UNIVERSITY OF CALIFORNIA

Los Angeles

Private Computations on Streaming Data

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Kevin Matthew Garbe

2024

ABSTRACT OF THE DISSERTATION

Private Computations on Streaming Data

by

Kevin Matthew Garbe
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2024
Professor Rafail Ostrovsky, Chair

We present a framework for privacy-preserving streaming algorithms which combine the memory-efficiency of streaming algorithms with strong privacy guarantees. These algorithms enable some number of servers to compute aggregate statistics efficiently on large quantities of user data without learning the user's inputs. While there exists limited prior work that fits within our model, our work is the first to formally define a general framework, interpret existing methods within this general framework, and develop new tools broadly applicable to this model. To highlight our model, we designed and implemented a new privacy-preserving streaming algorithm to compute heavy hitters, which are the most frequent elements in a data stream. We provide a performance comparison between our system and Poplar, the only other private statistics algorithm which supports heavy hitters. We benchmarked ours and Poplar's systems and provided direct performance comparisons within the same hardware platform. Of note, Poplar requires linear space compared to our poly-logarithmic space, meaning our system is the first to compute heavy hitters within the privacy-preserving streaming model. A small memory footprint allows our algorithm (among other benefits) to run efficiently on very large input sizes without running out of memory or crashing.

The dissertation of Kevin Matthew Garbe is approved.

Raghu Meka

Todd D. Millstein

Amit Sahai

Rafail Ostrovsky, Committee Chair

University of California, Los Angeles

2024

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Rafail Ostrovsky, for his support and guidance throughout my degree, and especially for this work. I would also like to thank Prof. Vladimir Braverman for his insight and feedback into streaming algorithms for this work.

VITA

2012	Wrote *Patterns in the Coefficients of Powers of Polynomials Over a Finite Field* as part of the Research Science Institute at MIT
2014	Wrote *Random Walks on Products of Cyclic Groups* as part of the Stanford University Research in Mathematics Program
2016	Teaching Assistant for Automata and Complexity at Stanford University
2017	Published honors thesis *Concentration Bounds for Functions of Negatively Dependent Variables* advised by Prof. Jan Vondrák
2017	B.S. with honors in Mathematics, and M.A. in Computer Science (Theory and Artificial Intelligence), Stanford University
2017-2019	Full time Software Engineer at Google
2019	Received NSF Fellowship
2021	Published *Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares*, later presented at CRYPTO 2022 PPML
2021-2022	Teaching assistant for: Introduction to Algorithms and Complexity, Introduction to Cryptography, and Data Management Systems
2019–Present	Student and research assistant at UCLA advised by Professor Rafail Ostrovsky

CHAPTER 1

Introduction

A typical modern approach for data analysis involves a centralized server gathering a large amount of user data in order to compute data analytics via aggregate functions over the data. For example, determining popular websites from browsing history or popular locations from phone telemetry data can help in targeting advertising strategies. From the perspective of the server working with the data, the aggregation of the data needs to be efficient. In particular, for large-scale applications it is often infeasible to store all of the data received. Streaming algorithms are able to compute such statistics in a memory-efficient manner by performing on the fly calculations as data is "streaming" in, keeping the necessary storage low. From the perspective of the users providing the data, the process should respect their privacy. The server should not have access to an individual user's inputs, but should still be able to extract the relevant aggregate statistic from the overall set of user data. Secure multiparty computation (MPC) strategies achieve user privacy, as they allow the users to split their data across two or more servers in such a way that servers may jointly compute the desired statistic on the split inputs without ever learning the original value, relying only on the assumption that these servers do not collude with one another to reconstruct client data.

Privacy-preserving streaming algorithms achieve these joint objectives by computing aggregate statistics on data in a memory-efficient manner without revealing any individual inputs. Although some prior works have achieved this goal for specific statistics, the overall framework has not yet been well-defined. In this work, we formally define the Privacy-Preserving Streaming model of algorithms. This includes a general layout which such algorithms may follow (and have followed in the past) as well as the formal efficiency and security properties they should satisfy. We also discuss methods and tools for efficiently converting a general streaming algorithm into a privacy-preserving streaming algorithm. Furthermore, investigate the benefits of precomputed correlated randomness within our model. Precomputed randomness is a powerful tool in multiparty computation generally, and we examine how it can be adapted to be streaming-efficient. To this end, we develop a new algorithm for efficiently validating daBits [RW19], a type of correlated randomness which is particularly useful in our setting, using a new variant of share multiplication. We also analyze some prior works in the space and evaluate the extent to which they match our new model.

Using this framework, we design a novel algorithm for identifying the most frequent items in a data stream, known as heavy hitters. Prior heavy hitter results are either streaming algorithms that are not private [BCI16, BCI15], are privacy-preserving algorithms which do not qualify as streaming algorithms due to requiring linear space [BBC21] , or are distributed with multiple rounds of communication [HKR12]. We compare implementations of our new algorithm to the privacy-preserving system Poplar [BBC21] on cloud servers to show that computing the top K heaviest hitters via our algorithm requires only polylog space compared to the linear space required by Poplar. We also show that there are no drawbacks in terms of time efficiency. Privacy-preserving heavy hitters is a key building block toward enabling more privacy-preserving streaming algorithms and we consider it to be a major stepping stone toward that goal.

Contributions. In this paper, we:

1. formally define a new model of privacy-preserving streaming algorithms for computing statistics over large data sets in a space-efficient manner while maintaining user privacy,
2. create a new algorithm for batch validating daBits [RW19] for use in privacy-preserving streaming algorithms,
3. create a new privacy-preserving streaming algorithm for finding the most frequent items of a data stream, with significant space improvements over previous privacy-preserving algorithms [BBC21] for the same function.

CHAPTER 2

Technical Overview

In this chapter we briefly overview the remaining sections of the paper.

Streaming Primitives: A streaming algorithm computes a function $f(x_1, \ldots, x_m)$ on a stream of m inputs $x_1, \ldots, x_m$. The values are in the domain of $x_i \in [[n]]$, where $[[n]]$ denotes the set $\{0, \ldots, n-1\}$. The stream is a sequence of inputs provided by a collection of clients/users, and is sent to a server that wants to compute f on the client data. The core goal of streaming algorithms is to be able to do this efficiently, namely without storing all inputs of the stream at once, instead processing the stream using only polylog total space. This can come at the cost of accuracy, which can be parameterized and minimized. The streaming algorithm we focus on is identifying heavy hitters (the most frequent values), specifically the K most frequent. We discuss several results from prior works that help with our result. Similar to other heavy hitter algorithms [BCI16, BCI15], we have a similar building block where values are classified by value into two buckets, so that the "larger" bucket is more likely to have the heaviest value, and we use prior results to help quantify this probability. To determine how good a found value is, we use the CountMin [CM05] frequency estimator to determine which of those we found are approximately the most frequent.

Cryptography Primitives: Secret sharing (specifically, threshold secret sharing) is a method that allows a private value x to be split into n shares such that any subset of t shares can reconstruct the value, but any smaller subset reveals nothing about x (i.e., is indistinguishable from random). For this work we have two servers, so private values are split into two shares such that both shares together recover the value, but any share on its own is indistinguishable from random so reveals nothing. We use $[x]$ to denote a secret share of the value x. Specifically, Server 0 has the share $[x]_0$ and server 1 has $[x]_1$, and neither knows x itself. When the subscript is excluded, it is not referring to a specific server's share. This can also represent more complex values, for example $[x+y]$ denotes shares of the secret value "$x + y$".

We use two types of secret shares, arithmetic and Boolean. Arithmetic shares $[x]^A$ satisfy $x = [x]^A_0 + [x]^A_1 \pmod N$ for some λ bit prime modulus N. The modulus is public and constant, so will be implied unless necessary. Boolean shares $[x]^B$ satisfy $x = [x]^B_0 \oplus [x]^B_1$, where $\oplus$ is Boolean XOR applied bitwise. As explored in Prio+ [AGJ21], Boolean shares are useful for client shares because they provide length validation, while arithmetic shares are generally more useful for accumulation. Hence, we use the Boolean to Arithmetic (B2A) share conversion protocol in Prio+ [AGJ21], which allows servers to efficiently convert shares.

Privacy preserving streaming model: We describe a new model that combines the efficiency of streaming algorithms with privacy guarantees of secure multi-party protocols. The goal is to provide a framework for algorithms that take in a large quantity of secret shared user input and want to efficiently compute statistical functions on them without breaking privacy and learning any individual inputs. We describe the general flow of such algorithms and the properties they should satisfy to be streaming efficient and privacy preserving. Furthermore, we discuss trade-offs that pure streaming algorithms must make in order to be able to efficiently add privacy. While there are generic methods to make MPC protocols out of streaming algorithms they suffer from overhead complexity, so we describe ways that streaming algorithms can be designed with privacy directly in mind. Most notably, algorithms should be input oblivious, doing the same computations and operations regardless of what the input value is. This is because gaining efficiency through e.g. stopping computation early on some values would reveal information to the servers about the value depending on what computation path was taken. This is the inspiration for our top K algorithm, which draws on prior streaming algorithm ideas, but also makes algorithmic trade-offs to work well as a private algorithm.

Correlated Randomness: Correlated randomness are values that can be computed "offline" before any inputs are received, to efficiently do computations on shares such as conversion [AGJ21] or multiplication [Bea92]. However, the standard approach to correlated randomness is to generate all necessary values beforehand, which scales linearly in the number of inputs and so does not work well in a streaming scenario with many inputs. Therefore, we discuss several methods of generating correlated randomness in a space-efficient manner in batches, for use in the private streaming framework. The first is for servers to generate the next batch of necessary randomness in parallel with processing a batch of client inputs using the current batch of randomness. This is dependent on the complexity of generating the randomness, which does not necessarily work efficiently in parallel. The second is for clients to send the correlated random shares themselves, which the servers can efficiently validate the entire batch in bulk. For this second method, we created a new algorithm for validating daBits [RW19], similar to prior works [BFO11] for batch validation of multiplication triples. This algorithm also uses new modified alternative multiplication triples that allow for more efficient share multiplication in a specific context.

Top K: We introduce a new private streaming algorithm for approximately identifying the top K most frequent items in a data stream. Following the private streaming algorithm framework, an input oblivious streaming algorithm is built up from progressive building blocks, for more efficient conversion to a privacy preserving algorithm. At a high level, the Pair algorithm splits the stream into two buckets and identifies the larger bucket. Single-Heavy then, given a stream with a single heavy value, uses multiple bucket Pairs in parallel to obtain enough information to identify the single heavy value. For multiple heavy values, the stream is split into multiple parallel substreams to try and have each likely to have exactly one heavy value, and this is repeated multiple times. Finally, to account for different possible distributions where the heaviest values may not be significantly heavy, the full top K algorithm repeats this multiple times in parallel, each time removing some of the values at random so that any surviving heavy values are more likely to stand out. This produces a list of candidates from all of the SingleHeavy sub-algorithms, which are sorted by their CountMin frequency estimate to obtain the K best. This input oblivious design then allows for the protocol to be relatively easily converted into one using secret shares, to efficiently take constant rounds.

Practical Comparison: The closest algorithm we can compare our top K against is Poplar [BBC21]. Poplar also has clients split inputs into shares to two servers in a data stream, and the servers identify the most frequent items based on a threshold. However, Poplar stores the client shares until the end, reading over them multiple times during evaluation and so requires linear space, compared to the streaming efficient polylog space of this algorithm. We compare our implementation against the Rust implementation of Poplar, benchmarked on the same setup of AWS servers in different regions acting as the clients and two servers. We found that as expected, TopK required significantly less space on large numbers of inputs, and had significantly faster evaluation of the final inputs. In return, as expected Poplar had faster accumulation (live handling of inputs) due to just storing values. We found that together the end-to-end times were comparable, meaning TopK's per-input runtime cost during accumulation is comparable to Poplar's per-input runtime cost during evaluation. Most notably, on 5 million inputs, Poplar always crashed due to memory issues, while TopK was able to handle it properly, showing a concrete benefit of the space improvements.

CHAPTER 3

Primitives and Background

We introduce the streaming and cryptography background and primitives used for our work. This covers general definitions, and also specific prior results and protocols used.

3.1 Streaming background

Streaming algorithms compute a function $f(x_1, \ldots, x_m)$ on a stream of m inputs $x_i \in [[n]]$. We let $L = \lceil \log_2 n \rceil$ to be the number of bits to represent each element, i.e. the size of each input. This means that storing all values takes $O(m \log n)$ space, which is unwieldy for massive m, such as a million phones sending hundreds of inputs each. Therefore, a key component of streaming algorithms is to be polylog in the input size, where polylog is defined as $polylog(n) = O((\log n)^c)$. Since typically there are many more items than item size ($m >> L = \log_2 n$), we are primarily concerned with the impact of m on the space complexity.

We focus on 1-pass algorithms that pass over the data stream once. This means that the servers take in inputs, and can cache a polylog number (typically used for batching inputs), but once it discards its input it no longer can receive it again. This reflects the clients sending all relevant info (such as their input) in one single message and not needing to send anything more or otherwise interact with the server.

Many streaming algorithms are approximation algorithms due to limited space, because exact answers may not be able to be computed efficiently. For example, being able to return the exact frequency of all elements requires storing the frequency of every value, which requires space that is exponential in input size, for the worst case of every item appearing once. Generally, an approximation algorithm returns some answer y' that with probability at least $1-\delta$ is within some range ϵ of the true answer $y = f(\{x\})$, such as $(1-\epsilon)y < y' < (1+\epsilon)y$.

Heavy hitters is the general problem of identifying the most frequent items in a data stream. There are multiple ways of doing this, depending on the desired accuracy guarantee and/or goal. For example, some prior algorithms [BCI15, BCI16] return a set of items which with high probability are guaranteed to include all items past some parameterized frequency threshold, and none less frequent than another dependent on ϵ. In chapter 6, we focus on returning approximately the top K heavy hitters, under a similar accuracy guarantee.

Notation: For accuracy, an event occurs "with high probability" of at least $1 - \delta$ for small parameter δ. The frequency f_i of an item i is the number of occurrences in the data stream. The frequency moments $F_k = \sum_i f_i^k$ are a useful value for streaming algorithms based on frequency [BCI15, BCI16]. Of note, F_0 is the number of unique items, F_1 is the number of items, and F_2 is commonly used to measure the "spikiness" of a distribution. For F_2 on a subset, we use superscript to notate the subset (such as the criteria), where $F_2^{criteria} = \sum_{j:criteria(j)} f_j^2$ denotes F_2 on the subset of values j satisfying the criteria. For example, $F_2^{\neq H}$ is $\sum_{j:j\neq H} f_j^2 = F_2 - f_H^2$ is the F_2 of all elements except H, and F_2^S for a set S means $\sum_{j\in S} f_j^2$. An item is α-heavy if $f_H^2 \geq \alpha F_2^{\neq H}$. Item k refers to the k^{th} most frequent value.

3.1.1 Prior results and math

CountSieve [BCI15] and BPTree [BCI16] are heavy hitter streaming algorithms, from which we use some baseline results. The proofs and theorems were modified slightly to fit with our use case.

The starting building block we use is an accumulator where each value x contributes $Z_x = s(x) = \pm 1$, for a vector Z and/or sign function s. We use the chaining inequality unchanged from Theorem 1 from BPTree [BCI16] and Theorem 15 in CountSieve [BCI15].

Theorem 3.1.1. *(Bucket bounds) Let $Z = \pm 1$ be drawn from a 4-wise independent family. Let $X = \sum_i f_i Z_i = \langle f, Z \rangle$. Then $E[|X|] < C\sqrt{F_2}$ for a constant $C = 23$.*

Corollary 3.1.1.1. *Applying Markov's inequality to Theorem 3.1.1, $Pr[|X| \leq (C/\delta)\sqrt{F_2}] \geq 1 - \delta$.*

Then there is the Pair subroutine, where values are split across two buckets X_0 and X_1, and each value x contributes $s(x) = \pm 1$ to bucket $X_{h(x)}$, for a classifying hash h. We form similar claims as in Lemma 6 of CountSieve [BCI15]. There are a few differences of the following result compared to the original. Namely, the original claim was missing the $\sqrt{2}$ factor, the original hid the δ term by using a non-constant parameter $C > 23/\delta$ instead, and the original had an extra factor due to pseudo randomness we don't use.

Theorem 3.1.2 (Pair)**.** *Let $s(x) = \pm 1$ be 4-wise independent, and $h(x) = \{0, 1\}$ be a classifier. Let X_0, X_1 be two buckets, with $X_j = \sum_{i:h(i)=j} s(i) f_i$ for $j \in \{0, 1\}$. Let $b = \arg\max_b |X_b|$ be the larger of the two buckets.*

Suppose there is an element H that is $2C^2/\delta^2$-heavy, for $C = 23$. Then $Pr[b = h(H)] \geq 1 - 2\delta$, so with high probability b is the bucket containing H.

Proof. Without loss of generality, suppose $h(H) = 1$, for the heavy element H. Let $Y = X_1 - f_H s(H)$ be everything in X_1 except H, and let $C = 23$. Then by Corollary 3.1.1.1 and union bound, both of the following hold with probability at least $1 - 2\delta$:

- $|X_0| \leq C/\delta\sqrt{\sum_{j:h(j)=0} f_j} = \frac{C}{\delta} F_2^{X_0}$
- $|Y| \leq C/\delta\sqrt{\sum_{j \neq H:h(j)=1} f_j} = \frac{C}{\delta} F_2^{X_1, \neq H}$

Therefore with probability at least $1 - 2\delta$, it follows:

$$\begin{aligned}
|X_1| - |X_0| &= |Y + f_H s(H)| - |X_0| \\
&\geq |f_H s(H)| - |Y| - |X_0| \\
&\geq f_H - C/\delta \left(\sqrt{F_2^{X_0}} + \sqrt{F_2^{X_1, \neq H}} \right) \\
&\geq f_H - \sqrt{2} C/\delta \sqrt{F_2^{X_0} + F_2 X_1, \neq H} \\
&= f_H - C\sqrt{2}/\delta \sqrt{F_2^{\neq H}} > 0
\end{aligned}$$

This uses the fact that $a+b \leq \sqrt{2(a+b)^2}$ and that by assumption on H, $f_H > C\sqrt{2}/\delta\sqrt{F_2^{\neq H}}$. Hence if $h(x) = 1$, then $|X_1| > |X_0|$ with probability at least $1 - 2\delta$, and symmetrically for $h(x) = 0$. □

3.1.2 Count-min sketch

The Count-min sketch [CM05] is a well-known sketching structure that provides a way to query for frequency estimates. Notably, while it is efficient to obtain an estimated frequency for any one element, it does not trivially reveal the most frequent elements (naively querying for every element is exponential work). This provides an efficient way to estimate the frequency of a queried element, which we use to select the most frequent items out of a list of candidate values. CountSketch [CCF04] is another common structure for frequency estimation, which is discussed in section A.1

For any value x, CountMin returns an estimate $\hat{f}_x$ that never underestimates f, and with probability at least $1 - \delta$ is less than ϵm of an overestimate. The parameters are $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$. The structure is a table S_i^j with d rows length w. The algorithm uses d pairwise independent public hashes $h_i(x)$ range w. For an input x, each row i has entry $h_i(x)$ incremented by 1, i.e. $S_i^{h_i(x)} \pm 1$ for each i. The frequency estimate is then $\hat{f}_x = \min_i S_i^{h_i(x)}$.

3.2 Cryptography background

In the context of this work, we focus on secure multi-party computation, specifically where a pair of servers are jointly computing on private inputs. Therefore all cryptographic tools are assumed to be for two parties unless otherwise specified. At a high level, the goal of the servers is to jointly compute $f(\{x\})$ without learning anything else about any x.

Secret sharing Secret sharing (specifically, threshold secret sharing) is a method that allows a private value x to be split into n shares such that any subset of t shares can reconstruct the value, but any smaller subset reveals nothing about x (i.e., is indistinguishable from random). For this work, we have private values split into two shares, such that both shares together recover the value, but any share on its own is indistinguishable from random so reveals nothing. We use $[x]$ to denote a secret share of the value x. Specifically, Server 0 has the share $[x]_0$ and server 1 has $[x]_1$, and neither knows x itself. When the subscript is excluded, it is not referring to a specific server's share.

A two party secret scheme consists of the following:

- An input domain $\mathbb{F}$ and secret domain $\mathbb{F}'$
- A secret value $x \in \mathbb{F}$.
- A share creation function $Share(x) = ([x]_0, [x]_1)$, where $[x]_b \in \mathbb{F}'$ and $[x]_b$ looks like a uniformly random element of $\mathbb{F}'$, for $b \in \{0, 1\}$.
- A share recovery function $Rec([x]_0, [x]_1) = x$.

It should also have the following properties:

- **Privacy** For $b \in \{0, 1\}$, $[x]_b$ looks uniformly random sampled from $\mathbb{F}'$. (Without knowing $[x]_{1-b}$, the value $[x]_b$ is indistinguishable from a random value). Hence a party holding only one of the two shares learns nothing about the original x, as it is equally likely to come from any valid input.
- **Correctness** The recovery function Rec provides correctness, ensuring the shares still represent x uniquely. It is usually not used directly (as that would reveal x), although

may be implicitly used during a protocol.

- **Expansion** The secret sharing representation of shares in $\mathbb{F}'$ should not be much bigger than the original representation of values in $\mathbb{F}$. Specifically, $\log|\mathbb{F}'| \le c \log|\mathbb{F}|$ for some small constant expansion factor c.

This can also represent more complex values, for example $[x+y]$ denotes shares of the secret value "$x+y$". Generally, operations similar to constant multiplication, and addition of shares can be done locally, while share multiplication and other more complex operations require the servers to communicate. Note that adding a non-shared (constant) value to a secret share only occurs on one server, while multiplication is on both. Namely, for $[y] = [x]+1$, this is typically done for example as $[y]_0 = [x]_0 + 1$ and $[y]_1 = [x]_1$, while $[2x] = 2[x]$ on both servers.

Arithmetic vs Boolean shares The two main types of shares we use are arithmetic shares and Boolean shares. Arithmetic shares $[x]^A$ sum to the secret value x under some λ-bit modulus N, for security parameter λ. For this work, N is prime for polynomial identity testing [Sch80, Zip79] and SNIPs [CB17]. Because the modulus is constant and public, the modulo is implied when working with arithmetic shares and not mentioned unless necessary. Addition and scalar multiplication are done locally, with $[x+y]^A = [x]^A + [y]^A$, and $[cx]^A = c[x]^A$. Scalar addition $c + [x]^A$ is done by one server adding the value via $[c+x]^A_b = b * c + [x]^A$ for $b \in \{0,1\}$.

Boolean shares $[x]^B$ bitwise XOR to the secret value. Bitwise XOR, denoted $\oplus$, is Boolean XOR applied to each bit. We use $\oplus$ to denote bitwise XOR, which is XOR (bitwise addition modulo 2) applied to each bit of the input independently. For k-bit Boolean shares, $\mathbb{F} = \mathbb{F}' = Z_{2^k}$, $Share(x) = (r, x \oplus r)$ for a random $r \in \mathbb{Z}_{2^k}$, and $Rec([x]^B_0, [x]^B_1) = [x]^B_0 \oplus [x]^B_1$. Share XOR can be done locally with $[x \oplus y]^B = [x]^B \oplus [y]^B$.

Correlated randomness. Correlated randomness is a tuple of random values $(r) = (r_0, \ldots, r_k)$ secret shared across the servers that allow for more efficient computation on secret shares. The values of (r) are correlated in some way, with the true values unknown to both servers, and are used once to assist in computation. Notably, they don't rely the input, so can be computed "offline" by the servers before any inputs are received, essentially for free, to allow the "online" phase with inputs to be done more efficiently. However the straightforward approach to generate randomness for all inputs beforehand would require linear space to store all computed values, so section 5 discusses obtaining correlated randomness in a streaming-efficient manner.

Multiplication. Multiplication of shares requires communication between servers, but this can be made more efficient using precomputed multiplication triples [Bea92]. Note that Boolean bitwise AND is multiplication mod 2, so this works for both multiplication of arithmetic and Boolean shares. A multiplication triple is a set of secret shared random values $([a], [b], [ab])$, where a and b are both random and unknown to both parties. This allows for very efficient one-round multiplication of two shares $[x]$ and $[y]$ to get $[xy]$, using a multiplication triple and communicating a single share.

SNIPs. Since secret sharing inherently hides clients' inputs, there should be some mechanism for servers to ensure that the shares are well-formed and represent a valid input while maintaining that privacy. Furthermore, protocols can be designed such that client messages consist of multiple shares to more cleverly represent inputs and allow for more efficient server computation, so servers will want to also validate that the shares are well-formed. Prio [CB17] introduces SNIPs, *secret-shared non-interactive proofs*, that allows clients to prove properties of their shares. In a SNIP, there is a public arithmetic validation circuit representing the desired property, i.e. the circuit on the values represented by the shares returns the validity. The client evaluates the circuit on their shares, and send some additional shares based on the evaluation to the servers. The servers then efficiently check that the client's evaluation of the circuit was valid and hence the input is valid, and the client can lie with only a negligible probability. This can also be considered as auxiliary information attached to the user's input shares. This takes two rounds of communication with two arithmetic shares total communicated, and the circuit takes in arithmetic shares as input. The SNIP client share has $2M + 5$ arithmetic shares, where M is the number of non-constant multiplication gates.

Share Conversion. It can be useful to convert shares between Boolean and arithmetic. Of interest is Boolean to Arithmetic conversion (B2A). This is because a b-bit Boolean share bounds the secret value to b-bits to help with validation, while arithmetic shares typically accumulate better. With two servers, B2A can be done efficiently in one round [AGJ21] using correlated randomness known as a daBit [RW19], which is $([r]^A, [r]^B)$ for a random bit r, detailed below. For a b-bit value x, B2A converts $[x]^B$ to $[x]^A$ by converting all bits in parallel in one round using b daBits, and sending b Boolean messages. This provides a more efficient bit length validation [AGJ21] compared to using a SNIP [CB17], since it does not require clients to build a proof of length. There are also methods to convert shares in other directions [DSZ15], which are not used in our algorithms.

General evaluation. There are general multiparty computation methods for evaluating any function on secret shared data, such as BGW [BGW88] and garbled circuits [Yao86]. These methods generally achieve constant rounds, but require more computation and large messages. This is helpful for complex computations that require many MPC rounds in series, such as indexing into an array or sorting. Our implementation uses EMP Toolkit [WMK16] for garbled circuits.

Malicious security. Malicious security in multi-party computation is when one more more parties are not following the protocol, and can be assumed to be colluding as a common adversary. In all cases, the honest players should be able to maintain privacy, keeping their secrets hidden from the adversary. In some cases, the honest parties may be also able to maintain soundness and correctness, detecting and working around the adversary's misbehavior. For example, two servers can discard badly formed client data, and operate on the honest inputs. If one server is misbehaving, the other server focuses only keep its shares of honest client inputs private. Soundness and correctness are done by showing how any badly formed messages can be caught. For privacy, informally this can be argued by arguing all pieces are malicious secure, since the adversary has to learn the information from somewhere. Similarly, if values such as correlated randomness are generated in a malicious secure manner, then malicious secure algorithms using these are also maliciously secure. For example, daBits can be generated using maliciously secure OT [AGJ21, WMK16], or by validating client randomness later in section 5. Then because share multiplication is maliciously secure [Bea92], the B2A protocol using daBits and multiplication [AGJ21] is also malicious secure. Similarly, SNIPs are malicious secure [CB17].

Oblivious Transfer Oblivious Transfer (OT) is a well-studied protocol. We take the 1-out-of-2 OT protocol as a given we can use. In the OT protocol, there is a sender with two messages (m_0, m_1) and a receiver with a choice bit b. After the protocol, the receiver obtains m_b but learns nothing about m_{1-b}, and the sender learns nothing about b. This can be written as the receiver getting $(1-b)m_0 + bm_1$. One specific example is Correlated Oblivious Transfer (C-OT) of the form $(r, r+m)$ for sender's message m and random mask r. With C-OT, the sender has a message and learns r from the protocol but not b, and the receiver gets $r + bm$. This allows for additive shares of bm for bit b and arithmetic value m, with $[bm]^A_S = -r$ and $[bm]^A_R = r + bm$. This can also be done in parallel in the other direction, to allow for multiplication of bit share $[b]$ with arithmetic share $[m]$. This is used for example to generate daBits as in Prio+ [AGJ21]

Oblivious transfers can be done with precomputation, where some correlated randomness can be computed beforehand for efficient online OT with inputs, such as in [ALS13, DSZ15, IKN03]. Furthermore, OT extension such as in [ALS13, IKN03], where a small amount of precomputed randomness can be efficiently extended, requiring sub-linear precompute storage. Therefore privacy preserving streaming algorithms can use OT efficiently without concern for precompute space. In [ALS13] the online phase is done in two rounds, with the sender sending a total of mL bits, and receiver sending a total of $m\lambda$ bits, for m messages length L and security parameter λ. For our use case, C-OT is almost always done using field values (arithmetic shares, and/or random field elements) as messages in OT, so the size of each message is usually the security parameter. So, typically this is one round with $m\lambda$ bits each way to do C-OT with m arithmetic shares.

3.2.1 Secret Share Protocols

There are two underlying types of secret shares used in this work: arithmetic and Boolean. General secret shares such as for client inputs can be an array of these type of these secret shares, together derived from the underlying original secret value.

Share conversion. As noted, B2A share conversion is a useful way to convert from Boolean shares $[x]^B$ to arithmetic shares $[x]^A$ of the same value, using precomputed daBits.

The daBit generation is as follows. Both servers first sample a random bit $[b]^B$. The first server then samples a random mask x and sets $[b]_0^A = [b]_0^B - 2x$, and has the Oblivious Transfer message $(x, x + [b]_0^B)$, which the second server queries with b_1 to get $y = x + [b]_0^B[b]_1^B$. The second server then sets $[b]_1^A = [b]_1^B - 2y$. It follows that $[b]_0^A + [b]_1^A = [b]_0^B + [b]_1^B - 2[b]_0^B[b]_1^B = [b]_0^B \oplus [b]_1^B$, as desired. In addition, neither server knows b, even with one being malicious.

The single bit B2A on a bit $[x]^B$ is as follows. Both servers compute $[v]^B = [b]^B \oplus [x]^B$ and reveal it to each other get v. Each server $i \in \{0, 1\}$ then has $[x]_i^A = vi + (1 - 2v)[b]_i^A$. This takes a single round sending one bit (revealing v), and is secure since v is masked by the unknown b. For a multi-bit $[x]^B$, the servers convert each bit $[x_i]^B$ in parallel, and then locally compute $[x]^A = \sum_i 2^i [x_i]^A$.

Share multiplication. The Beaver multiplication protocol [Bea92] works for both arithmetic and Boolean shares, as noted. Boolean Beaver triples are generated using Oblivious Transfer. Our protocol currently does not use arithmetic multiplication, although they can be generated using Oblivious Transfer or semi-homomorphic encryption. The protocol to multiply $[xy]$ is as follows. Each server holds inputs $[x], [y]$, and a precomputed triple $([a], [b], [ab])$ for random a, b, and $c = ab$. Both locally compute $[d] = [x - a]$ and $[e] = [y - b]$, then send them to each other, revealing d and e. Finally, they compute $[xy] = [c] + [x]e - [y]d - ed$.

Share Comparison. Sometimes, a desired operation is to compare two secret values. Namely, given shares $[x]^A$ and $[y]^A$, we want shares $[x < y]^A$, where $(x < y)$ is 1 if y is greater than x, and 0 otherwise. For this to make sense M is chosen to be very large, for example at least twice as big as the largest possible secret value, to avoid any overflow, so that "very large" shared values ($> M/2$) are negative. Specifically, if $x \mod M < M/2$ then $0 \leq x < M/2$, and if $x \mod M > M/2$ then $-M/2 < x < 0$. This gives a way to define what greater than and negative mean when represented with additive secret shares. Prior works such as [DFK06] gives protocols for computing $[x < y]^A$ given $[x]^A$ and $[y]^A$. For this protocol they require M to be odd, which aligns with M being a large prime to make SNIPs from Prio [CB17] work properly. Comparison also allows for indicators/equality, with $[x = y] = (1 - [x < y])(1 - [y < x])$ taking two comparisons in parallel and a multiplication. This also allows for $[|x|] = (1 - 2[x < 0]) * [x]$.

We make several changes to this method to better server our purposes. First, the method in [DFK06] relies on correlated randomness of $([r]^A, \{[r_i]^A\})$, where r_i is the i^{th} bit of a random r. As an improvement, these values can be precomputed, which we do. This helps a lot, as the generation method requires bitwise addition, which is one round per bit by default but can be made constant, but it is also nondeterministic and is run until it makes a valid share. Second, we also make a second version that can produce Boolean shares $[x < y]^B$ from $[x]^A$ and $[y]^A$. This follows essentially the same logic, just using Boolean values throughout where relevant, such as with the correlated randomness.

CHAPTER 4

Privacy preserving streaming model

Recall typical streaming algorithm has m inputs x_i provided by clients, and a server wants to be able to compute some $f(\{x_i\})$ on the stream of inputs efficiently. In Privacy Preserving Streaming, the goal is to similarly evaluate some $f(\{x_i\})$ on a stream of inputs, except the inputs themselves should remain hidden from the party doing the evaluation. For this, the role of evaluator is taken by two independent servers, so that neither has access to any individual input. The two servers replicate the role of the streaming algorithm server, using multi-party computation to ensure privacy, while remaining efficient.

4.1 Algorithm stages

There are three main stages in a privacy preserving streaming algorithm. These are similar to the stages in a normal streaming algorithm, with extra considerations for privacy and the two server computation model. First, a client with an input creates shares based on their input, sending one share to each server. Then, the servers efficiently and securely accumulate the shares, also validating that the shares and input are valid. Finally, once all inputs are in, the servers do evaluation using their accumulators to obtain the final answer.

Making Client Shares: First, a client with an input x creates a pair of values (x_0, x_1) representing the input, with each server receiving their corresponding half of the pair. Typically, this is done with secret shares, such as shares of the input value. The pair can also include other data, to assist the servers in efficiently validating and computing with the data. For example, if the client also sends shares of $f(x)$, then the servers don't have to compute

$f(x)$ themselves, but will have to ensure that they are shares of $f(x)$. As a simple example of this, Variance in Prio [CB17] sends shares of $([x], [x^2])$ so that the servers check that the second share is the square of the first rather than do the squaring themselves. Each client locally produces their shares, though they can also use public data (such as server public keys or parameters) for the protocol. Because the algorithm is 1-pass, a client sends a single message to the corresponding servers.

There are two ways for the client to send their shares. The straightforward way is for the client to send each x_b to Server b, and e.g. add in a unique tag to both halves to ensure that the servers can sync up the shares, in case multiple client pairs are received in different orders. The alternative is to use public key encryption so that the client sends both shares to one server. The client sends server 0 both x_0 and an encrypted $Enc_1(x_1)$, server 0 forwards the encrypted $Enc_1(x_1)$ to server 1 who decrypts it. This removes the need for the sync tag and allows the client to only talk to the closer of the two servers, but will result in larger share sizes and increased client complexity for the added encryption. Further improvements can be made to reduce the total share size, as discussed in subsection B.2.1.

Validation and Accumulation: For each pair the servers receive, the servers jointly perform computation on the shared input, to mimic the accumulation update process of the standard streaming algorithm. They first do validation to make sure that the shares represent a valid input. Validation comes for free in standard streaming algorithms, where obviously invalid answers are trivially ignored, but the addition of privacy and shares makes this non-trivial, as described later below. If the input passes validation, then update their local state, likely with additional work performed. This is done via multi-party computation, where the servers take some number of rounds doing local computation, sending a message to the other and receiving one in return, and doing new computation using the message and their state. Similar to how the state of the server in a standard streaming algorithm is the accumulators, the local state in a privacy preserving streaming algorithm typically includes shares of said accumulators. The state also accounts for values both servers know, such as the number of inputs processed, and the number of valid inputs.

Evaluation: Finally, once the stream has been received, the servers jointly compute using their local states to evaluate the final answer. Typically this is simply evaluating the function on the final accumulators as usual, except with secret shared accumulators. For some simple protocols, the simplest way is to reveal the accumulators to each other then locally compute the answer equivalently to the normal streaming algorithm, as long as revealing the accumulators doesn't leak anything about individual inputs. More advanced algorithms will require computing on the shared state, using multi-party computation, as the accumulators would reveal additional undesired information.

Offline The servers can also take advantage of an "offline" stage, where the servers aren't performing any protocols so can do computations before any inputs are received. This can be used for cryptographic preprocessing, such as generating correlated randomness that can be used during the above "online" phases to reduce the complexity of the server work.

4.2 Threat model

Generally, there are two relevant threat models for a privacy-preserving streaming algorithm. The first is when an adversary corrupts any number of clients maliciously such that the clients may deviate from the protocol arbitrarily, and additionally one of the two servers is passively corrupted (meaning it obeys the protocol but colludes with the corrupted clients to try to obtain private information from honest clients). The adversary is also adaptive in the sense that it may dynamically choose which additional clients to corrupt based on the information it has gained from previous corruptions. In this case, a privacy-preserving streaming algorithm should protect the privacy of honest players' inputs and it should also protect the robustness of the resulting statistic. That is, honest players' inputs should remain unknown to the adversary, and the adversary should not be able to corrupt the output of the protocol (beyond misreporting inputs of corrupted clients). This protects against, for example, a malicious actor attempting to manipulate the output of the function by monitoring one server's traffic while also submitting maliciously chosen inputs from one or more devices.

In the second case, the adversary has all the power of the first case but has maliciously corrupted one of the two servers as opposed to the passive corruption of the first scenario, now allowing that server to deviate from the protocol at will. In this case, a privacy-preserving streaming algorithm should still protect the privacy of honest clients' inputs, but we do not require the robustness of the output to be maintained. This is chosen for two reasons: first, protecting the output is a lower priority (especially in the case where a presumably strictly monitored compute server has somehow been severely corrupted), and second, doing so would require heavy cryptographic machinery which significantly impacts performance in the much more common cases of either passive server corruption or no server corruption.

The servers communicate through a pairwise secure authenticated encrypted channel. Similarly, the clients send their message through a secure authenticated channel. Depending on the share distribution method chosen, the clients can have the connection to only one of the servers, or both. Multiple client messages can be in arbitrary order between clients, with no synchronicity requirements, to one or both servers.

4.3 Robustness

The system is robust if it returns the output of the function on the set of valid inputs, meaning all valid inputs are included, and all invalid inputs are rejected. Note that we only require this to hold in the first adversarial case involving an honest-but-curious server colluding with malicious clients. If only honest inputs are provided, the privacy preserving streaming algorithm will always give the same output as the streaming algorithm. If too many inputs are invalid, the protocol may instead choose to produce no output in order to maintain privacy of the valid inputs. For example, if all but one input is badly formed then the outputting f on the single honest input may reveal information about that honest input. Thus the algorithm may choose to abort if e.g. more than some threshold fraction of the inputs are invalid. Note that because streaming algorithms can include randomness, it should have the same output $f(S)$ on the same streaming randomness, which implies having the same output distribution.

Definition 4.3.1 (Robust)**.** *Let S be the subset of valid inputs, and T some predetermined threshold. Let f be a streaming function on a variable number of inputs, with random streaming input R. A protocol is perfectly robust if it rejects whenever $|S| < T$, and otherwise outputs $f(S, R)$.*

For a security parameter λ, let S' be a set such that the protocol either rejects if $|S'| < T$ or outputs $f(S', R)$ otherwise. The protocol is robust with respect to λ if $S' \supset S$ contains all valid inputs, and any invalid input has probability at most $2^{-\lambda}$ of being in S'.

Robustness is typically accomplished via input validation, where for any invalid input (whether outside of the range or badly formed), the protocol determines with overwhelming probability the provided input is not valid and should not be included in S. Therefore, as long as the number of invalid inputs does not exceed the privacy-preserving threshold, the output will be $f(S)$ where S contains only the valid user inputs except with negligible probability. Otherwise, if there are too many invalid inputs, the servers abort with high probability. Note that this approach reveals the number of valid inputs, which we discuss further below.

4.4 Privacy

Ideally, the servers should learn nothing about any individual x_i besides what can be inferred from the returned answer $f(x)$. This should hold even in the second adversary case with one malicious server collaborating with malicious clients. Essentially, the output and view of honest inputs should be equally likely to come from any set of honest inputs.

Definition 4.4.1. *Let $\{x\}$ and $\{y\}$ be two data streams. Suppose the two streams lead to* *the same output $f(x) = f(y)$. Furthermore, suppose that for invalid inputs, their locations are the same across the two streams, and which check each fails is also the same. Then the protocol is private if a server performing the protocol can't tell the difference between $\{x\}$ and $\{y\}$ better than random with a negligible advantage, i.e. $1/2 + 2^{-\lambda}$. In addition, if the server knows some of the client inputs, then it can't distinguish between two streams that also share those inputs. Malicious privacy holds if these properties hold even if one of the servers is allowed to arbitrarily deviate from the protocol.*

At a high level, the view of a single server should be equally likely to come from any stream leading the same output. The view of a server is all information it gains during the execution of the protocol. This consists of the client shares it receives, all messages from the other server, its local history, and the final output of the protocol.

For convenience, the output not only includes the output of f on the valid inputs, but also which inputs are invalid. While further privacy can be obtained by also hiding the validity of inputs, this is generally not worth the trade-off, requiring multiple rounds for low additional benefit. Multiple prior works [CB17,AGJ21,BBC21] all choose to reveal the input validity during accumulation. The additional information leaked by revealing whether each individual input passed a validity check has minimal impact, and does not impact privacy of the underlying value. Honest clients' inputs are expected to be valid in the first place, so revealing an input as valid is not any additional information. If an input is revealed to be invalid, this again does not impact privacy since we care only about protecting the privacy of honest players' inputs. Furthermore, the most that is learned is which check the input failed, but not why it failed the check, notably still keeping any submitted values hidden. For example, servers learning that an input is out of bounds does not reveal the value or how far out of bounds it was, and is not new information beyond the fact that it is invalid.

Consider any two data input streams $\{x\}$ and $\{y\}$ which result in the same output, i.e. $f(x) = f(y)$. We can consider all invalid inputs to be equivalently removed (same clients and failed checks across both streams), and therefore can assume both streams to consist of only valid inputs. Then for each server, the protocol run on $\{x\}$ versus on $\{y\}$ should provide indistinguishable views, meaning they can't tell which stream led to that answer. Since all streams that lead to $f(x)$ seem equally likely to a server, this implies that each server learns no additional information about the valid inputs besides that f returns $f(x)$. Notably, this also applies at all intermediate steps, which means that the view of the server before and after processing an input should not reveal any additional information about the input. For example, if only certain accumulators updated depending on the value, two streams can be distinguished if they have values leading to different accumulators being updated. Therefore, every accumulator needs to update for each value, generally done through "empty" updates (such as shares of 0).

This is very similar to the notion of privacy used in Oblivious Algorithms, where the algorithm is independent of some property, such as doing the same operations regardless of input. For example, Oblivious RAM [GO96] provides privacy by ensuring memory access is identical for all inputs. Similarly, a privacy-preserving streaming algorithm should have no way to distinguish between valid inputs from the computations it performs, outside of the final output of the algorithm. For example, conditional logic (update only some accumulators, or stop early if the value fits some criteria) is on its own not oblivious because the information of whether or not the algorithm followed a particular branch of logic reveals additional information about the input. Hence any conditional logic needs to either be made oblivious in some way (generally by following the wrong branch with an empty value), or by intentionally revealing the information (such as if the input is valid or not).

This also covers the cases both where the server knows additional client inputs, and where the server is malicously collaborating with a subset of clients. Namely, a single server may collaborate with a subset of dishonest clients and choose bad inputs to learn information about honest inputs, and can also choose to send bad messages to the other server. At a high level, privacy should follow since the server has no access to the honest server's shares from the honest clients. Specifically, consider everything that the adversary knows, which is the view of the malicious server, the output $f(x)$ (or the fact that it aborts), and the malicious inputs. Then malicious privacy holds if there exists a simulator that can fake the same messages the honest server sent without knowing the honest inputs. This means that any possible honest inputs could have led to what the adversary sees, so the adversary learns nothing about the honest inputs.

Optionally, some additional leakage may be reasonable for efficiency, if it reveals minimal information about honest inputs. As noted, this generally includes the validity of inputs, and which checks failed. Furthermore, algorithms can choose to leak aggregate information during evaluation. Generally, a straightforward evaluation will reveal the value of each accumulator, after which the standard non-private evaluation math happens to return the answer. For example, Prio [CB17] reveals the expected value while computing variance, as it is one of the two accumulators maintained. Various strategies can be used to also hide this for efficiency trade-offs, further discussed in section 4.6.

4.5 Efficiency

The efficiency requirement is that everything should be efficient. Across the whole execution, the total space requirement used by the servers should be polylog in the number of inputs and input size, which matches standard streaming algorithm behavior. Per client input during accumulation, the server computation time, communication message size, and number of server communication rounds should all be polylog in the input size. The final evaluation phase should have the computation, communication size, and number of rounds all polylog in the input size and number of inputs. For each client, the computation cost, space, and total share size should be polylog in the size of the original input value.

Note that because most values are stored as secret shares, their size scales with the security parameter λ. Therefore generally λ is chosen to be large enough to not overflow compared to the number of inputs, and so can be considered logarithmic in the number of inputs. So a single accumulator value can be considered to be size $\lambda = O(\log n)$, as desired.

Beyond the requirement that the different measures of efficiency are all polylog, there is a scale of which should be prioritized to be more efficient. Client devices are generally much weaker than servers, so the computation and client share size are higher priority than optimizing the server behavior. Similar to standard streaming, accumulation occurs live while processing inputs and for every input, while evaluation happens after and only once, so improving accumulation generally has higher impact than evaluation. Generally the bottleneck in multi-party computation is the lag between the servers, so minimizing server communication rounds is more impact compared to computation, though total bytes communicated is also impactful to minimize. Ideally, the number of rounds should be constant with polylog total message size.

Privacy-preserving streaming algorithms can be thought of as building off of streaming algorithms, using a similar structure as a normal streaming with additional overhead for privacy. Therefore, it can be helpful to compare efficiency of an algorithm with and without the privacy. Specifically for accumulation and evaluation, the amount of additional computation along with the cost of communication rounds ideally is not too much greater than that of the standard algorithm. Because standard streaming does not have communication rounds, exact ratios apply only to computation. Generally, simple streaming algorithms should require a very small number of rounds, ideally constant, and as complexity grows the "leeway" for complex privacy preserving computation and messages grows. Normal streaming algorithms always have clients just sending their input, so comparison isn't as helpful.

For example, some streaming algorithms [BCI15] take advantage of short circuiting with conditional logic on inputs to eliminate many inputs early to reduce work. This logic can't occur in oblivious algorithms, so privacy-preserving streaming algorithms lose this advantage, leading to a larger relative efficiency loss. Correspondingly, its easier to focus on streaming algorithms that don't rely on this type of logic, as the advantage will be lost.

Furthermore, the servers can handle a polylog sized batch of clients input at a time, validating and accumulating them in parallel. Ideally, this parallel process takes the same number of rounds, which can be done if the update is independent of the accumulator value. For example [CB17, AGJ21], if the servers compute an update share independent of the accumulator value and only use the accumulator value to add the update, then the server can compute all the shares for the batch in parallel and update once at the end.

4.6 Conversion From Streaming to Privacy

An approach to creating a privacy streaming algorithm is to start with a streaming algorithm and add privacy. We first discuss generic conversion methods, then discuss more specific algorithms and techniques and tools that can be used.

4.6.1 Generic Conversion

It is possible to generically turn any streaming algorithm into a privacy preserving streaming algorithm. Accumulation and evaluation can be thought of as functions, where accumulation takes in secret shared input and state and produces a new secret shared state, and evaluation takes the secret shared state and produces a new output. As noted, there are multiple works on oblivious multi-party algorithms on secret shared values, such as BGW [BGW88] and Garbled circuits [Yao86]. These methods convert arbitrary functions into oblivious secret shared versions of these functions, as desired. This also means that the input shares are just shares of x, giving the client minimal work. Validation can also be treated as an additional function, either within accumulation or before it, which can be similarly generically made secure. For example, general conversion can have the client send shares of x, the servers update with $[state_i] = accum_{BGW}([state_{i-1}], [x_i])$, and then at the end produce $f(x) = eval_{BGW}([state_{final}])$, using accumulation and evaluation made secure via say BGW. However, these generic conversion methods have large overhead and general costs in order to support all possible functions of any size, leaving room for optimization.

4.6.2 Improved Conversion

There are various ways to improve on these generic methods, taking advantage the structure of the streaming algorithm, the data, and the privacy preserving streaming model. Therefore, the goal of designing efficient Privacy Preserving Streaming algorithms is to choose input shares that are small and efficient for the client to calculate, yet allow the servers to efficiently accumulate them in a small number of rounds.

There are several multi-party computation strategies to reduce rounds. For example, the servers can both check validity and compute the new state at the same time, and only save the new state if validity passes. They can also use correlated randomness to do some operations in less rounds.

One key thing is that clients know their own input, and so can provide different or extra shares that reduce the server work required. Generally, if the accumulation is $g(h(x))$, then the client can provide an additional y and a proof that $y = h(x)$, and the servers then can in parallel compute $g(y)$ and also validate that $y = h(x)$. Prio [CB17] provide an efficient method of general proofs, where clients send some extra data as a proof of validity. Then the servers can efficiently check the provided proof rather than needing to run through the whole proof themselves.

Prio+ [AGJ21] takes advantage of efficient share conversion to prove that inputs are of the right length in a single round. Share conversion allows for efficiently taking advantage of different properties of the shares, which can help reduce rounds. For example, Boolean shares require multi-round addition circuits to add numbers but give length validation and XOR for free, while arithmetic shares require no rounds to add but require a multiplication round for XOR and a proof of length.

Another tool is to use frequency vectors as a selector mask, such as for accumulators. For example, suppose there are K accumulators, and based on some $h(x)$, only one accumulator is updated by some $h'(x)$. Rather than the servers using multiple rounds of multiplication and comparison to select, the clients can provide a length K frequency 0/1 vector that has a 1 at $i = h(x)$, and 0 elsewhere. Then the servers need just one multiplication to multiply $v_i * h'(x)$ to update accumulator i in parallel across i, as it will zero out $i \neq h(x)$ as desired. Servers also have simple validation that the provided shares represent a frequency vector, using the technique of validating frequency vectors from Prio+ [AGJ21]. Depending on the protocol, the servers may need to also check that the frequency vector did come from a valid or consistent x, which can also be done in parallel using e.g. SNIPs.

4.7 Prior works

We look at several prior works, and how well they fall under the model of privacy preserving streaming algorithms.

Prio [CB17] supports a variety of simple functions, most of which fall under this model. However, two of their protocols, for frequency and max, use shares of size exponential in the input length. They focus on exactly computing frequency, which can't be done in efficient space, and they encode the inputs and accumulator as the exponential length frequency vector. Their protocol for maximum encodes inputs and the accumulator in unary, which is similarly exponential. Maximum can be done with arithmetic shares and a single accumulator, but would require comparisons to accumulate and does not batch easily.

Prio+ [AGJ21] relies on precomputed randomness (daBits) to significantly reduce the size of the proofs needed in Prio, so with the space efficient correlated randomness from chapter 5, Prio+ then falls under this model. Prio+ also supports similar functions as Prio, so also has the same flawed frequency and max algorithms.

Poplar [BBC21] finds the most frequent item, but the algorithm's linear space requirement is not streaming friendly. Their algorithm stores each player's shares during accumulation, and evaluation repeatedly iterates over all of the stored input shares. This work in chapter 6 improves this by creating a privacy preserving streaming-friendly top K algorithm.

There are also a few works which solve more specific problems in specific contexts which qualify as privacy-preserving streaming algorithms. One such work trains a k-nearest-neighbors model over private client data using a count-min sketch [MDC15]. Although it almost meets our criteria, it does not provide robustness in the presence of malicious clients. If a space-efficient zero-knowledge proof was used to prove validity of client inputs, however, it could be made to match our model. It is also worth noting that instead of MPC, this work uses homomorphic encryption to achieve privacy, and this is perfectly acceptable under our model.

Some works compute heavy hitters, but in a different context, such as [HKR12], in which all participants perform a distributed computation over multiple rounds, rather than streaming to dedicated servers.

Another pair of works privately and efficiently computes the number of unique inputs among users [WJS19, HLL21]. These protocols do indeed match our model. In fact they achieve a stricter level of privacy than we require, security-with-abort against malicious servers. In comparison, while we simply require privacy with malicious servers, these works also halt to prevent faulty output. Most applications do not need to defend against malicious servers, as servers can usually be stored in highly controlled environments, but since these works specifically aim to secure the Tor network (where no assumptions can be made about particular nodes), such a strict requirement makes sense.

CHAPTER 5

Generating correlated randomness

While the use of precomputed correlated randomness has advantages in privacy preserving streaming, the straightforward approach has downsides. Specifically, precomputing all necessary correlated randomness offline before the first input requires storage linear in the number of inputs. Some methods such as Oblivious Transfer are able to use a small amount of precomputation for a large amount of online computation using extension to use sublinear precompute space, which is further discussed in section 3.2. In this section, we explore how to generate other forms of correlated randomness, specifically daBits, in a streaming friendly manner. The general idea is that if clients are in batches sized K, the server caches $O(K)$ correlated randomness which it refreshes as it handles inputs, rather than generating and storing all at once. A simple but unreliable method is that if client inputs are spaced out enough, then there is enough downtime to refresh this cache "offline" between processing inputs, but this is very dependent on the expected client behavior. Instead, we look at two methods. The first is to generate the randomness for the next batch in parallel with processing the current batch. The second is to have the clients provide the randomness, and for the servers to validate them. For the second, we create a new algorithm for batch validation of daBits.

5.1 Parallel next batch generation

One solution is to generate the correlated randomness for the next batch of inputs at the same time the current batch. For example, suppose it takes 2 rounds to generate some randomness, and 2 rounds to do the accumulation and validation. Then on a batch of inputs, the servers use the existing cache to accumulate, and in parallel generate a new cache for the next batch. This still takes 2 rounds total, as opposed to generating then using the same values in 4 rounds. This works best when the generation can be done in parallel efficiently. In some cases however, secure generation may take many rounds, such as the random shares used to compare shares in [DFK06], further explored in subsection 3.2.1. Furthermore, in our implementation, the Oblivious Transfer library used [WMK16] to generate daBits and Boolean multiplication triples are not easy to do other communication rounds in parallel with.

5.2 Client provided randomness

Another solution is for the clients to provide the correlated randomness for the servers to use. Since clients can be malicious, servers need to validate the randomness, which they can do efficiently in bulk. This requires servers to have several caches, for verified values, unverified values (received but not checked), and server generated values. The server generated values use a constant number to check each batch, and serve as backup values if the validation fails. This allows for the store of precomputed randomness the server needs to only be polylog size, as it's only drawn on for large batches, and the infrequent failed batch replacement.

Correlated randomness can be validated and used in parallel, which save rounds if the inputs are all honest, and has minimal downside if the batch is invalid. For an invalid batch, the conversion needs to be rerun with the server cached randomness, which is the same total rounds as doing validation then conversion in series, with low extra computation of the first few rounds of aggregation on invalid randomness. Privacy is maintained by using client provided randomness only on their own input. If the servers are honest, the result is the same, as all calculations are still done with honest randomness, so clients can't tell the difference. For the servers (even if one is malicious), maliciously provided randomness is only used on inputs from the malicious clients, so honest inputs are still properly masked by honestly provided randomness that neither server can fully know. Therefore a malicious server can't learn the randomness used to convert honest inputs, maintaining malicious privacy. Multiplication triples can be efficiently validated by the server in two rounds [BFO11]. We create a new algorithm using similar strategies to validate daBits.

5.2.1 Polynomial daBit validation

Here we provide a new algorithm where the servers can validate a batch of N daBits via polynomial identity testing [Sch80, Zip79]. At a high level, the algorithm verifies a set of relations $a_i * b_i = c_i$ by forming polynomials f, g, h through the points a, b, c respectively, and then checking if $fg = h$, and this is done through secret shares.

Theorem 5.2.1 (New: Batch daBit Validation)**.** *There is an efficient multi-party batch validation protocol for N unverified daBits over prime modulus M. It requires N unverified multiplication triples, and one verified multiplication triple. It takes 1 round of $N+1$ arithmetic multiplications, and one round of sending an arithmetic share. It requires $O(N)$ space. It always accepts if all are valid, and incorrectly accepts if there are invalid inputs with probability at most $\frac{2N-1}{M}$.*

The protocol to verify N daBits is as follows:

1. Servers pre-agree on $2N$ distinct evaluation points x_i and a random evaluation point σ beforehand.
2. The servers have one precomputed multiplication triple.
3. The input is N unverified daBits $([b^i]^B, [b^i]^A)$, along with N unverified arithmetic multiplication triples.
4. Server 0 builds $[p^i]_0 = 2[b^i]^B_0$ and $[q^i]_0 = 0$, and Server 1 builds $[p^i]_1 = 0$, and $[q^i]_1 = q^i = [b^i]^B_1$. Both servers make $[r^i] = [b^i]^B - [b^i]^A$.
5. Each server builds the degree $N-1$ polynomials $[f]$ and $[g]$ such that $[f](x_i) = [p^i]$ and $[g](x_i) = [q^i]$ for $i \in [[N]]$.
6. Each server extends the points, computing $[p^i] = [f](x_i)$ and $[q^i] = [g](x_i)$, for $i \in \{N, \ldots, 2N-1\}$.
7. The servers extend r, computing $[r^i]^A := [p^iq^i]$ as shares of $r^i = p^i * q^i$ for $i \in$

$\{N, \ldots, 2N-1\}$, using the N unverified multiplication triples.

8. Each server then builds the degree $2N-1$ polynomial $[h]$ such that $[h](x_i) = [r^i]$ for $i \in [[2N-1]]$.
9. The servers also compute $[fg](\sigma) = [f(\sigma) * g(\sigma)]$ using the precomputed triple.
10. Both compute $[fg](\sigma) - [h](\sigma)$, and then send them to each other, to get $f(\sigma) * g(\sigma) - h(\sigma)$ in the clear. They accept if this is equal to 0, and reject otherwise.

Proof. The goal of the validation is checking that each $b^i = [b^i]_0^B \oplus [b^i]_1^B = [b^i]_0^A + [b^i]_1^A$ for all of the input daBits $([b^i]^B, [b^i]^A)$. This is equivalent to checking that $p^i q^i = r^i$ for each i. With the construction of the polynomials, this is equivalent to $f(x_i)g(x_i) = h(x_i)$ for all $i \in [[N]]$. Since f, g, and h are the unique polynomials that go through their corresponding points, this is equivalent to checking that $f(x) * g(x) = h(x)$ for all x. This is checked by evaluating the degree $2N-1$ polynomial $f(x)*g(x)-h(x) = 0$ on a random evaluation point σ, and accepting if and only if the difference is zero. If all daBits are valid, then the $fg = h$, and the difference polynomial is the zero polynomial and is therefore accepted. Because the modulus M is prime, the probability of incorrect acceptance of a nonzero polynomial is equal to the probability of the polynomial identity test failing, which is at most $\frac{2N-1}{M}$ for a degree $2N-1$ polynomial [Sch80, Zip79].

If the servers are honest, then the clients don't know σ. Hence even if all the clients in a batch are collaborating to have full control over f, g, h, they have at most a $\frac{2N-1}{M}$ probability of creating a nonzero polynomial $fg - h$ that evaluates to 0 on the unknown σ. Even if the clients manipulate their triples to make the extension of h incorrect, this is still before h is evaluated on σ, so just builds some other polynomial h'. Furthermore, the final multiplication of $f(\sigma)g(\sigma)$ uses a server-generated triple, so clients can't influence the calculation of the final evaluation point. Therefore this is robust against malicious clients.

If one of the servers is malicious, then they can't learn anything about honest inputs, even by collaborating with some of the clients. Because of the honest clients, the honest triples ensures that adversary does not have full control of the polynomial extension. The final multiplication done with a malicious-secure triple so won't reveal anything about fg, and therefore the final stage of evaluation just reveals $fg - h$ to both servers.

The first round is the multiplications in parallel, consisting of N unverified multiplications for extending $r = p * q$, and the single verified multiplication of $f(\sigma) * g(\sigma)$, sending $2(N+1)$ arithmetic shares. The second round is for revealing the final value, sending another share. This totals to 2 rounds, with $2N + 3$ total arithmetic shares sent. $\square$

Appendix 5.3 improves on this with a different multiplication method to send only $N+2$ shares and also reduce correlated storage. Also, Several optimizations done in Prio [CB17] for SNIP evaluation can be used here. The point σ can be fixed, and only updated occasionally, rather than needing to be reset (and resent) before every batch. This is because an adversary may try to break the reused value by repeatedly trying different inputs, but after q attempts, they can pass one validation with odds at most $q(2N-1)/M$. With large modulus M, σ can be reused e.g. $q = 2^{10}$ times before the servers spend an offline round refreshing the point. Furthermore, with a fixed evaluation point, the interpolation and evaluation can be improved by Lagrangian interpolation and fixed points x_i.

5.3 Alternative multiplication triples

In standard secret shared multiplication with multiplication triples [Bea92], each player has $[x], [y]$ and wants $[xy]$, which is done with triples of the form $([a], [b], [ab])$. In some cases, such as validating daBits, instead player 0 has x, player 1 has y, and they still wish to obtain shares $[xy]$. The straightforward approach is for each player to act as their share of the other value is 0, and the share of their value is just the value. Then using standard multiplication, this is one round and still secure. However, the size of the messages during the multiplication, and the size of the correlated randomness can both be reduced. To improve on this, we define a new multiplication method for values in this context, using a modified correlated random alternative multiplication triple to make the computation more efficient.

Definition 5.3.1 (New: Alternative Multiplication Triple)**.** *An Alternative Multiplication Triple is correlated randomness where Player 0 has $(a, [ab]^A_0)$, and Player 1 has $(b, [ab]^A_1)$,* *where a and b are uniform random, a is not known to Player 1, and b is not known to player 0.*

This requires two arithmetic values per correlated value, compared to normal multiplication triples requiring 3. Furthermore, this also improves the multiplication, requiring one value communicated during the online phase rather than two [Bea92].

Theorem 5.3.1 (New: Alternative Multiplication)**.** *There is a multiplication protocol where Player 0 has x, Player 1 has y, and they create secret shares $[xy]^A$ without learning the other's shares. This protocol uses one alternative multiplicative share, and requires one round of communication in which each sends one arithmetic value ($O(\log n)$ for values modulo n).*

The alternate multiplication protocol is as follows:

1. Player 0 computes $x - a$ and sends it to Player 1
2. Player 1 computes $y - b$ and sends it to Player 0.
3. Player 0 computes $[xy]_0 = [ab]_0 + (y - b)a$.
4. Player 1 computes $[xy]_1 = [ab]_1 + (x - a)y$.

Proof. For correctness, we have as desired:

$$\begin{aligned}[xy]_0 + [xy]_1 &= ([ab]_0 + (y - b)a) + ([ab]_1 + (x - a)y) \\ &= [ab]_0 + [ab]_1 + (y - b)a + (x - a)y \\ &= ab + ay - ab + xy - ay \\ &= xy\end{aligned}$$

For security, Player 0 learns $y - b$ but since b is random and unknown to Player 0 it servers as a mask to hide y, and since it's the only received message privacy is conserved. Similarly, Player 1 gets $x - a$ masked by random a. □

Corollary 5.3.1.1. *Theorem 5.2.1 can be improved using alternative multiplication triples, with all multiplications using alternative multiplication instead. This reduces the total communicated arithmetic shares from $2N+3$ to $N+2$, the size of the client provided randomness from $4\lambda + 1$ to $3\lambda + 1$, and the one precomputed triple from 3 to 2 arithmetic shares.*

Proof. This follows from replacing all multiplications with alternative multiplications. Recall only server 0 holds $[b]_0^B$, while both servers build $[p]$ out of it, and similarly for $[q]$ from $[b]_1^B$. Therefore, p and f can be held by just server 0, and q, g by just server 1. Then, the multiplications of both $r = pq$ and fg are of the form for alternative multiplications, as desired. The client values include the daBit (1 arithmetic share and 1 bool), and the unverified multiplication triple (reduced from 3 arithemtic shares to 2). □

CHAPTER 6

Finding heavy hitters

We create a privacy preserving streaming algorithm for finding the most frequent values. This is generally known as the heavy hitters problem, where a heavy hitter is a significantly frequent item by some metric. One common notion is past some threshold, and some algorithms [BCI16] aim to return all items with a frequency past for example ϵF_2. Another common notion is in terms of ranking compared to all other values, such as the K most frequent, which is what we focus on. Because it is impossible to always return exact answers in sublinear space (distinguishing all possible permutations of uniform except a few slightly more frequent), the algorithm is an approximation algorithm.

6.1 Outline

We first build an input oblivious streaming algorithm, following the framework. The algorithm has logic for choosing which accumulator to update, which we resolve by the client also sending masks for selection, so that the servers can obliviously update the correct accumulator. With this, we convert the streaming algorithm to a privacy preserving streaming algorithms, using the masks and validation for efficient accumulation.

The base building block is a pair of buckets, where each item is sorted by value into one of the two buckets. So if there is a single heavy H, the bucket with H will be larger, giving one bit of information. SingleHeavy has $L = \log n$ pairs of buckets with different hashes, to identifying a single heavy H. For a single layer, the stream is split into B disjoint substreams, with SingleHeavy run on each, so heavy items are likely to be in their own substream and found, and this is repeated Q times. Layer produces QB candidates, which includes all items with values $f > T$ for some threshold T, such as $\epsilon\sqrt{F_2}$. To account for heavy items not being past T, this process is repeated for R layers. Each layer progressively deletes half of the items at random, which correspondingly lowers the target threshold T. Then the heaviest items are likely to be found in their corresponding halving layer and be candidates, if they survive the halving. Finally, the candidates are sorted by frequency estimate, and the top K are returned.

At a high level, the guarantee is that for some parameter $M > K$, all returned values are within the top M. This is needed since some items may be lost to the halving. This can also be made more exact, focusing on being more likely to return the more frequent of the M, depending on the values. For example, if an item is sufficiently heavy (say $> \sqrt{F_2}/K$), then it is very likely to be a candidate. In addition, the final step of sorting by frequency estimate ensures that the approximately most frequent of the candidates are found, which is better. A future avenue of exploration is refining the parameters and guarantees if additional information is known about the distribution.

Theorem 6.1.1 (Privacy Preserving Top K)**.** *Let there be a set of m inputs with range n, held by a set of clients. A pair of servers want to find the K most frequent items in the stream. Then there exists a polylog privacy preserving streaming algorithm that provides an approximation of the top K elements of the honest inputs to the stream.*

The algorithm takes security parameter λ, accuracy parameters δ, ϵ, and target parameter K, and returns a set of K values. With probability at least $1 - \delta$, the following guarantees hold on the values returned by the algorithm: All returned values have $f > f_M/\sqrt{2}$, for the item rank $M = K + 2\log(1/\delta)$. If there are at least K values with $f \geq \epsilon\sqrt{F_2}$, then all returned values are from this set. If there are less than K values with $f \geq \epsilon\sqrt{F_2}$, then all of these values are returned.

Each client send shares sized $O(\lambda \log n + \log(1/\delta)(\log K + 1/\delta))$. The servers validate and accumulate each input in 3 rounds communicating $O(\log K \log(1/\delta)(K^2 \log^2 n + \lambda))$ bits. The accumulators are of size $O(K^2 \log K \log(1/\delta) \log^2 n)$. Evaluation time is time polylog in the input size.

If the servers are honest, then any bad input is caught and discarded with probability at least $1 - 2^{-\lambda}$*. Each server cannot learn anything about any honest client's inputs beyond what is implied the output of the function and the number of honest inputs, even if the server is malicious and collaborating with dishonest clients.*

6.2 Streaming algorithm

First is the streaming algorithm variant, similar to prior algorithms [CCF04, BCI15, BCI16]. The algorithm is designed with privacy preserving in mind, so accumulation has no conditional logic and is additive. We start with a frequency estimator, which can obtain the best values out of a list of candidates. We then build up to the full algorithm to find candidate heavy values, with increasing repetition to work on more general distributions. Combined, the estimator then returns the top items out of the candidates, and optionally their approximate frequencies.

6.2.1 SingleHeavy

We start with a pair of buckets $X_{0,1}$, where each item x contributes $s(x) = \pm 1$ to bucket selected by $h(x) = \{0, 1\}$.

Theorem 6.2.1 (Pair)**.** *Let* $s(x) = \pm 1$ *be 4-wise independent, and* $h(x) = \{0, 1\}$ *be a classifier. Let* X_0*,* X_1 *be two buckets, with* $X_j = \sum_{i:h(i)=j} s(i) f_i$ *for* $j \in \{0, 1\}$*. Let* $b = \arg\max_b |X_b|$ *be the larger of the two buckets.*

Suppose there is an element H *that is* $2C^2/\delta^2$*-heavy, for* $C = 23$*. Then* $Pr[b = h(H)] \geq 1 - 2\delta$*, so with high probability* b *is the bucket containing* H*.*

This is based on prior works [BCI16, BCI15], and proved in subsection 3.1.1. Pair gives one bit of information, that $h(H) = b$. Repeated runs with different hash choices allow for identifying H.

Theorem 6.2.2 (SingleHeavy)**.** *Let* L *be the bit length of the inputs, and* $i \in [[L]]$*. Let*

$h_i = \{0, 1\}$ be L linearly independent hashes, and $s_i(x) = \pm 1$, and X^i_j be the buckets for $j \in \{0, 1\}$. Let $b_i = \arg\max_b |X^i_b|$ be the Pair result for each i. If H is $8C^2L^2/\delta^2$-heavy, then it can be found by the b_i with probability at least $1 - \delta$.

Correctness follows by union bound on *Theorem* 6.2.1, and relabeling δ. Recovery follows as long as the choice of h gives that $x \to \{h_i(x)\}_{i=0}^{L}$ is a bijection. We choose each $h_i(x)$ to be a linear combination of the L bits of x, and the set of h_i forms a linearly independent system. This also allows for easy recovery via matrix inversion.

6.2.2 MultiHeavy

Next, to detect multiple heavy hitters in a stream, the stream is split into B disjoint substreams, so that each substream is likely to contain a single heavy hitter that can be detected by Single Heavy. To amplify the probabilities, this is done Q times with independent randomness. Note the F_2 of a substream is reduced, as there are less elements contributing.

Lemma 6.2.3 (F2 reduction). *Consider a substream with $1/B$ of the values, selected randomly. Specifically, each value has $1/B$ probability of being in the substream. Then for F_2^B being the F_2 of this substream, it follows that $E[F_2^B] = F_2/B$ and $Pr[F_2^B < F_2(a/B)] \geq 1 - 1/a$.*

This follows by linearity of expectation and Markov's inequality, since $F_2 = \sum f_i^2$ and items are included or not by value, i.e. each f_i is fully included or not included at all.

Repeating this Q times gives QB candidates from the SingleHeavy, with at least B distinct values. The goal is for these candidates to include all heavy items of interest with high probability. However, the set can also include other values, such as values that happened to be heavy in their substream and be found but aren't globally heavy, or even random "junk" values returned by SingleHeavy when the assumption of a single heavy value fails within that substream.

Theorem 6.2.4 (Single Layer). *Split the stream into $B = 16(10LC)^2/\epsilon^2$ substreams by value, with SingleHeavy run on each, and repeat this $Q = O(\log(1/\epsilon)\log(1/\delta))$ times. Let C*

be the set of between B and QB unique candidates found, and $H_1 = \{x : f_x > \epsilon\sqrt{F_2}\}$. With probability of at least $1 - \delta$, all of H_1 is found, i.e. $H_1 \subseteq C$.

Proof. For a value H, consider a fixed q, and let $B_q = h_q(H)$ be the substream with H. Then $E[F_2^{B_q,\neq H}] \leq E[F_2^{B_q}] = F_2/B$. Let E_i be the event that $F_2^{B_q,\neq H} \leq 8F_2/B$, and by Markov $Pr[E_i] \geq 7/8$. Then for $H \in H_1$, event E_i implies that $f_H \geq \epsilon^2 F_2 \geq \epsilon^2(B/8)F_2^{B_q,\neq H}$, which means H is $\epsilon^2(B/8)$-heavy in its substream. Plugging in B, SingleHeavy finds H with probability at least $9/10$ if E_i, which combined gives odds of at least $7/8 * 9/10 > 3/4$ of H being a candidate.

For $Q = O(\log 1/\epsilon)$ repetitions, by union bound across the repetitions and Chernoff bounds across all items in H_1, there is at least a $2/3$ probability that all values in H_1 are candidates. With further repetitions of $Q = O(\log(1/\epsilon)\log(1/\delta))$, this gives a probability of at least $1 - \delta$ of success. □

Finally, we want to select the best candidates out of those found. The most straightforward way is to select them based on their frequency estimate, using e.g. CountMin aggregated at the same time. For top K, we will simply sort by frequency estimate, and pick the top K.

Matching prior algorithms Note that keeping all elements past a threshold based on F_2 gives similar guarantees as many other algorithms [CCF04, BCI15, BCI16]. For this, an F_2 estimator is done in parallel, which can be done efficiently [AMS99]. Then, at the end all items with $\hat{f} < 3\epsilon/4\sqrt{\hat{F}_2}$ are discarded. This gives a list that with high probability contains all items with $f > \epsilon\sqrt{F_2}$ and no items with $f < \epsilon/2\sqrt{F_2}$. This then can be made privacy preserving using a similar strategy to section 6.3, as the AMS estimator adds ± 1 to select buckets similar to Pair, and circuit can do the filtering. However, the number of returned values is random and dependent on distribution, so it can't guarantee at least K are found. The idea is to be able to repeat this for lower and lower ϵ until enough are found, but this has to be done in one pass efficiently, which leads to our algorithm.

6.2.3 Full top K algorithm

Finally, the algorithm needs to be able to find the heaviest items even if they aren't that heavy. To do this, rather than reducing ϵ in repeated runs which grows the complexity, instead F_2 is reduced by sampling. We do this by repeating this process in R layers. Each layer has half as many values as the previous layer, chosen randomly. For example, hashing via $h'_r \to \{0, 1\}$ and only keeping those that are 0 for all of the first r hashes. Each layer then has a lower threshold than the previous, reduced by a factor of $1/\sqrt{2}$ via Theorem 6.2.3, letting the layer find lower frequency heavy items out of those remaining. Another interpretation of this is that SingleHeavy can fail if there are multiple heavy values in the substream, so by removing half of the items randomly there is a chance that the substream will end up with exactly one heavy item. Then all the candidates are gathered across the layers, and the top K are picked according to frequency estimate. Since items can be lost in this halving process, if the heaviest items aren't heavy, then not all of them can be found.

Therefore, we want to claim that for some $M > K$, the K values we return are approximately among the top M. Specifically, up to the layer containing M. A value is in a layer if it's found by that layer with high probability, but not the layer above. Recall that each layer finds items with frequency at least $T = \epsilon\sqrt{F_2}$ for some ϵ, and that by Theorem 6.2.3 this threshold reduces by a factor of $\sqrt{2}$ each time. Therefore layer 0 is $> T$, and each layer after contains frequencies $[T2^{-k/2}, T2^{-(k+1)/2}]$. Then the sorting by estimate will find the best K out of these, and so the goal is to ensure there are enough good candidates.

Lemma 6.2.5. *Let r be the layer containing item M. Suppose that $M/2^r > K + 2\log(1/\delta)$. Then with probability at least $1 - \delta$, at least K among the values in layer r or better survive.*

Proof. We bound the case where there are less than K that survive halving. The worst case is if the top M elements all lie in layer r, but there can be more items and/or better probability than layer r. Each item has an independent survival chance $p \geq 1/2^r$. Let X be the number of survivors, so $E[X] \geq pM$. By choice of M and Chernoff, $Pr[X > K] \leq exp(-(1 - K/\mu)^2\mu/2) < \delta$, as desired. □

Once a low number of values are left, they likely to be spread out enough that all are found, leading to the whole layer being found regardless of threshold and frequency.

Lemma 6.2.6 (Lowest Layer)**.** *Once a layer has less than $\sqrt{B}$ values, all values are found, and no more halving is needed.*

Proof. If a value is alone in one of the B substreams, it's trivially always found. So if across the Q repetitions, if each value is alone somewhere, all values are candidates, as desired. By the Birthday problem, if there are at least $\sqrt{B}$ items distributed B buckets, there is a constant probability they are all alone. With $Q = O(\log(1/\epsilon)\log(1/\delta)) > O(\log 1/\delta)$, this give at least $1-\delta$ they are all alone. □

These cases matching means once the last layer with all items is reached, there are enough good candidates found. In other words, either M's layer is early and enough survive, or late enough and all are found.

Lemma 6.2.7. *Let r be the last relevant layer, with at most K items. By the above two results, if $B > K^2$, then for $M/2^r \geq K + 2\log(1/\delta)$, at least K items are found among the top M with probability at least $1-\delta$.*

The parameter ϵ from Theorem 6.2.4 is flexible, so can be tuned relative to K as desired. For example, ϵ can be made large so that $B = (40CL/\epsilon)^2 = cK^2$ for some $c \geq 1$.

Some of the random hashes can be reused to reduce share and validation size. Because this algorithm can be thought of as doing the halving process in each of the QB substreams, the same randomness can be used across R. Similarly, across B, the substreams are disjoint, so they can re-use the same SingleHeavy randomness. However, SingleHeavy by design need different Pair randomness across L pairs, and each layer needs different substream randomness across Q.

6.2.4 Parameters and guarantees

There are two forms of the guarantee on the set of K values returned. All returned values at least close to the M^{th} largest element's frequency or better. There is also a set of most frequent items which has maximal overlap with the returned values, either fully returned or having all returned values.

Theorem 6.2.8 (Streaming Top K accuracy)**.** *There exists an algorithm that given a data stream with of L-bit values provided by clients to a pair of servers, returns a set of K values. The total accumulator space used by the servers $O(K^2 \log(K) \log(\delta^{-1}) \log(n)^2 \lambda)$.*

With probability at least $1 - \delta$, the following properties all hold: All returned values have $f > f_M/\sqrt{2}$, where f_M is the frequency of the item rank $M = K + 2\log(1/\delta)$. Let $S_\epsilon = \{x : f_x \geq \epsilon\sqrt{F_2}\}$. If $|S_\epsilon| \geq K$, then all returned values are from S_ϵ. If $|S_\epsilon| < K$, then all of S_ϵ are returned.

Proof. The parameters chosen are $Q = O(\log(K)\log(1/\delta))$, $R = \log n/2$, $B = O(K^2)$, $L = \log n$, and Count-min $w = e/\delta$ and $d = \ln(1/\delta)$. Because of the substreams, there is guaranteed to be at least $B > K^2$ unique candidates. By frequency estimation the top K candidates are returned with high probability.

For the first guarantee, recall that layer 0 is $> T$, and each layer after is frequencies $[T2^{-k/2}, T2^{-(k+1)/2}]$. By Theorem 6.2.5, all candidates are from the layer with M or better, which worst case are frequency at least $f_M/\sqrt{2}$.

For the guarantees with the most frequent set S_ϵ, this comes from layer 0, when no values have been eliminated. By construction, all of S_ϵ are candidates. If there are more than K, then all returned are among them. If there less than K, then the whole set is returned because they are the most frequent. □

6.3 Secret shares

Lemma 6.3.1. *The client shares are* $\lambda(4R+2)+QB+R+dw+L$ *bits. The accumulation is 3 rounds communicating* $O(QBRL+\lambda(Q+d))$ *bits, consuming* $3QBRL+QBR+QL$ *Boolean multiplication triples and* $4QBRL+QB+dw+L+R$ *daBits. These allow for secure accumulation and evaluation that is robust against malicious clients, and privacy preserving against a malicious server.*

6.3.1 Full protocol

The indices are r, b, q, l, d, w are over R, B, Q, L, D, W respectively. There are public hashes $h_{q,l} \to \{0,1\}$, $s_{q,l} \to \{0,1\}$, $g_q \to [[B]]$, $\sigma_r \to \{0,1\}$, and $\gamma_d \to [[W]]$. Both h and s are linear combinations of the bits of x mod 2. So $h_q^l(x) = \sum_i^L a^{q,l,i} x_i \pmod 2$ for constant bits a, which can be considered as multiplying the $L \times L$ matrix A_q with the vector of the bits of x. Furthermore, for each q, the L equations representing h_q^l are linearly independent, meaning that A_q^{-1} exists and is publicly known. The servers have accumulators $[X_0^{r,b,q,l}]^A$, $[X_1^{r,b,q,l}]^A$, and $[F_d^w]^A$. The rest of the hashes are pairwise independent.

Consider a client with input x. They build the Q length B frequency vector m_q, with 1 at index $g_q(x)$ and 0 elsewhere, and similarly D length W frequency vectors c_d from $\gamma_d(x)$. They also build the length R vector z, where $z_i = \prod_i^r \sigma_r(x)$, i.e. the first i values of σ_r are all 1. These vectors are all split into element-wise Boolean shares. The input x is represented as $[x_i]^B$ for each bit x_i of x. Finally, the client creates a SNIP proof [CB17] that each value of $[z]^B$ came from σ_r on R, producing shares $[SNIP]$. Finally, the client sends the shares $[x_i]^B$, $[m_q]^B$, $[c_d]^B$, $[z]^B$, and $[SNIP]$ to each server.

Accumulation: During accumulation, multiplication is done with Boolean Beaver triples, and B2A with daBits, expanded on in subsection 3.2.1.

1. The servers receive $[x_i]^B, [m_q]^B, [z]^B, [c_d]^B$, and $[SNIP]$.
2. The servers locally compute $[u_q^l]^B = h_q^l(\{[x_i]^B\}_i)$ and $[v_q^l]^B = s_q^l(\{[x_i]^B\}_i)$.

3. For the first round, the servers do the following:
   (a) B2A to get $[m_q]^A$, $[z]^A$, $[x_i]^A$, and $[c_d]^A$ from the corresponding inputs.
   (b) Multiply $[u_q^l]^B$ and $[v_q^l]^B$ to get $[(uv)_q^l]^B$
   (c) Outer multiply $[m_q]^B \times [z]^B$ to get $[mz_{q,b,r}]^B$
4. Using $[x_i]^A$, $[z]^A$ and $[SNIP]$, the servers run through the beginning of the SNIP protocol of [CB17].
5. For each frequency vector in $[m_q]^A$ and $[c_d]^A$, each server locally sums their shares, to get a vector $[sums]^A$ of length $Q + D$.
6. For the second round, the servers do the following:
   (a) Perform the first communication round of SNIPs.
   (b) Reveal $[sums]^A$ to each other in the clear.
   (c) Multiply $[mz]^B$ against $[u]^B, [v]^B$, and $[uv]^B$ to get $[umz]^B$, $[vmz]^B$, and $[uvmz]^B$ respectively. These are repeated outer multiplications, where QL against QRB are Q outer multiplications, obtaining sized $QBRL$ outputs.
7. If one of the *sums* is not 1, discard the input.
8. Continue the SNIP validation process using the intermediate shares from round 2, producing the final validation share $[Scheck]^A$.
9. For the third round, the servers do the following:
   (a) Reveal the value of $[Scheck]^A$ to each other.
   (b) For each r, b, q, l, use B2A to get $[mz]^A$, $[mzu]^B$, $[mzv]^B$, and $[mzuv]^B$.
10. If $Scheck \neq 0$, then discard the input.
11. For each r, b, q, l, update $[X_0]^A \mathrel{+}= [mz]^A - [mzu]^A - 2[mzv]^A + 2[mzuv]^A$, $[X_1]^A \mathrel{+}= [mzu]^A - 2[mzuv]^A$, and $[F_d]^A \mathrel{+}= [c_d]^A$.

Evaluation: The servers use multiple rounds of share comparison [DFK06] to obtain the shares $[b_l^{r,b,q}]^B = [|X_0^{r,b,q,l}| < |X_1^{r,b,q,l}|]^B$. For each r, b, q, the servers locally apply the $L \times L$ inverse matrix A_l^{-1} to $[b_l^{r,b,q}]^B$ across L, to obtain the bits $[y_l^{r,b,q}]^B$. The servers then do one round of B2A to obtain $[y_l^{r,b,q}]^A$, and then produce the candidates $[y_{r,b,q}]^A = \sum_{i=0}^{L-1}[y_i^{r,b,q}]^A$.

The servers then use a garbled circuit to perform the final steps, with $[]^C$ representing the secret shared circuit values.

1. For each candidate $[y_{r,q,b}]^C$, the frequency is estimated as $[\hat{f}_y]^C = min_d F_d^{h_d(y)}$.
2. The list of pairs $([\hat{f}_y]^C, [y]^C)_{r,b,q}$, is sorted by frequency then by value.
3. Adjacent pairs are iterated over, and for pairs of duplicates replaces the first by $(0, 0)$, and finally sorts again by descending frequency.
4. The top K values $[y]^C$ (and optionally, their frequencies) are revealed to obtain the approximately K most frequent items.

Secret share comparison is used to obtain $[b_l^{r,b,q}]^B = [|X_0| < |X_1|]^B$, identifying the larger bucket. The share conversion used [DFK06] is expanded on in subsection 3.2.1, with some optimizations. The inverse of the bucket classifiers is applied locally to $[b_l^{r,b,q}]^B$ across L to obtain shares of the l bits $[y_l^{r,b,q}]^B$ of each candidate. After a round of B2A, the candidates are computed from the bits as $[y_{r,b,q}]^A = \sum_{i=0}^{l} 2^i [y_l^{r,b,q}]^A$. A garbled circuit The candidates then are queried for their frequency estimate in the CountMin. The circuits take in the CountMin accumulators $[F]^A$ and the candidates $[y]^A$, and obtains estimates $[\hat{f}_y]^C = \min_d [F_d^{h_d(y)}]$. The circuit then sorts $([\hat{f}_y]^C, [y]^C)$ by frequency, then value so duplicate values are adjacent. Duplicates are removed by iterating down pairs, and replacing the first $(0, 0)$ if the values are equal, leaving exactly one of each set of duplicates, and so all nonzero values are unique. The array is then sorted again by value, and then the first K values by frequency are revealed, as desired. To reduce circuit size, the servers can first partially evaluate the d CountMin hashes on their shares of the candidates beforehand. For example, if the hashes are the pairwise independent $h(x) = ax + b \pmod{w}$, the servers can locally evaluate each $[ay + b]^A$ on each candidate y, and leave the modulo for the circuit. In addition, either the candidates or the frequencies can be revealed early for efficiency gains but adds leakage, which is a tradeoff that can be explored in the future.

6.3.2 Correctness

On honest inputs, the sub-protocols such as multiplication and B2A are correct. What remains is to show that this matches the pure streaming algorithm on honest inputs. This means that for a value x, we show that the correct buckets are updated with the corresponding shares, and all other buckets are given an "empty" update of a share of 0, through the masks. We consider for a given layer and iteration of the substream. Therefore the buckets should be updated if and only if both the layer mask $z_r = 1$ and also the substream mask $(m_q)_b = 1$, which for brevity we denote z and m below. The value $u = h(x) = h(\{x_i\}_i)$ is the bucket, and for $v = s(x) = s(\{x_i\}_i) \in \{0, 1\}$, the sign $1-2v = \pm 1$ is the value added to that bucket. At the end of accumulation, X_0 is updated by $mz - mzu - 2mzv + 2mzuv = mz(1-u)(1-2v)$, and X_1 by $mzu - 2mzuv = mzu(1-2v)$. These add $s(x) = 1 - 2v$ if and only if the bucket index u is the same, and both masks m and z are one. Therefore exactly the correct X buckets are updated with the right value, and the rest are updated by 0. Similarly, the CountMin is also updated with $[F_d]^A \mathrel{+}= [c_d]^A$, which matches the streaming algorithm.

6.3.3 Complexity

We break down the complexity, with a focus on the size of the shares and communication, along with communication rounds and space required.

The client shares are $[x_i]^B$, $[m_q]^B$, $[c_d]^B$, $[z]^B$, and $[SNIP]$. The Boolean shares total to $QB + DW + L + R$ bits. The SNIP can be done efficiently to check that $z_i = \prod_i^r \sigma_r$, with σ' a random linear combination of the bits of x. Since the hash is public, the constants can be baked into the circuit, so $\sigma'(x)$ can be done without multiplication. Therefore there are $R-1$ multiplications to produce the $\sigma(x)$, and with R values checked another $R-1$ to ensure they all pass, totaling $M = 2R-2$ multiplication gates, requiring $4R+1$ arithmetic shares for the SNIP proof. In total, the client share size is $QB + DQ + L + R + (4R+1)\lambda$.

The total accumulator space is the $[X]$ and $[F]$. This totals to $2QBRL + DW$ arithmetic shares needed for accumulator storage. With parameter choice, this is polylog as desired.

As noted, rounds are done in parallel when possible, done in series only when the prior result is necessary for computation. Therefore, the total number of rounds is based on the longest series, which comes out to 3, for both B2A followed by two round SNIPs and also two rounds of multiplication followed by B2A.

The SNIP proof given arithmetic shares communicates two arithmetic values in total, and requires additional B2A conversion of x_i and z. The frequency validation of the vectors m_q and c_d use the method of Prio+ [AGJ21], requiring B2A conversion of both, and an additional arithmetic share for each vector being checked, totaling $Q + D$. The Boolean to Arithmetic conversion communicates one bit and requires one daBit per bit of the number being converted, and is applied to the above for validation, and the additional various cross multiplications. The Boolean multiplications communicate one bit and consume a Boolean multiplication triple each. In total, accumulation takes $3QBRL + QBR + QL$ Boolean multiplications and $4QBRL + QB + DW + L + R$ B2A conversions, each communicating one Boolean bit, and consuming a single Boolean multiplication triple or daBit respectively, and an additional $Q + D + 2$ arithmetic shares.

As noted, our usage of secret share comparison and garbled circuits for evaluation allows it to require constant communication rounds. In addition, because accumulators take polylog space and evaluation runs off of them, the total space is also polylog.

6.3.4 Robustness and privacy

Robustness A malicious client is unable to manipulate aggregation beyond representing a different valid input. Because the shares are all Boolean, they are all of the right length. The only shares that can be internally badly formed are the frequency vectors m_q and c_d, which are checked following the protocol of Prio+ [AGJ21]. The idea is that since the shares are Boolean, as long as the vector is not too long, the only vectors that sum to 1 are valid frequency vectors and vice versa. What remains is to prove that the shares are all either correlated (are computed directly from x), or can only represent other possible valid inputs.

Note that after the first round both servers locally have shares $[x_i]^A$ of the bits x, which they can also trivially make shares $[x] = \sum 2^i[x_i]$ of x. The SNIP ensures that the halving z mask was computed from x correctly. The remaining unverified correlation is that the substream and CountMin masks m_q and c_d come from x, and since they are checked to be well formed, the client can only change the mask, contributing their value to a different bucket instead. This is equivalent to submitting a value that contributes to those buckets instead. Note also that the client does not control the specific pair contributions to that different SingleHeavy beyond their actual value, so again trying to rig it is equivalent to contributing a different valid input value. The layer selector R is checked because it per layer inclusion, so lying can make a value over-represented (under-represented is similar to just not contributing).

Privacy: Privacy of honest inputs holds with an adversary maliciously controlling dishonest clients and up to one server. Trivially, a single secret share (e.g. the one received from the client) maintains privacy, as it's equally likely to come from any input value over all values of the unknown second share. If a server chooses a random mask to apply to a secret value to send to the other server, then all possible secret value are equally likely to lead to that message (with different masks), also maintaining privacy. This also holds with masks that are indistinguishable from random, such as precomputed correlated random values (multiplication triples and daBits). Therefore, since both Beaver multiplication [Bea92] and B2A share conversion [AGJ21] send a single message with a value masked by a correlated random value, they maintain privacy. Similarly, secret share comparison [DFK06] and garbled circuits [Yao86] both also maintain malicious privacy. As noted, revealing the validity of an input is fine leakage that doesn't harm privacy. The frequency vector validation [AGJ21] is a single extra round (after B2a) that reveals the sum of the vector, which is the same as revealing the validity (as all valid frequency vectors sum to 1). SNIPS [CB17] also maintains privacy, where the first round is values masked by client provided randomness, which maintains privacy for honestly provided client values, and the second round is revealing the validity. The accumulator updates are all arithmetic shares added to secret shared accumulators, which is no new information. Therefore during accumulation, the server view (input and messages) for valid inputs are all equally likely to come from any honest input. Similarly, the server view during accumulation of invalid inputs are with high probability equally likely to come from any invalid input that fails the same validity check. Together, when accumulating an input the view of the server is equally likely to come from any input with the same validity (valid, failed frequency, or failed SNIP), maintaining privacy. Furthermore, secret share comparison [DFK06] and garbled circuits [Yao86] both also maintain malicious privacy. Evaluation occurs on secret shared accumulators, and uses comparison, B2A, and garbled circuits, so similarly the views are equally likely to come from any accumulators leading to the same final output. Putting it all together, the view of a server over the whole protocol is equally likely to come from any data stream with the same output, and the same validity information (which inputs are valid and invalid), as desired.

CHAPTER 7

Benchmarks

We benchmarked implementations of both this work and Poplar [BBC21]. We chose Poplar, as it is the only other private algorithm for heavy hitters that has some server(s) accept a stream of client inputs. Our code is 15,000+ lines of C++, and is available online at `https://github.com/KuraTheDog/Prio-plus/tree/heavy`. All tests for both implementations were done with the same setup on Amazon EC2 c4.2xlarge machines. One server and the client were in Amazon's us-east-1 region (N. Virginia), and the second server was in the us-west-1 region (N. California).

All tests were run on a ZipF distribution with parameter 1.03, as in Poplar [BBC21], which they describe as a natural distribution in network settings though with a more conservative parameter tending towards uniform. The tests were done with 8-bit client input values, and returning the top 5 most frequent values. Both were run on various input amounts, from $10,000$ to $5,000,000$. For this work, parameters similar to subsection 6.2.4 were chosen with $\delta = 0.05$ and $\lambda = 62$. The accuracy parameters gave higher accuracy than the value implies, as across most of the tests it provided the exact correct top K output that Poplar also provides.

For Poplar [BBC21], a threshold of 0.03 was used to obtain top 5 reliably on the ZipF distribution. Poplar parameters were otherwise the default, with internal tree layers having a batch size of 100k with $\lambda = 62$, and the leaves a batch size of 25k with $\lambda = 255$. It is important to note that the version of Poplar used for benchmarking [1] is missing the TLS

[1]The commit of Poplar used for benchmarking is `2ced5718897ba3dd`. The Readme states "Since then [commit used by Poplar paper [BBC21]], since some of the dependencies changed, I stripped out the TLS encryption between the parties so that the code still compiles."

encryption between parties that the original paper [BBC21] used. Therefore, the true timings of Poplar should be slower than what we measured, due to the missing feature. Notably, Poplar reliably crashed on 5 million inputs during evaluation, strongly suggesting a memory issue as opposed to any code issues.

Space: Figure 7.1 compares the accumulator space in log scale required between this work and Poplar. These values are calculated, based on the algorithms and numbers outputted during logging. Poplar reports a key size of 1337 bytes for the chosen parameters. Because poplar stores all keys until the end, the total accumulator space is linear in the input size (key per input). For this work, the main space usage is the accumulators, which are arithmetic shares of the buckets ($2RQBD$) and count-min (WD), which notably do not scale with the number of inputs, merely updating with new received inputs. Note that the size of the values of the accumulators scales with the security parameter λ, which is chosen to be large enough to not encounter overflows with inputs. This has a more significant impact on evaluation, as evaluation for both does computation across all accumulators.

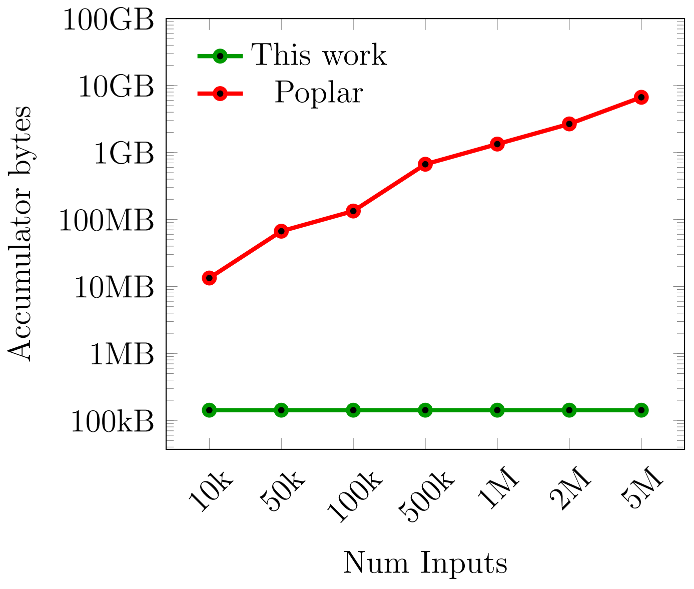

Figure 7.1: Calculated space for storing accumulators, in log scale. Poplar stores every input, so requires linear space. This work requires constant accumulator space.

Timing: Both this work and Poplar use a similar timing of stages. The accumulation stage is from the start of the protocol until the servers finish validating and accumulating all inputs. Note that this also includes the time for the clients to create and send their shares, for consistency. The evaluation stage is afterwards, when the accumulators are used to produce a final answer. End-to-end time is accumulation and evaluation combined, timing the full process from the first input sent to final output returned.

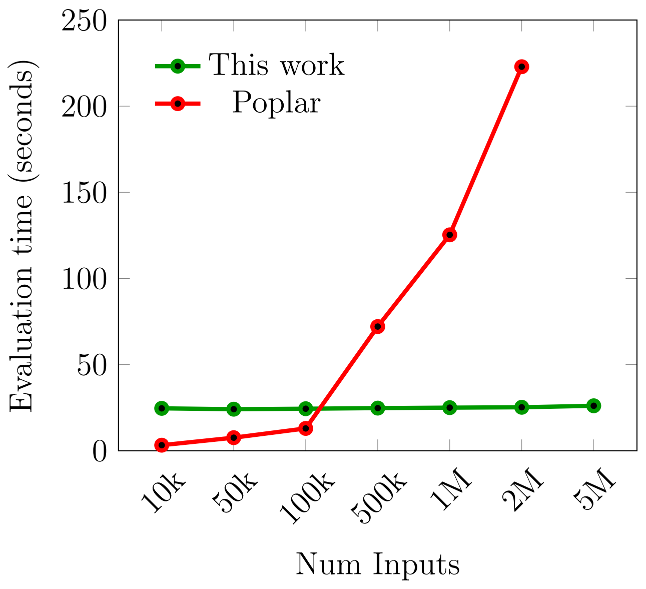

Figure 7.2: Evaluation timing by number of inputs. Notably, this work is constant in the number of inputs, while Poplar takes linear time. Poplar run out of memory and crashed on 5M input.

This work is only slightly slower to accumulate than Poplar. While the validation and accumulation itself for this work are more complex and take multiple rounds, the client shares are smaller and easy to make. In comparison, Poplar does minimal computation during accumulation itself, delaying validation via sketching for the evaluation phase, and instead the timing seems to come down to memory management of receiving and storing the keys, along with the key generation timing. The main advantage of this work shows during evaluation, as seen in Figure 7.2. The timing scales directly with the accumulator space, as both algorithms iterate over the accumulators. Therefore the evaluation for this work is constant time in the number of inputs, due to the accumulator space not being dependent on the number of inputs. On the other hand, Poplar evaluates by iterating multiple times across every key, of which it stores one for every client and so the work grows linearly in the number of inputs. Combined, both this work and Poplar have similar end-to-end times, up to the point where Poplar crashes at high input amounts, as seen in Figure 7.3. This shows that the space advantage has minimal trade-off in total computation timing. Note that this is for a single evaluation call, and evaluation may be called multiple times during the long term execution of a protocol, for example hourly statistical updates based on all inputs gathered so far. In these sort of cases, the end to end time will skew further due to the large difference in evaluation and the continually scaling inputs.

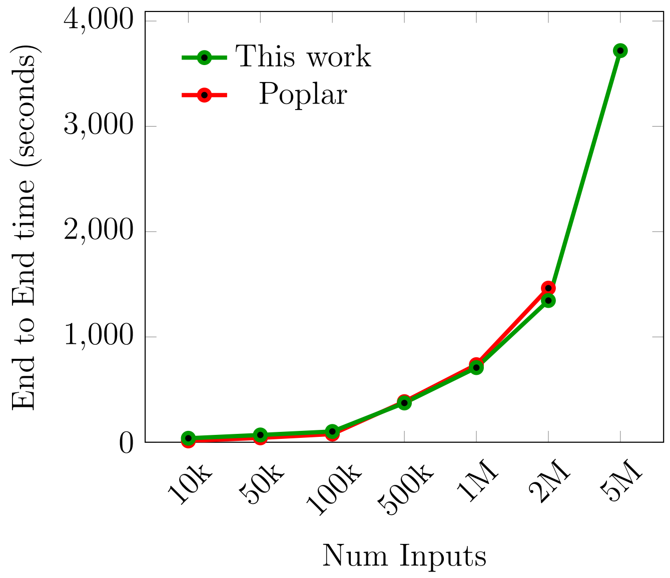

Figure 7.3: End to end timing by number of inputs. Notably, the timings are similar, except Poplar run out of memory and crashed during evaluation on 5M inputs.

CHAPTER 8

Conclusion

We defined a new framework for privacy preserving streaming algorithms that efficiently and privately evaluate functions on large data streams, merging the two areas that has not been well quantified before. We discuss tools and strategies for building algorithms that fit this framework. To support this, we create a new algorithm for batch validation of daBits for efficient correlated randomness. Finally, following our framework we create a new privacy preserving streaming algorithm for identifying the most frequent items, creating a new input oblivious streaming algorithm for efficient addition of privacy. This improves on a prior privacy preserving algorithm for most frequent item that required linear space, which we showcase through running both implementations over cloud.

In the future, we wish to expand this to more functionality, finding or building new input oblivious streaming algorithms that can be made private following this framework. Furthermore, heavy hitters is a useful building block for other streaming algorithms, so another avenue to explore is creating more privacy preserving streaming algorithms using heavy hitters as a building block. If correlated randomness extensions similar to OT extension [ALS13,IKN03] can be found for other random values such as daBits and multiplication triples, it would make the use of such easier to use in a streaming efficient manner. There are also opportunities for improving the heavy hitters algorithm. For example, additional distribution assumptions and other logic can help refine the accuracy guarantee and the algorithm. Some streaming algorithms such as SpaceSaving [MAE05] also use distribution assumptions to determine parameters. There is room to explore the trade-off of revealing aggregate information during evaluation for the purpose of efficiency.

APPENDIX A

Additional background

A.1 Count-Sketch

CountSketch [CCF04] is an alternative to Count-min [CM05] for frequency estimation. We describe it here, and provide a brief summary of a privacy preserving variant.

The CountSketch guarantee [CCF04] is that the frequency estimate $\hat{f}$ satisfies $|\hat{f} - f| \leq \epsilon\sqrt{F_2}$ with probability at least $1 - \delta$. There are two hashes, $h_i \rightarrow [[w]]$ and $s_i \rightarrow \{\pm 1\}$, for $w = \lceil e/\epsilon^2 \rceil$ and $d = \lceil \ln(1/\delta) \rceil$. The servers accumulate $X_i^{h_i(x)}$ += $s_i(x)$, and returns an estimate $\hat{f}_y = \text{median}_i s_i(y) X_i^{h_i(y)}$.

The shares of this follow a similar pattern to Top K and CountMin. The client sends shares of x, and frequency vectors for $h_i(x)$ and $s_i(x)$, which are validated to be frequency vectors [AGJ21] and also via SNIPs. Then there are again two options for accumulation of $h(1 - 2s)$ and $(1 - h)(1 - 2s)$. The servers then do a mix of B2A and multiplication in two rounds to get $h * s$, which lets them accumulate. Similarly to CountMin, a circuit is used for queries.

The main advantage of CountSketch is that it an accuracy guarantee in terms of F_2, versus CountMin being in terms of F_1. However, it has worse parameters, with $d = 1/\epsilon^2$ for CountSketch versus $1/\epsilon$ for CountMin. In addition, CountSketch takes two rounds of accumulation to CountMin's one, although both require the same for SNIP validation. CountSketch also uses two hashes, although the server can evaluate one of the hashes locally.

APPENDIX B

Future work

B.1 More functionality

Following this framework, it would be interesting to explore what more streaming functions can have privacy efficiently added. Similar to Count-Min, there are likely other simple streaming algorithms that are already input oblivious for easier addition of privacy.

B.2 Model extensions

Here we discuss some additional general strategies, optimizations, and options that can be applied to secret shared streaming algorithms, and their tradeoffs and viability.

B.2.1 PRG secret sharing

Secret shares are all of the form $(x - r, r)$ and similarly for Boolean shares. Since server 1's shares are all random values, they can all be generated from a PRG. Hence the first server gets $\{[x]_0 = x - r\}$, while the second server just needs the seed. When the shares are of the form $(x_0, Enc(x_1))$, having x_1 be a compact seed allows for the encrypted message to also be much smaller, which is helpful as public key encryption schemes are longer than the messages.

B.2.2 Parallel B2A

In specific cases, the B2A process itself can be skipped. Namely, suppose the client is sending in some share $[x]^B$ where the servers want $[x]^A$, such as in basic protocols like SUM. With client provided daBits, this is done with $[x]^B, \{[b]^B, [b]^A\}_i$, with a daBit per bit. This can be improved to $([x_i]^B, [x_i]^A)$, where x_i is the i^{th} bit of x. This provides $[x]^A = \sum_i 2^i [x_i]^A$ locally for free, and that $\{[x_i]^B\} = [x]^B$, so the total share size is reduced. This also provides the servers with $[x]^A$ immediately, allowing for accumulation to skip first round B2A. Since these pairs of $([x_i]^B, [x_i]^A)$ are essentially just daBits with client provided "random" x_i, the servers can check that they line up (and therefore x has the right number of bits) using the same batch daBit validation methods. In case of bad inputs, the server can try to recover as usual by using normal B2A on $\{[x_i]^B\} = [x]^B$ to get some input value with the right number of bits.

However, this works best when the client shares are being directly converted. When the servers do manipulation on the value (such as multiplying by masks, as in this work) before converting, this does not provide a benefit. Specifically for this work, B2A on direct client shares is used to validate frequency vectors (ensure they sum to 1) as in Prio+ [AGJ21], so can save some share space and communication, but this is done in parallel with the rest of the operations so does not save any rounds.

B.2.3 Compact final validation check

Note that many of the validations (including SNIPs [CB17]) have the final step checking that some share $[x]^A$ is equal to a known constant value c, typically 0 or 1. The servers then reveal these values to each other to complete the validation. First, the servers pre-agree on some random linear combination f of N values. Then, each of the N checks are changed to checking if $[x'_i]^A = [c_i - x_i]^A$ are shares of 0. The servers then take the linear combination of the shares $f(\{x'_i\})$, and check if this total is a share of 0. If all the shares are of 0, this is trivially 0. If any values are not 0, with "good probability", the result is not 0. These can be compacted into sending one message, which saves on total communication, but takes the same number of rounds. Depending on parameter choice too, the total communication of validation generally overshadows that of validation anyways.

Furthermore, this present some challenges to correctness. To avoid the clients from guessing the randomness through iteration, the servers need to refresh the randomness occasionally (which doesn't need to be often, e.g. every 1 million inputs). Another concern is a malicious server that knows the randomness collaborating with clients to get bad values incorrectly validated. Namely, the bad server can tell a client the random values, so that the client can try to send shares corresponding to a nonzero solution to f. This can be solved by ensuring that all values are affected by a random mask at some point, for example, as done in sketching validations such as Poplar [BBC21], which requires additional care.

B.2.4 Hiding the number of valid inputs

Depending on the protocol, it may be of interest to hide the number of valid inputs. Note that some protocols such as Mean [CB17, AGJ21] (which is done by sum divided by total) rely on the total for accumulation, so additional care is needed. Suppose there is a single share $[valid]^A$, which is 0 if it is invalid, and 1 if it is valid. Also suppose that all accumulator updates are additive via arithmetic shares, as in this work, and most protocols in Prio [CB17] and Prio+ [AGJ21]. Then the servers can spend an additional round multiplying $[valid]^A * [update]^A$ to get a share that only updates if the input is valid. The interesting question though is in obtaining an arithmetic share of the validity with the 0/1 property. The above section allows for combining multiple validity shares locally, that's likely correct, and is 0 for valid inputs and nonzero for invalid. This however requires additional work to make it exactly 0/1 as desired, for example using comparison with say $(1 - [x < 0])(1 - [x > 0])$ to be 1 exactly when it is 0, which takes additional rounds. Alternatively, if validity can be obtained as a Boolean instead, then $[valid]^B$ can be multiplied by $[update]^A$ using either OT or share conversion. However, as noted above most validity checks are if an arithmetic share is equal to a value, which doesn't have a straightforward way of mapping to a Boolean share.

B.3 Heavy Hitters Extensions

This section covers future work and alternatives considered for the top K algorithm.

B.3.1 Different guarantees

As noted, the algorithm assumes any possible distribution, so is generous on parameters for that reason. If additional information is known about the distribution, the parameters can be improved. Prior works for example have parameters that rely on f_K or the tail $F_2^{>K}$ of items past the K^{th}. For example, SpaceSaving [MAE05] and CountSketch [MAE05] are streaming algorithms maintain an updating of values, but the size of these sets depend on the distribution. In addition, updating a live list of values would require conditional logic, making accumulation and batching nontrivial to do efficiently. Similarly, if there is known to be a large "gap" of frequencies between elements, then SingleHeavy has better odds of finding items, meaning algorithms have a better chance of identifying the more frequent tier of values.

B.3.2 Alternative shares

There are various other ways the shares for Top K can be built, with their own tradeoffs.

Bucket split: Currently, a pair of buckets gives 1 bit of information, and takes 2 bits to encode (1 for which, 1 for sign). This can be changed to a set of 4 buckets, where the value goes to one of the four. This gives 2 bits of information, and takes only 3 bits to encode (2 for which, 1 for sign). This would require less values moving around for the same amount of accumulator space. Earlier versions sent $h(x)$ and $s(x)$, which means this requires 3 bits sent over 4 for the SingleHeavy shares, but this is less applicable now that the servers can compute it. This also requires new math to prove the accuracy. Further groups of buckets then starts taking up more total accumulator space, which could be worth, but requires additional investigation to determine the optimal trade off.

Re-use randomness: One avenue to investigate is reducing share size by re-using randomness. For example, if the different SingleHeavy randomness can be re-used across the Q iterations, with just different B substreams each time. The idea is that the different substreams might be enough for heaviest items to be alone in one of the iterations, and SingleHeavy randomness can be re-used on these substreams. This will improve the server work slightly, as there is less that needs to be validated (that they are frequency vectors). The work during actual accumulation doesn't change much though, as the servers will just do the multiplication against more repeated values, rather than the different values.

B.3.3 Alternative aggregation order

In chapter 6, there are 3 masks (layer, stream, pair) along with the value (± 1). We currently Boolean multiply the four values first in two rounds, then do B2A at the end. An alternative approach is to do B2A first. This requires less share conversion, and since it's direct conversion of user data it can use the techniques of subsection B.2.2 to further improve it. Then, masking is done afterwards using OT rather than Boolean multiplication, which is larger messages but can take advantage of OT extension [ALS13, IKN03] to have easier precomputation. This also requires 3 rounds rather than 2 to do the multiplication of the arithmetic share of ± 1 by the three masks.

B.3.4 Early reveals during evaluation

In some places during evaluation of top K, revealing some of the aggregate information early can provide notable efficiency benefits, at the cost of revealing additional minimal aggregate information. Since this is during evaluation, it reveals nothing about individual inputs besides what can be inferred by the aggregate info (e.g. there exists one or more inputs with these values, but not which).

Candidate reveal For slightly weaker privacy, the servers can reveal the candidates to each other early, before the frequency queries. The advantage of this is that it allows duplicate candidates to be eliminated before the CountMin, reducing the number of queries needed.

The lack of duplicates also allows the final reveal to be one round, just revealing the top K values at once, without needing to check for duplicates. This also prevents a malicious server from trying to reveal extra values by faking duplicates.

This also allows for the full hashes to be evaluated before the circuits, with more complex operations like modulo. This will still leave their frequencies and relative ordering private, because they get "re-hidden" when they get sorted by hidden frequencies. Furthermore, this does not reveal anything about individual inputs. However, this does reveal additional information about the distribution, which may or may not be a valuable trade-off. The amount of actual information revealed is low, because the candidates is a mix of actual heavy values, real values that happened to win, and random values, and it's hard to distinguish between them. The main distinguishing is that if a value repeats, then it is more likely for that value to be heavy, but also since it's heavy it's likely to be an actual answer anyways. So the likely leakage is revealing values near the top K that were heavy but not heavy enough, and standalone values that are either a random client's input or an actual random value.

There may be more leakage if the actual input domain is sparse compared than the checked domain, for example strings directly mapped to bits. Then the random candidates are detectable as gibberish, while the input candidates are real words, which will leak additional information of some values that were likely submitted by some client (which is still hidden). However, if the inputs strings are mapped compactly to inputs (e.g. hash or dictionary), which is preferable anyways to reduce n, then gibberish candidates map to some reasonable input.

Frequency reveal. After Count-Min, the servers can reveal frequencies to each other, getting $([c]^G, \hat{f}_c)$, where $[c]^G$ represents the candidate being still hidden/shared by the garbled circuit. This allows them to sort using local information, which won't require any communication. Then, when iterating down items starting at most frequent, we can group together items with the same frequency to help with de-duping. However, this doesn't mix well with the candidate reveal. This is because before the sort the frequencies found are in the same order as the provided candidates, so if both are revealed then they can be matched up, revealing everything. Therefore, the pairs need to be shuffled at some point before the frequencies are revealed. This can be done by the garbled circuit, which loses the benefit of not needing to sort in the garbled circuit. However, this still allows for a single round reveal. Revealing frequencies of candidates may reveal information about the distribution. These values are for a non-uniform sample of unknown candidates, so what can be inferred about the full distribution is unclear.

BIBLIOGRAPHY

[AGJ21] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. “Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares.” Cryptology ePrint Archive, Report 2021/576, 2021.

[ALS13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. “More Efficient Oblivious Transfer and Extensions for Faster Secure Computation.” Cryptology ePrint Archive, Paper 2013/552, 2013.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. “The Space Complexity of Approximating the Frequency Moments.” *Journal of Computer and System Sciences*, **58**(1):137–147, 1999.

[BBC21] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. “Lightweight Techniques for Private Heavy Hitters.” Cryptology ePrint Archive, Paper 2021/017, 2021.

[BCI15] Vladimir Braverman, Stephen R. Chestnut, Nikita Ivkin, and David P. Woodruff. “Beating CountSketch for Heavy Hitters in Insertion Streams.” *CoRR*, **abs/1511.00661**, 2015.

[BCI16] Vladimir Braverman, Stephen R. Chestnut, Nikita Ivkin, Jelani Nelson, Zhengyu Wang, and David P. Woodruff. “BPTree: an l_2 heavy hitters algorithm using constant memory.” *CoRR*, **abs/1603.00759**, 2016.

[Bea92] Donald Beaver. “Efficient Multiparty Protocols Using Circuit Randomization.” In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO ’91*, pp. 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[BFO11] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. “Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority.” Cryptology ePrint Archive, Paper 2011/629, 2011.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation.” In *STOC ’88*, STOC ’88, p. 1–10, New York, NY, USA, 1988. Association for Computing Machinery.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. “Prio: Private, robust, and scalable computation of aggregate statistics.” In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 259–282, 2017.

[CCF04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. “Finding frequent items in data streams.” *Theoretical Computer Science*, **312**(1):3–15, 2004.

[CM05] Graham Cormode and Shan Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications." *Journal of Algorithms*, **55**(1):58–75, 2005.

[DFK06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. "Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation." In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pp. 285–304, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY - A framework for efficient mixed-protocol secure two-party computation." In *NDSS*, 2015.

[GO96] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs." *J. ACM*, **43**(3):431–473, may 1996.

[HKR12] Justin Hsu, Sanjeev Khanna, and Aaron Roth. *Distributed Private Heavy Hitters*, p. 461–472. Springer Berlin Heidelberg, 2012.

[HLL21] Changhui Hu, Jin Li, Zheli Liu, Xiaojie Guo, Yu Wei, Xuan Guang, Grigorios Loukides, and Changyu Dong. "How to Make Private Distributed Cardinality Estimation Practical, and Get Differential Privacy for Free." In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 965–982. USENIX Association, August 2021.

[IKN03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. "Extending Oblivious Transfers Efficiently." In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pp. 145–161, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[MAE05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Efficient Computation of Frequent and Top-k Elements in Data Streams." In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005*, pp. 398–412, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[MDC15] Luca Melis, George Danezis, and Emiliano De Cristofaro. "Efficient Private Statistics with Succinct Sketches." *CoRR*, **abs/1508.06110**, 2015.

[RW19] Dragos Rotaru and Tim Wood. "MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security." Cryptology ePrint Archive, Paper 2019/207, 2019.

[Sch80] J. T. Schwartz. "Fast Probabilistic Algorithms for Verification of Polynomial Identities." *J. ACM*, **27**(4):701–717, oct 1980.

[WJS19] Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. "Stormy: Statistics in Tor by Measuring Securely." In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, p. 615–632, New York, NY, USA, 2019. Association for Computing Machinery.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. "EMP-toolkit: Efficient MultiParty computation toolkit." `https://github.com/emp-toolkit`, 2016.

[Yao86] Andrew Chi-Chih Yao. "How to generate and exchange secrets." In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pp. 162–167, 1986.

[Zip79] Richard Zippel. "Probabilistic algorithms for sparse polynomials." In Edward W. Ng, editor, *Symbolic and Algebraic Computation*, pp. 216–226, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg.