency histogram is noised by $\eta^{\text{aggregator}} + \sum_{j\in[w]} \eta^{\text{worker}(j)}$. (Note that this noise is a $f_{\max}$-dimensional vector.) When setting the parameters in our protocol each entry of this noise vector has variance

$$\begin{aligned}\sigma_\eta^2 &:= (w+1)\cdot \text{var}(\text{tPolyaDiff}_{\mu_\eta, 1/T, e^{-\epsilon_\eta/2}}) \\ &\le (w+1)\cdot 2 \cdot \text{var}(\text{Polya}_{1/T, e^{-\epsilon_\eta/2}}) \\ &= (w+1)\cdot \frac{e^{-\epsilon_\eta/2}}{T(1-e^{-\epsilon_\eta/2})}.\end{aligned} \tag{9}$$

Similar to above, ϵ_η can be based on the total ϵ.

Example. Suppose that we have $\epsilon = \ln(3), \delta = 10^{-9}$, the number of workers (not including the aggregator) is $w = 2$, the number of publishers is $p = 3$ and $f_{\max} = 5$. Moreover, suppose that the adversary compromises only one of the workers or aggregator (i.e., there are $T = 2$ uncorrupted parties). While there are several ways to split the privacy budget among $\epsilon_\upsilon, \epsilon_\eta, \epsilon_\lambda, \epsilon_\kappa, \epsilon_\chi$, recall that only the first two ϵ's affect the accuracy of the protocol. As such, to maximize utility, we should spend as little budget as possible among the latter three ϵ's. In this example, we only allocate 30% of the total ϵ to the latter three. Specifically, we pick $\epsilon_\upsilon = \epsilon_\eta = 0.35\ln(3)$ and $\epsilon_\chi = \epsilon_\kappa = \epsilon_\lambda = \ln(3)/10$.

For δ, how we divide it among $\delta_\upsilon, \delta_\lambda, \delta_\kappa, \delta_\eta, \delta_\chi$ does not make a big difference since the dependency is only $\ln(\delta_*)$ and the total δ is already quite small to begin with. Thus, we only split it equally in this example for simplicity, i.e., $\delta_\upsilon = \delta_\lambda = \delta_\kappa = \delta_\eta = \delta_\chi = 0.2 \cdot 10^{-9}$. Our protocol, specifically the SampleNoise procedure (Algorithm 4), will select the following parameters: $\mu_\upsilon = 65, \mu_\eta = 132, \mu_\kappa = 459, \mu_\lambda = 680, \mu_\chi = 699$. Plugging this back into (7) implies that the expected total number of noise registers added is $\le 18K$, which is an acceptable overhead for, e.g., sketch size $m = 100K$. Note that the number of noise registers can be reduced further by allocating more privacy budget to $\epsilon_\chi, \epsilon_\kappa, \epsilon_\lambda$, but will result in worse accuracy (as indicated by (9), (8)).

7 Efficiency

We briefly discuss the efficiency of our protocol; we will focus on the total number of cryptographic operations (i.e., encryptions and decryptions) performed by each party, which is also an upper bound, e.g., on the communication complexity. For brevity, we will refer to this as the *complexity* of that party. Furthermore, we assume that $\epsilon \le O(1)$ and δ is sufficiently small (e.g., $1/\delta > 10w/p$) to simplify the bounds. First, observe that each publisher constructs at most m non-noise encrypted registers, and at most $2\mu_\lambda$ registers for publisher noise. Thus, its complexity is at most $O(m + \mu_\lambda) \le O\left(m + p \cdot \log(1/\delta_\lambda)/\epsilon_\lambda\right)$.

Next, we consider each worker or the aggregator. In the *Setup* phase, it processes a total of at most $O((m+\mu_\lambda)p + B)$ registers (where B is as defined in the previous section). In the *Aggregation* phase, each worker/aggregator has to rerandomize $O((m+\mu_\lambda)\cdot p + w\cdot B)$ registers. In the *ReachEstimation* phase, it processes at most $O(m + w\cdot B + w\cdot\mu_\eta\cdot f_{\max})$ tuples, where the first term comes from the fact that after joining there can be at most m non-noise tuples left, the second term comes from the fact that joining in the previous step does not increase the number of tuples, and the last comes from the frequency noise. In the *FreqEstimation* phase, the matrix M consists of $O(m \cdot f_{\max})$ entries, which also upper bounds the complexity of the worker. In total, each worker/aggregator's complexity is at most

$$\begin{aligned}&O((m+\mu_\lambda)\cdot p + m\cdot f_{\max} \\ &\quad + w\cdot(\mu_\chi + \mu_\upsilon + \mu_\kappa\cdot p^2 + \mu_\eta \cdot f_{\max})).\end{aligned}$$

To simplify the expression above, suppose that we set $\delta_\lambda, \delta_\chi, \delta_\upsilon, \delta_\kappa, \delta_\eta \ge \Omega(\delta)$ and $\epsilon_\lambda, \epsilon_\chi, \epsilon_\upsilon, \epsilon_\kappa, \epsilon_\eta \ge \Omega(\epsilon)$. Recall also that it suffices to consider $f_{\max} \le p$. Then, the complexity is upper bounded by

$$O\left(m\cdot p + \frac{\log(1/\delta)}{\epsilon}\cdot wp^2\right).$$

In other words, when the number of publishers is sufficiently small, each worker/aggregator's complexity grows linearly with the number of publishers; but once it is sufficiently large, it grows quadratically. However, as we observe in the experiments in the next section, for reasonable values of p (e.g., ≤ 20) and m (e.g., 10^5), we are mostly in the former case.

Phase	CPU time (s) with noise	CPU time (s) w/o noise	Bytes sent (MB) with noise	Bytes sent (MB) w/o noise
Setup	341.9	0.9	781.4	429.8
Aggregation	3600	2280.2	862.4	528.7
ReachEstimation	385.1	132.3	83.2	52.9
FreqEstimation	47.7	22.3	26.0	13.4
Total	4374.9	2435.7	1752.9	1024.8

Table 6. Average (10 runs) computation cost of running the protocol with $p = 20$, reach per publisher = 10M, $w = 2$, $T = 3$, LiquidLegions ($a = 12$, $m = 100K$), $f_{\max} = 15$, $\epsilon_\upsilon = \epsilon_\eta = 0.35\ln(3)$, $\epsilon_\chi = \epsilon_\kappa = \epsilon_\lambda = \ln(3)/10$, and $\delta_\upsilon = \delta_\lambda = \delta_\kappa = \delta_\eta = \delta_\chi = 2 \cdot 10^{-10}$.

	Total CPU time (s) with noise	Total CPU time (s) w/o noise	Total Bytes sent (MB) with noise	Total Bytes sent (MB) w/o noise
$p = 5$, reach per publisher = 100K	533.2	323.9	232.6	147.5
$p = 5$, reach per publisher = 10M	927.4	675.6	413.7	323.3
$p = 20$, reach per publisher = 100K	2577.5	1150.5	1119.6	457.0

Table 7. Average (10 runs) computation cost of running the protocol with different p and reach per publisher. The other settings are the same as the test in Table 6.

	Total CPU time (s) $w = 2$, $T = 3$	Total CPU time (s) $w = 5$, $T = 6$	Total Bytes sent (MB) $w = 2$, $T = 3$	Total Bytes sent (MB) $w = 5$, $T = 6$
$p = 5$, reach per publisher = 10M	927.4	2711.2	413.7	900.6
$p = 20$, reach per publisher = 10M	4374.9	12962.4	1752.9	4291.1

Table 8. Average (10 runs) computation cost of running the protocol with different w and T. The other settings are the same as the test in Table 6.

8 Evaluation

8.1 Computational Costs

We implemented the proposed MPC protocol in C++ using the cryptographic primitive provided by Google's Private-Join-And-Compute library [40], which is built on top of OpenSSL. All crypto operations are done on the NID_X9_62_prime256v1 elliptical curve and each crypto word is 32 bytes. Deploying our experimental implementation of the protocol on the Google Kubernetes Engine (CPU base frequency = 3.1 GHz, Memory = 16 GB), with data sets containing 20 publishers, 10M reach per publisher and 3 MPC nodes ($w = 2$, $T = 3$), the measured computation and communication costs are shown in Table 6. For the dataset, we sampled the IDs independently and uniformly at random from a known universe, and inserted them into each sketch. We set the $\textsc{Hash}(\cdot)$ function to SHA-256.

As expected, the *Aggregation* phase consumes the majority of the total CPU time, since it deals with non-aggregated sketches, whose total size highly depends on the number of publishers. The noises added also have a significant impact on the total computation and communication costs. Table 7 shows the cost of three other scenarios for comparison.

The total computation cost is linear in the number of workers in the system if no noise is added, since each additional non-aggregator worker introduces the same amount of extra work. When noise is applied, a larger number of MPC nodes will also result in more noise being added. Thus, the protocol costs more at each worker. As a result, the total computation cost increases somewhere between linearly and quadratically in the number of total MPC nodes. Table 8 shows the cost of two scenarios with 3 and 6 MPC nodes respectively.

8.2 Accuracy

Section 3 described the accuracy of cardinality and frequency estimation for unnoised LiquidLegions sketches. We recommended the paramter $a = 12$ for Internet-scale estimation. Here, we extend the theory to the case including the noise added by our protocol. Accordingly, we provide recommendations for the other parameters (m and $f_{\max}$).

Theorem 5. *Fix n/m and let $m \to \infty$. With noise,*

$$\frac{\sqrt{\mathrm{var}(\hat{n})}}{n} - \sqrt{\frac{g(a,m,n)}{m}} \to 0,$$

where

$$g(a,m,n) = f\left(\frac{n}{m}, a\right) + \frac{a^2\sigma_\upsilon^2}{m\left(\exp(-e^{-a}c) - \exp(-c)\right)^2},$$

with σ_υ^2 given in (8), f in (4), and c in (5).

The proof sketch of Theorem 5 is given in Appendix B. As in Section 3, we plot the dependence of the minimum sketch length $m_{\min}$ on the other parameters. We follow

the recommendation of $a = 12$ in Section 3, and still consider the rstd threshold $\alpha = 2.5\%$. Figure 6 shows the dependence of $m_{\min}$ on ϵ_υ and $\log_{10}(n)$. It can be seen that for $m = 10^5$ we can afford any $\epsilon_\upsilon \geq 0.1$ (while larger m is needed for smaller values of ϵ_υ).

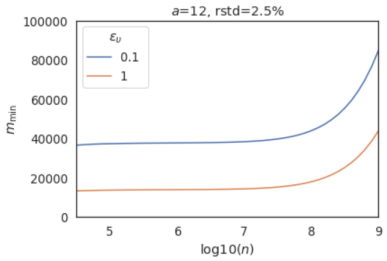


Fig. 6. The minimum m to achieve $\alpha = 2.5\%$ rstd for cardinality estimation, for different values of ϵ_υ and $\log_{10}(n)$, when $a = 12$.

Now, fix $m = 10^5$ and consider frequency estimation with noise. In this case, there does not exist a closed-form for $\text{std}(\hat{r}_i)$. It is easy to see though that the larger $f_{\max}$, the larger $\text{std}(\hat{r}_i)$. By simulation we obtained the dependence of the maximum affordable $f_{\max}$ on ϵ_η and $\log_{10}(n)$, as shown in Figure 7. It can be seen that for any $\epsilon_\eta \geq 0.1$ and $n \in [10^5, 10^9]$, we can achieve an std of 1% for any $f_{\max} = 15$. With larger $\epsilon = 1$, we can even afford an $f_{\max}$ of 200. Note that Figure 7 shows that $f_{\max}$ decreases for large n. This is indeed to be expected since the LiquidLegions sketch gradually saturates and there are fewer active registers.

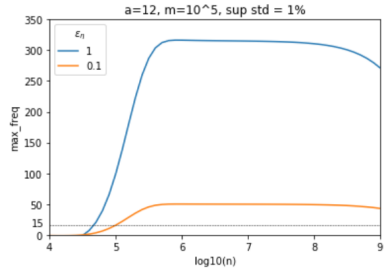


Fig. 7. The maximum $f_{\max}$ to achieve $\alpha_{\text{freq}} = 1\%$ std for frequency estimation, for different values of ϵ_η and $\log_{10}(n)$, when $a = 12$ and $m = 10^5$. Dashed line: $f_{\max} = 15$.

A cross-media measurement initiative study run by the World Federation of Advertisers has evaluated the accuracy of our sketch against several additional datasets and has shown the accuracy of the method to be independent of the input data distribution [52].

9 Conclusions & Future Work

In this work, we presented a scalable, secure, and private protocol for reach and frequency estimation.

With the goal of keeping the implementation simple, we assumed in this work that each publisher communicates to exactly one worker at the beginning of the protocol. It would be interesting to investigate if allowing interactive communication between the publishers and the other parties (workers and/or aggregator) could lead to better privacy-accuracy-communication trade-offs.

The DP guarantees we proved in fact bound the privacy leakage both from the execution of the MPC protocol, and from the release of its output, using a single (ϵ, δ) pair of privacy parameters. As the output is usually to be shared much more broadly than the internal state, it seems natural to separate the privacy guarantee into two (ϵ, δ) pairs of parameters: one for the output (which is still set strictly), and one for the internal execution (which could be set more generously). Our preliminary investigations indicate that doing so could result in a decent reduction in the communication and computational overhead of the protocol.

Finally, it would be very interesting to extend our protocol to the malicious setting where a publisher could deviate from the protocol, either to compromise the privacy of users or to distort the protocol's output.

10 Acknowledgements

We thank Matthew Clegg, Raimundo Mirisola, and Jelani Nelson for helpful discussions and feedback.

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

[1] J. M. Abowd. The US Census Bureau adopts differential privacy. In *KDD*, pages 2867–2867, 2018.

[2] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *Computing Surveys*, 79, 2018.

[3] M. Alaggan, M. Cunche, and S. Gambs. Privacy-preserving wi-fi analytics. *PoPETs*, pages 4–26, 2018.

[4] M. Alaggan, S. Gambs, and A.-M. Kermarrec. Blip: Non-interactive differentially-private similarity computation on Bloom filters. In *Stabilization, Safety, and Security of Distributed Systems*, pages 202–216, 2012.

[5] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *JCSS*, 58(1):137–147, 1999.

[6] Apple Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 2017.

[7] V. Balcer, A. Cheu, M. Joseph, and J. Mao. Connecting robust shuffle privacy and pan-privacy. In *SODA*, pages 2384–2403, 2021.

[8] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.

[9] A. Beimel, K. Nissim, and E. Omri. Distributed private data analysis: Simultaneously solving how and what. In *CRYPTO*, pages 451–468, 2008.

[10] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD*, pages 199–210, 2007.

[11] A. Bittau, Ú. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnés, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, pages 441–459, 2017.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[13] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, pages 1175–1191, 2017.

[14] D. Boneh. The decision Diffie–Hellman problem. In *ANTS*, pages 48–63, 1998.

[15] J. Brody, A. Chakrabarti, R. Kondapally, D. P. Woodruff, and G. Yaroslavtsev. Beyond set disjointness: the communication complexity of finding the intersection. In *PODC*, pages 106–113, 2014.

[16] T. H. Chan, E. Shi, and D. Song. Optimal lower bound for differentially private multi-party aggregation. In *ESA*, pages 277–288, 2012.

[17] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

[18] L. Chen, B. Ghazi, R. Kumar, and P. Manurangsi. On distributed differential privacy and counting distinct elements. In *ITCS*, pages 56:1–56:18, 2021.

[19] A. Cheu, A. D. Smith, J. Ullman, D. Zeber, and M. Zhilyaev. Distributed differential privacy via shuffling. In *EUROCRYPT*, pages 375–403, 2019.

[20] S. G. Choi, D. Dachman-Soled, M. Kulkarni, and A. Yerukhimovich. Differentially-private multi-party sketching for large-scale statistics. *PoPETs*, 3:153–174, 2020.

[21] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *JCSS*, 55(3):441–453, 1997.

[22] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[23] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. *TALG*, 7(2):1–20, 2011.

[24] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, pages 259–282, 2017.

[25] D. Desfontaines, A. Lochbihler, and D. Basin. Cardinality estimators do not preserve privacy. *PoPETs*, pages 26–46, 2019.

[26] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *NIPS*, pages 3571–3580, 2017.

[27] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA*, pages 605–617, 2003.

[28] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT*, pages 486–503, 2006.

[29] C. Dwork, F. McSherry, K. Nissim, and A. D. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[30] C. Dwork and A. Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[31] R. Egert, M. Fischlin, D. Gens, S. Jacob, M. Senker, and J. Tillmanns. Privately computing set-union and set-intersection cardinality via Bloom filters. In *Information Security and Privacy*, pages 413–430, 2015.

[32] Ú. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, K. Talwar, and A. Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *SODA*, pages 2468–2479, 2019.

[33] Ú. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067, 2014.

[34] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *SIGCOMM Comput. Commun. Rev.*, 32(4):323–336, 2002.

[35] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *IMC*, pages 153–166, 2003.

[36] D. Evans, V. Kolesnikov, and M. Rosulek. A Pragmatic Introduction to Secure Multi-Party Computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246, 2018.

[37] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, pages 137–156, 2007.

[38] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *JCSS*, 31(2):182–209, 1985.

[39] A. Ghosh, T. Roughgarden, and M. Sundararajan. Universally utility-maximizing privacy mechanisms. *SICOMP*, 41(6):1673–1693, 2012.

[40] Google. Private join and compute. https://github.com/google/private-join-and-compute.

[41] A. Greenberg. Apple's "differential privacy" is about collecting your data – but not your data. *Wired, June*, 13, 2016.

[42] H. Harmouch and F. Naumann. Cardinality estimation: An experimental survey. *VLDB*, 11(4):499–512, 2017.

[43] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially-private histograms through consistency. *VLDB*, 3(1):1021–1032, 2010.

[44] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*, page 683–692, 2013.

[45] M. Ion, B. Kreuter, A. E. Nergiz, S. Patel, S. Saxena, K. Seth, M. Raykova, D. Shanahan, and M. Yung. On

deploying secure computing: Private intersection-sum-with-cardinality. In *EuroS&P*, pages 370–389, 2020.
[46] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography from anonymity. In *FOCS*, pages 239–248, 2006.
[47] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *PODS*, pages 41–52, 2010.
[48] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, and A. Smith. What can we learn privately? *SICOMP*, 40(3):793–826, 2011.
[49] D. Mir, S. Muthukrishnan, A. Nikolov, and R. N. Wright. Pan-private algorithms via statistics on sketches. In *PODS*, pages 37–48, 2011.
[50] I. Mironov, O. Pandey, O. Reingold, and S. P. Vadhan. Computational differential privacy. In *CRYPTO*, pages 126–142, 2009.
[51] G. W. Oehlert. A note on the delta method. *The American Statistician*, 46(1):27–29, 1992.
[52] W. F. of Advertisers. Cross-media measurement initiative. https://github.com/world-federation-of-advertisers/cross_media_measurement_project_site/blob/master/public_papers/PRFE_results/PrivateReach&FrequencyEstimatorsEvaluationResults.md, 2020.
[53] R. Pagh and N. M. Stausholm. Efficient differentially private f_0 linear sketching. In *ICDT*, 2021.
[54] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over $\mathrm{GF}(p)$ and its cryptographic significance. *IEEE TOIT*, 24(1):106–110, 1978.
[55] S. Shankland. How Google tricks itself to protect Chrome user privacy. *CNET, October*, 2014.
[56] E. Skvortsov, J. Wilhelm, W. Bradbury, J. Bao, A. Ulbrich, and L. Tsang. Tracking audience statistics with hyperloglog. *Google Research Tech Report*, 2021.
[57] A. T. Suresh. Differentially private anonymized histograms. In *NeurIPS*, pages 7971–7981, 2019.
[58] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of Bloom filters for distributed systems. *IEEE Communications Surveys Tutorials*, 14(1):131–155, 2012.
[59] S. Vadhan. The complexity of differential privacy. In *Tutorials on the Foundations of Cryptography*, pages 347–450. Springer, 2017.
[60] T. Wang, J. Blocki, N. Li, and S. Jha. Locally differentially private protocols for frequency estimation. In *USENIX*, pages 729–745, 2017.
[61] D. P. Woodruff and Q. Zhang. An optimal lower bound for distinct elements in the message passing model. In *SODA*, pages 718–733, 2014.
[62] Y. W. Yu and G. M. Weber. Balancing accuracy and privacy in federated queries of clinical data repositories: Algorithm development and validation. *J Med Internet Res*, 22(11):e18735, Nov 2020.

A Privacy Proof

We will now give more details about the differential privacy proof of the protocol (Theorem 4).

In order to prove that the *view* of the corrupted parties is *computational* DP, we may invoke the security property of our protocol (Theorem 3) to reduce the task to showing that the *output* of the corrupted parties is DP. To summarize, it suffices for us to prove:

Theorem 6. *The outputs of any set of all but T parties (i.e., workers or aggregator) of the protocol is $(\epsilon_\upsilon + \epsilon_\lambda + \epsilon_\kappa + \epsilon_\eta + \epsilon_\chi, \delta_\upsilon + \delta_\lambda + \delta_\kappa + \delta_\eta + \delta_\chi)$-DP.*

The rest of this section is devoted to the proof of Theorem 6 and is organized as follows. First, in Appendix A.1, we provide several additional definitions, facts and lemmas that will be used throughout the proof. Then, in Appendix A.2, we specify the properties of the sketches that we need for the privacy proof. The main proof itself is separated into two cases, one where the aggregator is corrupted and the other where the aggregator is not corrupted; these two cases are handled in Appendix A.3 and Appendix A.4, respectively.

A.1 Privacy Primitives

For a discrete distribution D, we use $D(x)$ to denote the probability mass at x. By $X \sim D$, we denote a random variable X independently sampled from D. Let $D * D'$ denote the convolution of D, D', i.e., the distribution of $X + Y$ where $X \sim D, Y \sim D'$ are independent. For brevity, let D^{*n} denote the n-fold convolution of D (i.e., convolving D with itself $n-1$ times).

We now introduce a few distributions that are used throughout this work, starting with the discrete Laplace (aka symmetric geometric) distribution:

Definition 4 (Discrete Laplace Distribution)**.** *The discrete Laplace distribution with parameters $\mu \in \mathbb{Z}, s \in \mathbb{R}_{>0}$, denoted by $\mathrm{DLap}_{\mu,s}$, is supported on $\mathbb{Z}$, and given by $\mathrm{DLap}_{\mu,s}(x) \propto \exp\left(-|x-\mu| \cdot s\right)$.*

We will also use (a truncated version) of the Polya (aka Negative Binomial) distribution, defined next.

Definition 5 (Truncated Polya Distribution)**.** *The truncated Polya distribution with parameters $r > 0, p \in [0,1], u \in \mathbb{Z}_{\geq 0}$, denoted by $\mathrm{tPolya}_{r,p,u}$, is supported on $\{1, \ldots, u\}$ with the following probability mass function:*

$$\mathrm{tPolya}_{r,p,u}(x) \propto \binom{x+r-1}{x}(1-p)^r p^x.$$

Finally, for notational convenience, we define the distribution of the difference of two truncated Polya distributions (shifted by μ):

Definition 6 (Truncated Polya Difference Distribution). *The Truncated Polya Difference distribution with parameters $\mu \in \mathbb{Z}_{\geq 0}, r \in \mathbb{R}_{>0}, p \in [0,1]$, denoted by $\text{tPolyaDiff}_{\mu,r,p}$, is defined as the distribution of $\mu + X - Y$ where X, Y are independent identically distributed $\text{tPolya}_{r,p,\mu}$ random variables.*

Let $\mu(\epsilon, \delta, \Lambda, T) := \lceil \ln(2T\Lambda(1+e^{\epsilon})/\delta)/(\epsilon/\Lambda) \rceil$ be as in the SampleNoise algorithm (Algorithm 4). Observe that the noise output by SampleNoise is exactly distributed as $\text{tPolyaDiff}_{\mu(\epsilon,\delta,\Lambda,T),1/T,e^{-\epsilon/\Lambda}}$.

It is known [39] that the algorithm that adds discrete Laplace noise to the output is DP (when parameters are appropriately chosen). Furthermore, it is also well-known that the discrete Laplace distribution can be written as a convolution of the (non-truncated and non-shifted) Polya Difference. By bounding the difference in the statistical distance due to truncation, we arrive at the following DP guarantee of an algorithm that adds a random variable distributed as the convolution of truncated Polya Difference to the output. Recall that for a function f whose range is $\mathbb{R}^d$, its (ℓ_1-)sensitivity is $\max_{\mathbf{X},\mathbf{X}'} \sum_{i \in [d]} |f(\mathbf{X})_i - f(\mathbf{X}')_i|$ where the maximum is over neighboring datasets $\mathbf{X}, \mathbf{X}'$.

Lemma 1. *Let f be any function whose output is a vector of integers, and whose sensitivity is at most Λ. For any $\mu \geq \mu(\epsilon, \delta, \Lambda, n)$, an algorithm that adds an independent noise term sampled from $(\text{tPolyaDiff}_{\mu,1/n,e^{-\epsilon/\Lambda}})^{*n}$ to each output coordinate is (ϵ, δ)-DP.*

In our analysis below, we often have multiple overlapping parts of the input. The following is an observation that we may subtract a part of the output from another part without compromising on privacy; this will allow us to substantially simplify the analysis.

Fact 1. *Let Alg be any algorithm whose outputs are $\lambda_1, \ldots, \lambda_\Lambda$. Let i, j be any fixed indices in $[\Lambda]$, and define Alg$'$ to be an algorithm which is the same as Alg except that λ_i is changed to $\lambda_i - \lambda_j$. Then, Alg is (ϵ, δ)-DP iff Alg$'$ is (ϵ, δ)-DP.*

Once we simplify the outputs, we are often left with parts of the input that are either constants (e.g., zeros) or just an independent noise that does not appear anywhere else. The following lemma lets us completely discard such a part.

Fact 2. *Let Alg be any algorithm whose outputs are $\lambda_1, \ldots, \lambda_\Lambda$. Suppose that λ_1 is a random variable sampled from a distribution that does not depend on the users' inputs. Furthermore, suppose that $\lambda_2, \ldots, \lambda_\Lambda$ are independent of λ_1. Let Alg$'$ be the algorithm that is the same as Alg except that it only outputs $\lambda_2, \ldots, \lambda_\Lambda$. Then, Alg is (ϵ, δ)-DP iff Alg$'$ is (ϵ, δ)-DP.*

In working with multiple output parts, it is sometimes useful to include more parts in the output in order to further simplify the other terms. Intuitively, this can only increase the power of the adversary. The following lemma formalizes this property.

Fact 3. *Let Alg be any algorithm whose outputs are $\lambda_1, \ldots, \lambda_\Lambda$. Let $\lambda_{\Lambda+1}$ be any random variable (possibly dependent on $\lambda_1, \ldots, \lambda_\Lambda$). Consider an algorithm Alg$'$ whose outputs are $\lambda_1, \ldots, \lambda_{\Lambda+1}$. If Alg$'$ is (ϵ, δ)-DP, then Alg is (ϵ, δ)-DP.*

We will also use the following well-known property of differential privacy.

Theorem 7 (Basic Composition Theorem; e.g., [30]). *An algorithm that applies a (possibly adaptive) sequence of (ϵ_1, δ_1)-DP, $\ldots$, (ϵ_m, δ_m)-DP algorithms is $(\epsilon_1 + \cdots + \epsilon_m, \delta_1 + \cdots + \delta_m)$-DP.*

A.2 Properties of the Sketches

To state the properties of the sketches that we need, let us first introduce a few additional notation for the sketches and intermediate states:

- h: the "blinded histogram" of the noise registers. Specifically, $h \in \mathbb{Z}^p_{\geq 0}$, and h_a denotes the number of register IDs that are non-empty in exactly a of the publisher sketches.
- $\hat{F}$: the frequency histogram with flags. Specifically, $\hat{F} \in \mathbb{Z}^{[f_{\max}] \times \{(0,1,1),(0,0,1),(1,1,1)\}}_{\geq 0}$ where $\hat{F}_{(b,\mathit{flag}_1,\mathit{flag}_2,\mathit{flag}_3)}$ denotes the number of registers which are undestroyed and have count b for $b \leq f_{\max} - 1$ or count at least $f_{\max}$ for $b = f_{\max}$, the first flag value set to flag_1, the second flag value set to flag_2 and the third flag value set to flag_3. (This is with respect to the *unnoised* sketch. Moreover, for convenience, we use 1 to denote any non-zero flag values.)

– For notational convenience, we define F to be the "undestroyed" counterpart of $\hat{F}$; specifically, for all $b \in [f_{\max}]$, we let $F_b = \hat{F}_{(b,0,1,1)}$.

Throughout our analysis, we use the following properties of the sketches:

– Adding or removing a user changes at most p coordinates of the sketches in total across *all* publishers. This holds for LiquidLegions because adding/removing a user only changes at most one coordinate in each publisher's sketch.
– Adding or removing a user increases at most one entry of the blinded histogram h by a value of at most one and decreases at most one entry of h by a value of at most one. This holds for LiquidLegions because adding/removing a user affects a single register index. If this register ID appeared k times before the addition/removal and k' times after, then the blinded histogram will decrease at bucket k and increase at bucket k'.
– Adding or removing a user increases at most one entry of $\hat{F}$ by a value of at most one and decreases at most one entry of $\hat{F}$ by a value of at most one. The reason this holds for LiquidLegions is similar to that of h.

The last two items imply that the sensitivities of h and $\hat{F}$ are at most two. This is somewhat of an unusual bound as histograms often have sensitivity one in the case of addition/removal notion for neighboring datasets. However, as explained above, adding/removing a user in our setting corresponds to "moving" a register ID from one bucket to another, resulting in the sensitivity of two.

A.3 Privacy Proof for Collusion With Aggregator

We begin with the case when the collusion includes the aggregator, which is the more challenging one:

Theorem 8. *The outputs of the aggregator together with any set of all but T workers are $(\epsilon_\upsilon + \epsilon_\lambda + \epsilon_\kappa + \epsilon_\eta + \epsilon_\chi, \delta_\upsilon + \delta_\lambda + \delta_\kappa + \delta_\eta + \delta_\chi)$-DP.*

To prove the above theorem, we need to precisely state the outputs of the aggregator and any set of workers, for which we will prove the privacy guarantee. Here we use $x^{(i)}$ to denote the number of non-zero coordinates in the ith sketch (before noising).

Lemma 2. *Consider a set $\mathcal{I}$ of the aggregator and workers. The outputs of $\mathcal{I}$ are as follows, where $\bar{A}$ denotes the set of workers not included in the collusion.*

1. *Noisy number of non-empty registers:* $\left(\sum_{k\in[p]} h_k\right) + \sum_{j\in\bar{A}} \upsilon^{worker(j)}$
2. *Noisy frequency histogram:* $F + \sum_{j\in\bar{A}} \eta^{worker(j)}$
3. *Intermediate outputs:*
   (a) *The following for all $i \in [p]$:* $x^{(i)} + \lambda^{publisher(i)}$
   (b) $\left(\sum_{j\in\bar{A}} \chi^{worker(j)}\right) - \sum_{k\in[p]} k \cdot h_k$
   (c) $(h + \sum_{j\in\bar{A}} \kappa^{worker(j)})_k$ *for all* $k \in \{2, \ldots, p\}$
   (d) $-\sum_{k\in\{2,\ldots,p\}} h_k + \left(\sum_{j\in\bar{A}} \kappa_1^{worker(j)}\right)$
   (e) $\left(\sum_{b\in[f_{\max}]} \hat{F}_{(b,1,1,1)}\right) + \left(\sum_{j\in\bar{A}} \hat{\eta}^{worker(j)}\right)$

The proof of Lemma 2 is deferred to the full version. We are now ready to prove Theorem 8.

Proof of Theorem 8. Suppose that the adversary can access the outputs of the aggregator and workers j for all $j \in A$ where $|A| = w - T$. We write $\bar{A}$ to denote $[w] \setminus A$, i.e., the set of workers not compromised by the adversary.

Recall the outputs from Lemma 2. We may now compute the privacy guarantees for each part of the above output as follows:

1. Since each $\upsilon^{\text{worker}(j)}$ is sampled from $\text{tPolyaDiff}_{\mu_\upsilon, 1/T, e^{-\epsilon_\upsilon}}$ with $\mu_\upsilon = \mu(\epsilon_\upsilon, \delta_\upsilon, 1, T)$ and the sensitivity of $\sum_{k\in[p]} h_k$ is one, we can apply Lemma 1 to conclude that the algorithm that only outputs this item is $(\epsilon_\upsilon, \delta_\upsilon)$-DP.

2, 3(e). Since each $\hat{\eta}^{\text{worker}(j)}$ and each $\eta_b^{\text{worker}(j)}$ is sampled from $\text{tPolyaDiff}_{\mu_\eta, 1/T, e^{-\epsilon_\eta/2}}$, $\mu_\eta = \mu(\epsilon_\eta, \delta_\eta, 2, T)$ and the ℓ_1-sensitivity of $\hat{F}$ is at most two, we can apply Lemma 1 to conclude that the algorithm that only outputs items 2 and 3(e) is $(\epsilon_\eta, \delta_\eta)$-DP.

3(a). Since $\lambda^{\text{publisher}(i)} \sim \text{tPolyaDiff}_{\mu_\lambda, 1, e^{-\epsilon_\lambda/p}}$, $\mu_\lambda = \mu(\epsilon_\lambda, \delta_\lambda, p, 1)$ and the sensitivity of $(x^{(i)})_{i\in[p]}$ is p, we may apply Lemma 1, which implies that the algorithm outputting just the last item is $(\epsilon_\lambda, \delta_\lambda)$-DP.

3(b). Since $\chi^{\text{worker}(j)} \sim \text{tPolyaDiff}_{\mu_\chi, 1/T, e^{-\epsilon_\chi/p}}$, $\mu_\chi = \mu(\epsilon_\chi, \delta_\chi, p, T)$ and the sensitivity of $-\sum_{k\in[p]} k \cdot h_k$ is at most p), we can apply Lemma 1 to conclude that the algorithm that only outputs this item is $(\epsilon_\chi, \delta_\chi)$-DP.

3(c), 3(d). Consider $(h_2, \ldots, h_k, -\sum_{k\in\{2,\ldots,p\}} h_k)$. We claim that its sensitivity is at most two. To see that this is true, recall that adding or removing a user increases at most one entry of h by at most one and decreases at most one entry of h by at most one. As a result, if $-\sum_{k\in\{2,\ldots,p\}} h_k$ remains the same, then the total change in other coordinates is at most two as de-

sired. On the other hand, if $-\sum_{k\in\{2,\ldots,p\}} h_k$ changes, it can change by at most one and it also implies that at most one of $h_2,\ldots,h_k$ changes by at most one; hence, in this case, the total change is at most two as well.

Since $\kappa_k^{\text{worker}(j)} \sim \text{tPolyaDiff}_{\mu_\kappa,1/T,e^{-\epsilon_\kappa/2}}$, $\mu_\kappa = \mu(\epsilon_\kappa,\delta_\kappa,2,T)$ and the sensitivity of $(h_2,\ldots,h_k,-\sum_{k\in\{2,\ldots,p\}} h_k)$ is at most two, we can apply Lemma 1 to conclude that the algorithm that only outputs these two items is $(\epsilon_\kappa,\delta_\kappa)$-DP.

By basic composition of differential privacy (Theorem 7), all the outputs combined is $(\epsilon_\upsilon+\epsilon_\lambda+\epsilon_\kappa+\epsilon_\eta+\epsilon_\chi,\delta_\upsilon+\delta_\lambda+\delta_\kappa+\delta_\eta+\delta_\chi)$-DP as desired. □

A.4 Proof for Collusion Without Aggregator

Finally, we analyze the easier case where the aggregator is not part of the collusion, stated below.

Theorem 9. *The outputs of any set of all but $T-1$ workers of the protocol is $(\epsilon_\lambda+\epsilon_\upsilon+\epsilon_\eta,\delta_\lambda+\delta_\upsilon+\delta_\eta)$-DP.*

It should be noted here that the privacy guarantee in this case is quantitatively stronger than that of the general case (Theorem 6); specifically, the former guarantees $(\epsilon_\lambda+\epsilon_\upsilon+\epsilon_\eta,\delta_\lambda+\delta_\upsilon+\delta_\eta)$-DP, whereas the latter only guarantees $(\epsilon_\upsilon+\epsilon_\lambda+\epsilon_\kappa+\epsilon_\eta+\epsilon_\chi,\delta_\upsilon+\delta_\lambda+\delta_\kappa+\delta_\eta+\delta_\chi)$-DP. In other words, if we assume that the aggregator is not compromised, then we can add less noise while maintaining a similar level of privacy.

We now precisely state the outputs of any set of workers, for which we will prove the privacy guarantee:

Lemma 3. *Consider a set $\mathcal{I}$ of the workers. The outputs of $\mathcal{I}$ are as follows, where $\bar{A}$ denotes the set of workers not part of the collusion.*

1. $\left(\sum_{b\in[f_{\max}],(\mathit{flag}_1,\mathit{flag}_2,\mathit{flag}_3)\in\{(0,1,1),(0,0,1),(1,1,1)\}} \hat{F}_{(b,\mathit{flag}_1,\mathit{flag}_2,\mathit{flag}_3)}\right) + \upsilon^{\mathit{aggregator}} + \left(\sum_{j\in\bar{A}} \upsilon^{\mathit{worker}(j)}\right)$
2. $\left(\sum_{b\in[f_{\max}]} F_b + \eta_b^{\mathit{aggregator}}\right) + \left(\sum_{j\in\bar{A}}\sum_{b\in[f_{\max}]} \eta_b^{\mathit{worker}(j)}\right)$
3. *The following for all $i\in[p]$:* $x^{(i)} + \lambda^{\mathit{publisher}(i)}$

The proof of Lemma 3 is deferred to the full version. We are now ready to prove Theorem 9.

Proof of Theorem 9. Suppose that the adversary can access the outputs of workers j for all $j\in A$ where $|A| = w-T+1$. Recall the outputs from Lemma 3. We may now compute the privacy guarantees for each part of the above output as follows:

1. Since each $\upsilon^{\text{worker}(j)}$ and $\upsilon^{\text{aggregator}}$ is sampled from $\text{tPolyaDiff}_{\mu_\upsilon,1/T,e^{-\epsilon_\upsilon}}$, with $\mu_\upsilon = \mu(\epsilon_\upsilon,\delta_\upsilon,1,T)$ and the sensitivity of $\left(\sum_{b\in[f_{\max}],(\mathit{flag}_1,\mathit{flag}_2,\mathit{flag}_3)\in\{(0,1,1),(0,0,1),(1,1,1)\}} \hat{F}_{(b,\mathit{flag}_1,\mathit{flag}_2,\mathit{flag}_3)}\right)$ is at most one, we can apply Lemma 1 to conclude that the algorithm that only outputs this item is $(\epsilon_\upsilon,\delta_\upsilon)$-DP.
2. Since each $\eta_b^{\text{worker}(j)}$ and $\eta_b^{\text{aggregator}}$ are sampled from $\text{tPolyaDiff}_{\mu_\eta,1/(T),e^{-\epsilon_\eta/2}}, \mu_\eta = \mu(\epsilon_\eta,\delta_\eta,2,T)$ and the sensitivity of F is at most two, we can apply Lemma 1 to conclude that the algorithm that only outputs the second item is $(\epsilon_\eta,\delta_\eta)$-DP.
3. Since $\lambda^{\text{publisher}(i)} \sim \text{tPolyaDiff}_{\mu_\lambda,1,e^{-\epsilon_\lambda/p}}$, $\mu_\lambda = \mu(\epsilon_\lambda,\delta_\lambda,p,1)$ and the sensitivity of $(x^{(i)})_{i\in[p]}$ is p, we may apply Lemma 1, which implies that the algorithm outputting just the last item is $(\epsilon_\lambda,\delta_\lambda)$-DP.

By the basic composition of differential privacy (Theorem 7), we can conclude that the entire algorithm is $(\epsilon_\lambda+\epsilon_\upsilon+\epsilon_\eta,\delta_\lambda+\delta_\upsilon+\delta_\eta)$-DP. □

B Accuracy Proof for Cardinality Estimator

This section proves Theorems 1 and 5 on the accuracy of the cardinality estimator. We aim to estimate n from the LiquidLegions sketch. Let X denote the number of non-empty registers in the sketch. Then $X = X^{\text{raw}} + \xi$, where X^{raw} is the number of non-empty registers in the unnoised sketch, and ξ is the noise introduced by MPC. The distribution of X^{raw} is determined by the register probabilities p_j $(1\le j\le m)$. The noise ξ has mean zero and variance being the σ_υ^2 in equation (8). The cardinality is estimated as

$$\hat{n} = \mathcal{E}^{-1}(X/m),$$

with $\mathcal{E}$ given in (3).

Below we sketch the proof of Theorem 5; Theorem 1 is a special case of it.

Since $\mathcal{E}(\hat{n}) = X$, by the delta method [51],

$$\mathbb{E}(\hat{n}) \approx \mathcal{E}^{-1}\left(\mathbb{E}(X)/m\right).$$

and

$$\text{var}(\hat{n}) = \frac{\text{var}(X)}{m^2[\mathcal{E}'(\mathbb{E}(\hat{n}))]^2}.$$

asymptotically. The asymptotic conditions can be specified as "$m \to \infty$ while fixing $n/m = z$" as stated in the theorem.

$\mathbb{E}(X) = \mathbb{E}(X^{\text{raw}})$ and $\text{var}(X) = \text{var}(X^{\text{raw}}) + \sigma_v^2$. X^{raw} can be expressed as $\sum_{j=1}^{m} \Lambda_j$, where

$$\Lambda_j = I[\text{at least one item is in the } j\text{th register}].$$

By definition, it is easy to obtain that $\mathbb{E}(\Lambda_j) = (1-p_j)^n \sim \exp(-np_j)$, $\text{var}(\Lambda_j) = (1-p_j)^n - (1-p_j)^{2n} \sim \exp(-np_j) - \exp(-2np_j)$, and $\text{cov}(\Lambda_i, \Lambda_j) = (1-p_i-p_j)^n - (1-p_i)^n(1-p_j)^n \sim [\exp(-np_ip_j) - 1]\exp(-np_i)\exp(-np_j)$, where $\sim$ means "asymptotically equivalent to", for any $1 \le j \le m$ and any $i \ne j$.

Then $\mathbb{E}(X^{\text{raw}})$ and $\text{var}(X^{\text{raw}})$ can be expressed as summations of exponential functions on p_j and can be further approximated as exponential integrals. From this, $\mathbb{E}(X)$ and $\text{var}(X)$ and hence, $\mathbb{E}(\hat{n})$ and $\text{var}(\hat{n})$ can be obtained.

C Accuracy Proof for Frequency Estimator

This section proves Theorem 2 on the accuracy of the frequency estimator. As explained following Algorithm 2, the items in the active registers form an unbiased sample of all the items. In Algorithm 2, H is the frequency histogram of the unbiased sample, and thus H follows a multivariate hypergeometric distribution with parameters n, A, and $[r_i \cdot n]_{1 \le i \le f_{\max}}$, conditional on the number of active registers A. Intuitively, consider $f_{\max}$ different bins with $n \times r_i$ balls in the ith bin, and random drawing A balls from all the $f_{\max}$ bins—then H is distributed as the number of drawn balls from each bin. Following the properties of the multivariate hypergeometric distribution, we have $\mathbb{E}(H[i]) = Ar_i$ and

$$\begin{aligned}\text{var}(H[i]) &= A\frac{n-A}{n-1}\frac{r_i n}{n}\left(1 - \frac{r_i n}{n}\right) \\ &\approx \frac{A(n-A)}{n} r_i(1-r_i),\end{aligned}$$

and thus $\hat{r}_i = H[i]/A$ has $\mathbb{E}(\hat{r}_i) = r_i$ and

$$\text{var}(\hat{r}_i) \approx \frac{n-A}{nA} r_i(1-r_i) \sim \frac{n - \mathbb{E}[A]}{n\mathbb{E}[A]} r_i(1-r_i),$$

where the last step follows from the law of large numbers, and $\sim$ means "on the order of"—in our context, $a \sim b$ if and only if $a/b \to 1$ as $n, A \to \infty$.

To complete the proof of Theorem 2, it remains to show that $\mathbb{E}[A] \sim \gamma \cdot m$ with γ defined in the theorem. Note that $A = \sum_{j=1}^{m} \Lambda'_j$ where

$$\Lambda'_j = I[\text{exactly one item is in the } j\text{th register}].$$

By definition, it is easy to obtain that $\mathbb{E}[\Lambda'_j] = np_j(1-p_j)^{n-1} \sim np_j \exp[-(n-1)p_j]$. Then $\mathbb{E}(A)$ can be approximated as its integral which turns out to be asymptotically equivalent to $\gamma \cdot m$.

D Security Proof Sketch

Proof of Theorem 3. Let $\mathcal{I}$ be the parties under simulation. Wlog, assume $|\mathcal{I}| = W - 1$ i.e., assume only one party is honest. As in privacy, we consider two cases: the case where the honest party is the aggregator, and the case where the aggregator is not honest.

Honest Aggregator. In this case the only outputs are the sizes of the encrypted registers seen by each worker in each round of communication, corresponding to the input sizes and the noise registers added by the workers and aggregator. Since the aggregator is honest, the simulator will act as the aggregator to pad the combined register vector sent to the first worker with appropriate noise. The CRV sent by the simulator should consist only of random group elements for each encrypted tuple, and security follows from a reduction to the security of ElGamal encryption (which is a reduction to DDH). The simulator repeats this process, with fresh random elements, for the beginning of the *ReachEstimation* phase. Finally, in the *FreqEstimation* phase, the simulator constructs a matrix of pairs of random EC elements and sends this matrix to the first worker, and once all workers have partially decrypted the matrix the simulator outputs whatever each worker outputs.

Adversarial Aggregator. Here, in addition to adding appropriate noise, the simulator will also prepare messages for the aggregator. At a high level the simulation strategy involves discarding the messages sent to the honest party, and generating new, synthetic messages from the honest party that will be based on the output of the protocol in each phase. As in the honest aggregator case, the simulator will pad messages to the appropriate size; however, instead of random messages, the simulator will encrypt, using a public key based on the combined public key shares from the remaining parties between the honest worker and the aggregator (including the aggregator's key), a message that is of the correct distribution for the aggregator's output in that round.

In the *Setup* phase, the aggregator must receive a register vector from each worker with noise added.

The simulator will generate this message for the honest worker, using random EC elements for each encrypted register, adding noise according to the aggregator's setup phase output (message size).

In the *Aggregation* phase, the simulator generates a message based on the aggregator's cardinality histogram (the register ID histogram). For each histogram bar h, the simulator will choose a random EC element and encrypt that element h times, along with two pairs of random elements, and add these to the simulated crv. It then shuffles this crv and sends it the next worker.

In the *ReachEstimation* phase, the aggregator's output is the LiquidLegions estimator X and the number of active registers A. Given a received message of size N the simulator generates a synthetic message as follows. First, A four-tuples are generated with a random pair of EC elements for the encrypted count, an encryption of 0 for $flag_1$, and encryptions of random EC elements for $flag_2$ and $flag_3$. $A-X-kD-2$ four-tuples will be added with an encryption of a random count and random values for $flag_1$, $flag_2$, and $flag_3$ where k is the number of workers preceding the honest worker. Next, the length is padded to kD with a random count, 0 for $flag_1$ and $flag_2$, and a random value for $flag_3$. Finally the message is padded to length N by adding a four-tuple with a random count, 0 for $flag_1$, random for $flag_2 2$ and 0 for $flag_3$. The simulator then pads this by adding its own noise and forwards to the next party.

In the *FreqEstimation* phase, the simulator sets up the SKA matrix according to the histogram, choosing random columns to set the 0 index for the corresponding rows for the histogram bars. □