Privacy-Preserving Secure Cardinality and Frequency Estimation

Craig Wright, Evgeny Skvortsov, Benjamin Kreuter, Yao Wang, Raimundo Mirisola

Google LLC

May 29, 2020

Abstract

In this paper we introduce a new family of methods for cardinality and frequency estimation. These methods combine aspects of HyperLogLog (HLL) and Bloom filters in order to build a sketch that, like HLL, is substantially more compact than a Bloom filter, but like a Bloom filter maintains the ability to union sketches with a bucket-wise sum. Together these properties enable the creation of a scalable secure multi-party computation protocol that takes advantage of homomorphic encryption to combine sketches across multiple untrusted parties. The protocol limits the amount of information that participants learn to differentially private estimates of the union of sketches and some partial information about the Venn diagram of the per-sketch cardinalities.

1 Introduction

The cardinality estimation problem has a rich history in the computer science literature and a vast number of methods have been invented to solve it [12]. On the other hand, the frequency problem, as we define it, does not have such widespread recognition, one exception being in the advertising industry where it is a critical metric for audience reporting.

The reach of an advertising campaign is defined as the number of unique users or households, (henceforth just users), who have been exposed to that campaign. Essentially reach is just cardinality, and frequency is the number of times that a user has been exposed to an advertisement. There are two common ways of exposing the frequency of a campaign or inventory: the frequency histogram and k+ reach. Frequency histogram shows what fraction of users have each frequency, while k+ reach is the cardinality of the set of users that have frequency greater than or equal to k. Overall this is a relatively straightforward extension of the cardinality problem, but estimating frequency for a range of values with only a single pass over a dataset is important, and given the economic usefulness of this metric, worthy of consideration. Often the maximum value required for frequency is less than 15 as advertisers use this metric to help ensure people are not overexposed to their advertisements.

A further complication to the reach and frequency use case is that we wish to determine these metrics across $O(100)$ publishers for frequencies of up to 10 or 15, where a publisher is defined as an entity that hosts content and shows advertisements on behalf of advertisers. However, since publishers compete with each other for advertising revenue and do not necessarily trust each other, we cannot assume that data, even sketches, can be shared. More importantly, commonly used sketches are known to leak information about their contents, which is something that we believe must be avoided. Therefore it is important that methods for reach and frequency measurement can preserve the privacy of individuals that make up the content of the sketches and prevent publishers from learning information about each other. This is congruent with many other industries and indeed there has been a significant amount of research in privacy-preserving cardinality estimation techniques, some of which we discuss in the next section.

Another requirement of the multi-publisher use case is the need to define a common identifier space from which to create sketches, however we do not consider this here. Rather we simply refer to a modeling technique previously developed that is capable of probabilistically mapping disparate identifier spaces into a common identifier space for the purpose of counting [17].

The rest of the document proceeds as follows. First we cover background material related to secure multiple party computation, HyperLogLog, Bloom filters, differential privacy, and ElGamal encryption. Then we describe Cascading Legions, which is a geometrically distributed array of Bloom filters for estimating cardinality and frequency. After that we describe Liquid Legions, which is a continuous version of the Cascading Legions algorithm. Once both of these algorithms are introduced we explore two mechanisms for applying differential privacy to each of the sketches prior to using them to estimate multi-publisher reach. Then we describe a multi-party computation protocol for combining sketches of either the discrete or continuous type cryptographically. This protocol produces frequency estimates that are differentially private. We leave a detailed empirical evaluation to a future version of this paper.

2 Background

2.1 Secure Multiparty Computation

Secure multiparty computation (MPC) protocols allow two or more parties to compute a function over their collective inputs without revealing to any other parties more about their inputs than the result of the computation. Early research on feasibility demonstrated that MPC protocols exist for all polynomial time functions under various trust models [11, 20]. More recently a growing number of real-world applications using MPC protocols have been reported [3, 4, 14].

Broadly speaking there are two widely accepted security definitions for MPC protocols. In the honest-but-curious model, it is assumed that each party faithfully executes each step of the protocol. A protocol is considered secure in the honest-but-curious model if for any proper subset of the parties, nothing more can be learned from an execution of the protocol than the parties could compute given only their respective outputs. The second security model is the malicious one, in which the parties may arbitrarily deviate from the protocol. Security in the malicious model requires that, in addition to the properties of the honest-but-curious model, every party is *committed* to its input, and that the outputs of the protocol remain consistent with the inputs used by each party [15]. These two models deal only with the *standalone* setting, but various extensions have been considered for the case where the parties may engage in multiple concurrent protocol sessions [5].

Note that MPC security definitions are only concerned with the mechanics of how the function is computed. What the output might reveal about the parties' inputs is an orthogonal concern, which in some cases can be addressed by also achieving the differential privacy property discussed below in section 2.4. Similarly, an MPC protocol's output is only as good as the inputs provided by each party, and the security definitions leave open the possibility of parties giving dishonest inputs to the computation. However, for this work the risk of parties supplying dishonest inputs is considered low because participants in the system will have to undergo accreditation and periodic audits to ensure compliance.

2.2 Bloom Filters

A Bloom filter [2], is a well-known algorithm that can be used for cardinality estimation, though it is not often used as such due to its relatively large memory requirements. Nonetheless, the cardinality of a Bloom filter can be determined by evaluating:

$$n \approx -\tfrac{m}{k} \ln\left(1 - \tfrac{x}{m}\right)$$

where k is the number of hashes used in the filter, m is the number of bits of the filter, x is the number of filled buckets, and n is the estimated cardinality. [19] Allowing k to go to 1 and inverting the function, we can estimate the number of active bits given the set size using the following formula:

$$X \approx m\left(1 - \exp\left(-\tfrac{n}{m}\right)\right)$$

Note that it is possible to use the cardinality formula iteratively with a counting Bloom filter to estimate frequency and that the protocol below can also be used to aggregate standard counting Bloom filters.

2.3 HyperLogLog

HyperLogLog (HLL) [8,13] is a well-known and widely used cardinality estimator that is characterized by low error rates, low variance, and low memory usage. Moreover, the union of two HLLs is trivial to compute, which means that HLLs are ideal for distributed counting applications.

In some related work at Google, an extension of HLL, called FreqLogLog (FLL) that can estimate both cardinality and frequency was developed. To achieve this FLLs store two additional values per register, a count and a fingerprint. When the HLL register value is updated, the fingerprint of the item is stored and the counter is reset to one. Then when another item assigned to that register has the same number of leading zeroes as the previous item, the fingerprints are compared. If they are the same the count is incremented. Otherwise if the new fingerprint is larger than the old fingerprint the new fingerprint is stored and the count is reset to 1. Otherwise the old fingerprint is larger and nothing is modified. Additional material on FLLs will be published in the future.

Unfortunately, HLLs and thus FLLs, suffer from serious privacy issues, and it has been shown [6] that under conditions where intermediate sketches are required and the parameters for constructing the sketches are public, then any sufficiently accurate cardinality estimator is not privacy preserving. We have thus far been unable to construct a satisfactory MPC protocol for operating on HLLs, the requirements for which are that it should be relatively efficient for a large number of input sketches, hide publisher inputs, and provide differentially private reach and frequency estimates that are reasonably accurate.

2.4 Differential Privacy

In this paper we consider how to apply differential privacy to sketches from individual publishers prior to aggregation and directly to the estimates of cardinality and frequency after aggregation. However, before getting into some of the details of our application, a few basic definitions will be reviewed.

Recall that differential privacy is a mathematical framework for data privacy that involves quantifying the contribution of a single user to the input of some computation and ensuring that the presence (or absence) of that user, with some probability, can not be determined by inspecting the output. This process necessarily involves a degree of randomness on the part of the computation, the result of which is a noisy output.

We define the distance between two inputs, or databases D_1 and D_2 as the number of records in which they differ, such that two databases are said to be neighbors if they differ in exactly one record. That is, D_1 could be made equal to D_2 by adding or subtracting exactly one record from either.

Moreover, a randomized computation M has epsilon differential privacy if for any two neighboring databases x and y,

$$Pr\left(M(x) \in S\right) \leq \exp(\varepsilon) Pr(M(y) \in S)$$

where S is any subset in the range of M. Finally, we define the query sensitivity to be the maximum value of $|M(x) - M(y)|$ over all pairs of neighboring databases (x, y) [7].

We define a database as the underlying set of ad impression logs belonging to one or more publishers. In the case of reach either a user is present in the database or not, which again is just the cardinality estimation problem, where the sensitivity of the sketches is 1. As it relates to Bloom filters this area has been well explored and indeed our work below applies BLoom-and-flIP (BLIP) [1], which is a well-known technique for making Bloom filters differentially private. We also build upon work by Stanojevic et al. [18] for estimating the cardinality of a union of noisy Bloom filters.

While the work around differentially private Bloom filers is well-established, we believe our work on the frequency problem to be new. Here each impression in the database is captured by the sketch, and therefore the user with the most impressions (i.e. the maximum frequency) defines the sensitivity of these queries. Moreover, since the data structure used to capture this information is not a bit array but an array of counts, it is not possible to apply the techniques for differentially private Bloom filters referenced above. Instead since this is a case of generalized discrete data, we turn to the work of Ghosh et al. [10], and employ a two-tailed geometric distribution for generating differentially private noise. That distribution is defined as follows:

$$Pr(X = \delta) = \tfrac{1-\alpha}{1+\alpha}\alpha^{|\delta|}$$

where X is a two-sided geometric random variable with domain $\delta \in (-\inf, \inf)$ and parameter $\alpha, 0 < \alpha < 1$.

Given that a user may be exposed to an ad campaign easily 10 or more times, the amount of noise that may be required is quite high. For example with $\epsilon = 1$ and a maximum frequency of 10 we have $\alpha = 0.0952$, which implies a distribution with some very long-tails. Indeed, some of our early investigations have shown that adding noise directly to frequency sketches quickly destroys utility.

For this reason we have chosen to use MPC to compute a frequency histogram from the raw sketches, which means that since any user is represented in any histogram at most once, we maintain a query sensitivity of only 1 rather than the maximum frequency. Moreover, the protocol is constructed so that noise is injected into the histogram as it is constructed. Thus after combining the sketches via MPC we are left with only a noisy frequency histogram, and none of the raw sketches are ever decrypted.

2.5 ElGamal Encryption

ElGamal encryption is a simple public-key encryption scheme that can easily be adapted to support threshold decryption. It can be viewed as an offline variant of the Diffie-Hellman key exchange protocol. The basic scheme, for a group with generator G and order q, is as follows:

- $KeyGen$: Choose a random $sk \in \mathbb{Z}_q$, and set $pk = G^{sk}$.
- $Enc(pk, m)$: Choose a random $R \in \mathbb{Z}_q$ and compute the ciphertext tuple (G^R, mpk^R).
- $Dec(sk, c)$: Compute $(G^R)^{sk}$ and output $m = (mpk^R)/(G^R)^{sk}$.

An N-of-N threshold version of this scheme can be constructed by simply multiplying all of the public keys together. A simple example of this threshold variant is as follows: two parties generate keys G^X and G^Y. If these public keys are multiplied we get G^{X+Y}, which can be used as a public key for encryption; the corresponding secret key is $X + Y$. Notice, however, that for decryption the original secret keys can be applied one-by-one (in any order), so no single party needs to know the joint private key.

ElGamal encryption has an additional useful property: it supports a multiplication homomorphism. Given two ciphertexts $(G^{R_1}, M_1G^{XR_1}), (G^{R_2}, M_2G^{XR_2})$, we can compute $(G^{R_1+R_2}, M_1M_2G^{X(R_1+R_2)})$, which will decrypt to the product of the two messages. Note that this is an additive homomorphism on the discrete logarithms of the messages; in other words, we could have used this to compute $(G^{R_1+R_2}, G^{M_1+M_2}G^{X(R_1+R_2)})$, which works for small message spaces (small enough to compute discrete logarithms efficiently).

For our application we use the additive homomorphism in the exponent of the messages as described above to combine counts. However, it is also necessary to join the set of sketches on their register IDs, for which we use use deterministic encryption (specifically, the Pohlig-Hellman cipher [16]), which allows for equality testing without exposing the register ID itself. This involves each worker choosing a secret exponent that is applied to all of the partially decrypted register IDs, and is equivalent to changing the generator G to some random group element. We also apply a secure hash to the IDs before applying the ElGamal cipher to ensure that the malleability of the ciphertext cannot be abused to determine the register IDs after deterministic encryption has been applied.

3 CascadingLegions Cardinality and Frequency Estimator

The number of bits required for Bloom filter accuracy grows linearly with the cardinality of the set that needs to be measured. We solve this by arranging registers (i.e. bits that record the presence of the item) of CascadingLegions (CL) into a two-dimensional array. Columns of the array are called legions and rows are called positions. Each item is thus mapped into a (legion, position) tuple, which is the register. The probability distribution over the legions is geometric and over the positions it is uniform, which means that essentially each legion is a single hash Bloom filter. This scheme allows estimation of the cardinality with a fixed relative error, and the required sketch size grows logarithmically with the cardinality of the set that needs to be measured. Moreover, as with Bloom filters, CL sketch registers can be combined via a bitwise-or operation. This sketch can also be extended to support frequency estimation by making each register a counter and combining sketches with a register-wise sum.

Algorithm 1 shows how a CL sketch is initialized. There are two parts, the allocation of an array of registers (s) and the allocation of an array of same-key aggregators (b). The same-key aggregator is akin to what was described above for FreqLogLog. When an item is initially inserted into register (r), the same-key aggregator stores a fingerprint of that item and s[r] is set to 1. Then the next time an item is allocated to that register, its fingerprint is compared to the existing fingerprint. If they are equal s[r] is incremented. If the new fingerprint is larger than the existing one, then the existing one is replaced and s[r] is reset to 1. Otherwise the new fingerprint is smaller and no modifications are made. The process for updating the same-key aggregator is part of Algorithm 3 below, which shows how items are inserted into a sketch.

Algorithm 1 Sketch Initialization

```
Input: Number of legions $l$, Number of positions $m$
Output: Sketch $(s, b)$
  $s \leftarrow$ Array of size $l * m$
  $b \leftarrow$ Array of size $l * m$
  for $s_i \in s$ do
      $s_i \leftarrow 0$
      $b_i \leftarrow SameKeyAggregator()$
  end for
```

Before proceeding with insertion we introduce the register allocation algorithm. To allocate an item to a register it is first hashed. Then the legion is assigned by determining the number of leading zeros in the hash value. Next, the first non-zero bit is stripped and the remaining bits, modulo the number of positions, determine the position. See Algorithm 2. Note that it is also straightforward to allocate items to legions using a binomial distribution by using the sum of active bits in the fingerprint as the way of determining the legion, in which case similar adjustments to the estimation methods below are also required.

Insertion, which is shown in Algorithm 3, proceeds by allocating the item to a register (i, j). The next step is to check the same-key aggregator as described above and adjust the register values accordingly. The algorithm assumes the use of counters, but in the cases where frequency is not desired, changing to a bit-array is a straightforward modification.

Algorithm 2 Register allocation

Input: Fingerprint f, Num legions l, Num positions m
Output: CascadingLegions register (leg, pos)

```
$leg \leftarrow 0$
**while** $f \mod 2 \equiv 0$ and $leg < l - 1$ **do**
    $leg \leftarrow leg + 1$
    $f \leftarrow f/2$
**end while**
$f \leftarrow f/2$
$pos \leftarrow f \mod m$
**return** $(leg, pos)$
```

Algorithm 3 Item insertion

Input: Item $\mathcal{I}$, Sketch (s, b)
Output: Updated sketch (s, b)

```
$f \leftarrow$ fingerprint($\mathcal{I}$)
$(leg, pos) \leftarrow$ allocateRegister($f$)
$r \leftarrow leg * m + pos$
**if** $b_r \equiv$ null or $b_r \equiv f$ **then**
    $s_r \leftarrow s_r + 1$
    $b_r \leftarrow f$
**else if** $b_r < f$ **then**
    $s_r \leftarrow 1$
    $b_r \leftarrow f$
**end if**
```

To estimate the cardinality of the set of items we first derive the expected number of registers that are activated by a certain cardinality. Since each legion is a uniform Bloom filter, then if n_j items fell into legion j then it is expected to have $1 - exp(-n_j/m)$ fraction of registers activated. And as legion ℓ is expected to be hit with $2^{-(j+1)}$ fraction of items we observe that the total number of registers activated is equal to:

$$F(t) = m \sum_{j=0}^{\ell} (1 - exp(\frac{-n}{m \cdot 2^{(j+1)}})) \tag{1}$$

Since $F(t)$ is monotonic, we can use binary search to efficiently find the cardinality t given the observed number of activated registers. See algorithm 4.

Algorithm 4 Cardinality estimation from CascadingLegions sketch.

Input: CascadingLegions sketch s

Output: Estimated cardinality of the set stored in s

```
  Define $F(t)$ = Equation 1
  $c \leftarrow$ Count non-zeros in $s$
  $t \leftarrow$ BinarySearch for $t = F(c)$
  return $t$
```

For frequency estimation the distribution of counts is extracted from the registers and the frequency distribution is estimated from the sample.

Finally, it is worth noting the resemblance of the CascadingLegions and PCSA sketch structures [9], which are essentially transposes of one another, however the CascadingLegions estimator is compatible with MPC and differentially private noise, whereas PCSA to the best of our knowledge, is not.

4 Differentially Private Cardinality Estimation

Following the results of Allagan et al. [1], we can add ϵ-differential privacy to CL sketches by flipping each bit in the sketch with probability $p = \frac{1}{1+exp(\epsilon)}$. For example, given an epsilon of $\log(3)$ we should flip bits with probability 0.25, where flipping a bit constitutes XORing its current value with 1.

In Algorithm 5 we show how given a list of sketches S, all of which had bits flipped with probability p, that we can estimate the number of 0 registers in the union by removing noise in aggregate.

We could apply this strategy to the whole sketch, but since the sketch is expected to have many fully saturated and empty legions, the error of such estimation would be large. Therefore we apply the de-noising to each legion separately and use the legion that's in the process of saturation, which we call the *Golden Legion*, to do the estimate.

We use a heuristic such that the first legion with less than 60% of its registers activated is declared to be the Golden Legion. The rationale for this is that if too many registers are saturated then the number of newly activated registers with increased cardinality changes slowly, which leads to a high level of noise in the cardinality estimation. On the other hand, legions with too little saturation are sampling too few items. Formally, the legion that maximizes the expected number of incremental activated registers given the incremental cardinality should be used. We have observed empirically that the algorithm has sufficient accuracy with the 60% threshold, but we plan to do formal analysis of optimal selection of the set of Golden Legions in the future.

Algorithm 5 Estimating cardinality of the union, using GoldenLegion of noised CascadingLegion sketches

Input: a list of CascadingLegions sketches S that had bits flipped with probability p

Output: estimated cardinality of the union of sets stored in S

```
  $InverseF \leftarrow InverseF$ function from Algorithm 4
  procedure OneCountsVolumesVector($S, j$)
      $v \leftarrow [0] * len(S)$
      for $i$ in $range(m)$ do
          let c = number of sketches in S that have 1 in i-th position of j-th legion
          let v[c] += 1
      end for
      return $v$
  end procedure
  procedure TransitionProbability($a, b$)
      # This function is quite technical and is omitted here.
      return probability of a vector with $a$ ones to have $b$ ones
         after bits flipping with probability p
  end procedure
  procedure CorrectionMatrix:
      $N \leftarrow$ a matrix of transition probabilities, i.e.
         $N(a, b) \leftarrow$ TransitionProbability($a, b$)
      return linear algebra inverse of $N$
  end procedure
  for $g$ in range($l$) do
      $v \leftarrow$ OneCountsVolumesVector($S, g$)
      estimated zero count $\leftarrow$ first coordinate of  CorrectionMatrix$*v$
      if NOT $j < l - 1$ and estimated zero count $< 0.6 * m$ then
          return InverseF($m -$ estimated zero count)
      end if
  end for
```

5 Using continuous exponential distribution

The CascadingLegions sketch presented above is straightforward and achieves estimations of cardinality with fixed relative error. However, it has a few properties that are aesthetically displeasing, which may lead to unnecessary friction when the algorithm is applied. It is an arbitrary decision for legions to have a geometrically decreasing allocation probability with parameter 1/2. This discretization also requires the formula F for the expected number of legions to have a summation with l terms. Moreover, in the differentially private version, this decision required an abrupt switch between the Golden Legion and adjacent legions. These drawbacks are all alleviated in the continuous version of the algorithm that we call LiquidLegions and present in this section.

We can instead allocate items to registers using a geometric distribution with a varying decay factor. To be able to use this idea we need to find a way to do this allocation efficiently and to estimate the cardinality from the observed number of activated registers.

Consider a bounded geometric distribution

$$P(i) = \frac{q^i \cdot (1-q)}{1-q^m}, i \in \{1, \dots, m\} \ .$$

To sample an element from it naively we would need to run a for-loop that potentially goes up to m, however to perform this sampling efficiently and simplify the analysis, we will instead be using an exponential distribution

$$P(x) = \frac{ae^{-ax}}{1-e^{-a}}$$

truncated to segment [0, 1], with a resolution m.

To allocate an item to a register we split the segment [0, 1] into m segments of equal length, assuming that i-th segment corresponds to i-th register. Then we sample a real valued number from the exponential distribution and allocate the item to the register corresponding to the interval in which the real number fell.

It is well known that to sample a number from a real valued distribution we can sample a number from the segment [0, 1] uniformly and apply an inverse CDF. The inverse CDF of the truncated exponential distribution above is equal to:

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

Thus we arrive at Algorithm 6 for allocating an item to the LiquidLegions sketch.

Otherwise, the algorithms for creation and insertion are identical to those used for CascadingLegions with the exception that we take the number of positions per legion to be 1.

Algorithm 6 LiquidLegions register allocation

```
INPUT:item to allocate, number of positions $m$
OUTPUT:Zero-based index of LiquidLegions register
  $f \leftarrow \text{fingerprint64bit}(item)$
  $u \leftarrow f/2^{64}$
  $x \leftarrow 1 - \log(\exp(a) + u * (1 - \exp(a)))/a$
  return $\lfloor x * m \rfloor$
```

For cardinality estimation we need a function that maps cardinality to the expected number of registers activated r. This function is $t = c/m \to r/m$, and it can be obtained for an arbitrary distribution P via integration.

$$E(t) = 1 - \int_0^1 e^{-P(x)t} \mathrm{d}x \tag{2}$$

This formula can be obtained by writing the probability of the i-th register being allocated by at least one item and going to the limit when m goes to infinity. At Google we have initially observed this when developing methodology for efficient Reach model application [17]. The integral for the exponential distribution can be expressed via an exponential integral function as follows.

$$\begin{aligned} E(t) &= 1 - \int_0^1 e^{-P(x)t} \mathrm{d}x \\ &= 1 - \int_0^1 e^{-\frac{ae^{-ax}}{1-e^{-a}}} \mathrm{d}x \\ &= 1 - \frac{\mathrm{expi}(-\frac{at}{-1+e^a}) + \mathrm{expi}(-\frac{ae^a t}{-1+e^a})}{a} \end{aligned} \tag{3}$$

We thus arrive at the algorithm for cardinality estimation. It is again identical to its CascadingLegions counterpart, except for the replacement of the cardinality function.

Algorithm 7 Inverse cardinality estimation function from LiquidLegions sketch

```
  procedure ExpectedLegionaries($t$)
      return $1-(-expi(-a*t/(exp(a)-1))+expi(-a*exp(a)*t/(exp(a)-1)))/a$
  end procedure
```

To make the LiquidLegions sketch differentially private we then flip each bit with probability $p = \frac{1}{1+exp(\epsilon)}$. The Golden Legion in this case will be a continuous segment of registers. We have empirically observed that setting the length of the Golden Legion to

$$\tilde{m} = \min(m, m \cdot \log(10))/a$$

leads to substantial improvement in accuracy over CascadingLegions. The intuition of this length is that the last register of the Golden Legion is made to have probability 10 times smaller than the first register, and thus we can find the location before which almost all registers are saturated and after which almost no registers are saturated. We leave it to future work to accurately find the optimal length of the Golden Legion for the LiquidLegions algorithm.

Notice that for any x_0 registers at positions $\{x_0, \dots, x_0 + \tilde{m} - 1\}$ constitute another LiquidLegion with $\tilde{m}$ registers and exponent of $\tilde{a} = \frac{a \cdot \tilde{m}}{m}$.

We find the position of the Golden Legion by maximizing the probability that the next item added to the set flips a register of the clean sketch inside the Golden Legion. It is easy to see that for cardinality t this probability is equal to

$$p(x_0) \cdot \left(\frac{dE_g(\tilde{t}, x_0)}{d\tilde{t}} \Big|_{\tilde{t} = t \cdot p(x_0)} \right),$$

where

$$p(x_0) = \frac{e^{\frac{-x_0 \cdot a}{m}} - e^{\frac{-(x_0 + \tilde{m}) \cdot a}{m}}}{1 - e^{-a}}$$

is the probability of an item falling between registers x_0 and $x_0 + \tilde{m}$, while

$$\tilde{E}(\tilde{t}, x_0) = 1 - \frac{\text{expi}(-\frac{\tilde{a}\tilde{t}}{-1+e^{\tilde{a}}}) + \text{expi}(-\frac{a e^{\tilde{a}} \tilde{t}}{-1+e^{\tilde{a}}})}{\tilde{a}}$$

is the expected number of registers activated in the Golden Legion starting at position x_0 by $\tilde{t}$ items.

Empirically we observed that such position can be approximated by picking such position that about one half of the registers $\{x_0, \dots, x_0 + \tilde{m}\}$ are activated. Which is consistent with the heuristic used for CascadingLegions.

5.1 Any Distribution Bloom Filters

Note that the in the previous subsection we did not use any specific properties of the exponential distribution. Thus all the reasoning can be applied analogously to an arbitrary distribution. Specifically, given a continuous distribution over segment $[0, 1]$ with probability density function $P(t)$ and cumulative distribution function $F(t) = \int_0^t P(x)dt$ we can define a Bloom filter as follows.

- Sample a register index from the range $\{0, 1, \dots, m\}$ using formula $\lfloor F^{-1}(u) \rfloor$, where u is a random uniform variable from segment $[0, 1]$
- Estimate cardinality given n registers activated using formula $E^{-1}(n)$, where

  $$E(t) = 1 - \int_0^1 e^{-P(x)t} \mathrm{d}x$$

  is the expectation of the number of registers activated for cardinality t.

We chose to use an exponential distribution, as it achieves a fixed relative error regardless of the cardinality. Uniform bloom filters achieve minimal error (maximal accuracy) on the small cardinalities. One may choose a different distribution trading accuracy for certain cardinalities in favor of others.

If you use a distribution for which the integrals used in this section can be computed analytically, then that closed form solution should be used. Otherwise the distribution can be approximated with a piece-wise uniform distribution. See [17] on details of computing such integrals. Note that the piece-wise uniform distributions correspond to Dirac Mixture activity density functions in the terminology of [17].

In the extreme such an approximation can be done assuming each individual register is a distinct uniform distribution. The disadvantage of this non-parametric approach is the additional computation costs of computing the cardinality estimation, as a repeated summation over all registers is required when executing the binary search in Algorithm 4.

6 Differentially Private Frequency Estimation

The methods for differentially private sketches presented in the previous section, while promising for cardinality estimation have proven to break down for frequency estimation. Indeed, early experiments have shown that encoding frequency directly into just a single sketch results in very poor estimates. Other techniques that attempt to use multiple sketches, for example one for each frequency, also appear unworkable. This is due to the fact that after unioning less than ten noisy sketches, results tend to be both biased and have high variance. Given these poor results and our interest in estimating frequencies of up to 10 across as many as 100 publishers, we turn to multiparty computation methods as a solution.

The first subsection presents a cryptographic technique to replace the same-key aggregator discussed above, and the following subsection describes a protocol for computing differentially private reach and frequency estimates across an arbitrary number of publishers.

6.1 Cryptographic Same-Key Aggregator

The cryptographic same-key aggregator relies on homomorphic encryption to filter out registers to which more than one user contributed, thereby allowing us to obtain a clean sample of user frequencies. For any given register we wish to compute the sum of the counters where all keys are the same, and otherwise we would like the sum to become a random number. To accomplish this we track a flag such that when decrypted reveals nothing more than whether the counter was destroyed by randomness.

We begin with several tuples that we wish to combine:

$$[(E(C1), E(K1)), E(C2), E(K2)), \dots].$$

The overall list can be thought of as the values for a single register where each tuple is the value that is contributed by a particular publisher. Algorithm 8 shows how to construct the encrypted sum of the counters and a flag, E(IsDestroyed), indicating whether all of the keys are identical, which is only the case when the flag is equal to its initial value. This value can be any well-known constant value.

Algorithm 8 Homomorphic Same Key Aggregator

INPUT:$tuples$ is a list of tuples of encrypted counter and encrypted key of the form $(E(C), E(K))$
OUTPUT:$(E[c], E[sumofcounters])$ if all keys are the same; $(E[R1], E[R2])$ if any key is different where $R1$ and $R2$ are random numbers

```
$E[IsDestroyed] \leftarrow E[c]$
$E[CountFinal] \leftarrow tuples[0][0]$
$E[K1] = tuples[0][1]$
for $E[C], E[K]$ in $tuples[1:]$ do
    $R \leftarrow random()$
    $Destructor \leftarrow E[R] * (E[K] - E[K1])$
    $E[IsDestroyed] \leftarrow E[IsDestroyed] + Destructor$
    $E[CountFinal] \leftarrow E[CountFinal] + E[C] + Destructor$
end for
return $(E[IsDestroyed], E[CountFinal])$
```

Note that when merging buckets across more than two sketches we will destroy buckets when any single bucket disagrees on the key, but this should not be a problem for higher order legions and is irrelevant for lower order legions. This is different from the same key aggregator presented above, and loses some fidelity when compared to the max function used in the unencrypted case, however we have not found this to be a problem in practice.

6.2 MPC Protocol

As shown above, sketches with the same parameterization can be unioned by summing them register-wise. We make use of this fact to build a lightweight MPC protocol for computing a joint sketch, using the homomorphic properties of ElGamal encryption as the core operation of our protocol. The high-level approach is to first set up an N-of-N threshold key, and then have each of the sketch providers encrypt their inputs using the public key. Each secret key shareholder will then partially decrypt the joint sketch, apply a deterministic cipher using the homomorphic properties, and forward the result to the next shareholder. Once all decryption key shares have been used, the result will be a deterministically encrypted sketch, from which the desired statistics can be computed. We describe the protocol in detail below, including how a differentially private output can be computed.

In our application we anticipate that the number of sketch providers will be large. We also expect that most sketch providers will be too resource-constrained to fully participate in the MPC protocol as peers. Therefore we opt for a semi-honest worker model where a small number of independently operated workers run the protocol and the majority of sketch providers send encrypted inputs to one of the workers. This model has been deployed previously in other MPC applications [3]. Among the workers the communication graph follows a ring topology.

To begin each worker generates an ElGamal key pair (pki, ski) as described above. Each worker then broadcasts its key to each of the other workers. After this it is possible for each of the workers to form the combined public key and then the sketch providers can fetch the key from any of the workers. This combined public key will then be used by sketch providers to encrypt their sketches.

Next each of the sketch providers do the following:

1. Retrieve the combined public key from the workers.
2. Create a sketch that includes bucket fingerprints.
3. Package the sketch into a sparse representation where each non-zero register is represented as a three-tuple of $(SHA256(register), value, fingerprint)$.
4. Apply ElGamal encryption to each three-tuple with the combined public key.
5. Send the encrypted sparse sketch to the first worker.

Note that it's possible to generalize to having sketch providers send their sketches to a random worker and then having each worker shuffle their received sketches before sending them on to a single worker that commences with the protocol. To simplify our exposition we proceed assuming that all sketch providers send sketches to a single worker. Also note that the simple description for publisher sketch transmission above will leak each publisher's cardinality to the receiving worker. This issue is addressed below.

Once the first worker has received the encrypted sparse sketches of each publisher the protocol begins and consists of the following steps:

1. For all three-tuples from all publishers, subsequently referred to as just *three-tuples*, the first worker shuffles the three-tuples and transfers them to the next worker.
2. The next worker then performs the following steps:
   (a) It uses its ElGamal key share to partially decrypt each register id.

   (b) It applies a layer of deterministic encryption to each register id using the ElGamal homomorphism.

   (c) It shuffles the three-tuples.

   (d) It passes the three-tuples to the next worker.

3. This process continues until the first worker once again receives the three-tuples.

4. Then the first worker uses its key share to decrypt the three-tuples and joins the tuples on the now deterministically encrypted register IDs, which for each position results in the following structure

$$(E_{det}(r), [(E(c_1), E(f_1)), (E(c_2), E(f_2)), \ldots])$$

   where c_i and f_i are the count and the fingerprint of the i-th register with the register id r.

5. The first worker then combines the encrypted (value, fingerprint) tuples using the same-key aggregator method described above. This results in a combined three-tuple $(E_{det}(r), E(count), E(flag))$ for each unique register ID in the combined sketch.

6. From here the first worker initiates a second round of communication by forwarding the combined three-tuple to the next worker.

7. The next worker then performs the following steps:

   (a) It decrypts the count.

   (b) It decrypts the flag.

   (c) It shuffles the three-tuples.

8. This process continues until the first worker receives the set of combined three-tuples.

9. From here it's possible to obtain an estimate of cardinality. Technically this can be done after step 4, but once we include the updates below for making the protocol differentially private, it is necessary to place it here.

10. Next the workers collaborate to determine the frequency distribution:

   (a) The first worker finishes decrypting the flag and discards all tuples whose flag is not equal to the well-known same-key aggregator constant.

   (b) From here a frequency histogram can be constructed in order to estimate the true frequency distribution of the sketch.

Differential privacy can be added to the frequency histogram directly by generating a two-tailed geometric random variable for each histogram bin and adding it to that bin. The query sensitivity is one, which must be the case because any user is represented in the frequency histogram at most once. The downside of adding noise here is that the workers learn the true histogram and could choose to leak it.

An approach to adding noise that avoids this leakage involves distributing noise generation as part of the MPC protocol, however to do this we need a procedure to sample random variables whose sum will be a two-tailed geometric random variable. The details of this construction are currently being published by a colleague and will be properly cited in a subsequent version of this paper.

To add the noise the workers begin by agreeing on an arbitrary baseline, B, to be added to each bucket. Steps 1 and 2 of the above protocol are extended to have each worker draw max_frequency appropriately parameterized Polya random variables $(X_1, ..., X_i, ..., X_{max\_freq})$. These represent the number of noise values to add to the computation for the values $[1, max\_frequency]$ for the particular worker. To achieve this each worker adds $B - X_i$ tuples with value i to the sketch. These newly added registers will have a random register ID that is outside of the bounds of valid register IDs and a random fingerprint. This is done in order to prevent the noise registers from interfering with the same-key aggregator. Then in step 9, the first worker can subtract the value $B * W$ from each frequency histogram bucket, where W is the number of workers. This then leaves us with the required noise distribution. It also means that cardinality can only be estimated after the noise baseline has been subtracted. See algorithm 9 for a detailed look at how each worker adds noise to the computation.

One complication with this approach is that the maximum frequency that must be reported is far lower than the theoretical maximum frequency, which means that if the frequency histogram could be truncated that less noise would have to be injected into the final result. To accomplish this we first determine the maximum frequency, which is done by having each publisher cap their reported frequency and by knowing the number of publishers participating in the union. Then while assembling the frequency histogram the first worker sums the frequency histogram buckets greater than or equal to the maximum reportable frequency into a single bucket. After this each worker can report the amount of noise it added in aggregate to all buckets greater than maximum reportable frequency and these values can be subtracted from the final histogram bucket. An unfortunate result of this entire operation is that the size of the sketch is increased by a factor of the number of sketches being unioned. Finding a cheaper way to drop registers that have counts greater than the maximum reportable frequency is an open research topic.

One observation about the protocol described above is that it leaks the cardinalities of individual sketch contributors. There are several possible ways to solve this. One way is to use a pseudo-dense representation. In this case the zero-valued registers are encrypted, but all are given single invalid well-known register ID, which of course once encrypted, are indistinguishable from any valid register value and each other. This effectively pads the input sketch, but using invalid register IDs prevents these dummy registers from interfering with

Algorithm 9 Worker Distributed Noise

```
$noise\_tuples \leftarrow []$
for $v \in [0, max\_frequency]$ do
    $x \leftarrow drawPolyaRv()$
    for $i \in range(B - x)$: do
        noise_tuples.append((random(), v, random()))
    end for
end for
```

the same-key aggregator. One additional step is also required in the protocol, which is to deterministically encrypt the well-known register value and filter it out before estimation occurs. This can be done after the first round of communication once noise registers have been added by each worker. Computationally it is desirable to use less padding, but it is an open question whether less padding can adequately preserve privacy.

Another observation about the protocol is that after joining on register IDs it is possible to determine the number of publishers that contributed to each register. Unfortunately we have not been able to find a good way to hide this. In particular it is not possible to send truly dense sketches since this will break the same-key aggregator. This is an area of continued research for us, which could include either modifications to the protocol or the development of other methods for estimating the frequency histogram that do not require the same-key aggregator.

7 Results and Conclusions

A subsequent version of this paper will present simulation results and comparisons to standard cardinality and frequency estimation techniques.

References

[1] Alaggan, M., Gambs, S., and Kermarrec, A.-M. Blip: Non-interactive differentially-private similarity computation on bloom filters. In *Stabilization, Safety, and Security of Distributed Systems* (Berlin, Heidelberg, 2012), A. W. Richa and C. Scheideler, Eds., Springer Berlin Heidelberg, pp. 202–216.

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

[3] Bogetoft, P., Christensen, D. L., Damgård, I., Geisler, M., Jakobsen, T. P., Krøigaard, M., Nielsen, J. D., Nielsen, J. B., Nielsen, K., Pagter, J., Schwartzbach, M. I., and Toft, T. Secure Multiparty Computation Goes Live. In *Financial Cryptography* (2009), pp. 325–343.

[4] Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., and Seth, K. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 -*

November 03, 2017 (2017), B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, pp. 1175–1191.

[5] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.

[6] Desfontaines, D., Lochbihler, A., and Basin, D. Cardinality estimators do not preserve privacy. *Proceedings on Privacy Enhancing Technologies 2019*, 2 (2019), 26 – 46.

[7] Dwork, C., and Roth, A. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science 9*, 3–4 (2014), 211–407.

[8] Flajolet, P., Fusy, É., Gandouet, O., and Meunier, F. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms* (Juan les Pins, France, June 2007), P. Jacquet, Ed., vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) of *DMTCS Proceedings*, Discrete Mathematics and Theoretical Computer Science, pp. 137–156.

[9] Flajolet, P., and Martin, G. N. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)* (1983), pp. 76–82.

[10] Ghosh, A., Roughgarden, T., and Sundararajan, M. Universally utility-maximizing privacy mechanisms. *SIAM Journal on Computing 41* (12 2008).

[11] Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1987), STOC '87, ACM, pp. 218–229.

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

[13] Heule, S., Nunkesser, M., and Hall, A. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology* (New York, NY, USA, 2013), EDBT '13, Association for Computing Machinery, p. 683–692.

[14] Ion, M., Kreuter, B., Nergiz, A. E., Patel, S., Raykova, M., Saxena, S., Seth, K., Shanahan, D., and Yung, M. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. Cryptology ePrint Archive, Report 2019/723, 2019. https://eprint.iacr.org/2019/723.

[15] Lindell, Y. *How to Simulate It – A Tutorial on the Simulation Proof Technique*. Springer International Publishing, Cham, 2017, pp. 277–346.

[16] Pohlig, S., and Hellman, M. An improved algorithm for computing logarithms over gf (p) and its cryptographic significance (corresp.). *IEEE Transactions on information Theory 24*, 1 (1978), 106–110.

[17] Skvortsov, E., and Koehler, J. Virtual people: Actionable reach modeling. Tech. rep., 2019.

[18] Stanojevic, R., Nabeel, M., and Yu, T. Distributed cardinality estimation of set operations with differential privacy. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)* (2017), pp. 37–48.

[19] Swamidass, S. J., and Baldi, P. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling 47* (05 2007), 952–64.

[20] Yao, A. C. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1982), SFCS '82, IEEE Computer Society, pp. 160–164.