Distributional Secure Merge

Gayathri Garimella, Srinivasan Raghuramam and Peter Rindal

Abstract. Secure merge refers to the problem of merging two sorted lists. The problem appears in different settings where each list is held by one of two parties, or the lists are themselves shared among two or more parties. The output of a secure merge protocol is secret shared. Each variant of the problem offers many useful applications.
The difficulty in designing secure merge protocols *vis-a-vis* insecure merge protocols (which work in linear time with a single pass over the lists) has to do with operations having to be *oblivious* or *data-independent.* In particular, the protocol cannot leak the positions of items of each list in the final merged list. On account of this, sorting-based secure merge protocols have been a common solution to the problem. However, as they introduce (poly)logarithmic overheads, there has been active investigation into the task of building (near) linear time secure merge protocols. Most recently, Hemenway et al. put forth a protocol for secure merge that does achieve linear communication and computation and a round complexity of $\mathcal{O}(\log\log n)$, where n is the length of the lists being merged. While this shows the feasibility of a linear time secure merge, it still leaves room for the design of a concretely efficient linear time secure merge.
In this work, we consider a relaxation of the problem where the lists are uniformly random. We show a secure merge protocol for uniformly random lists that achieves $\mathcal{O}(n\log\log n)$, i.e., near linear communication and computation and a round complexity of $\mathcal{O}(\log\log n)$, where n is the length of the lists being merged. Our protocol design is general and can be instantiated in a variety of settings so long as the building blocks (basic ones such as comparisons and shuffles) can be realized in said settings. Although we do not achieve the same asymptotic guarantees as Hemenway et al., our work is concretely efficient. We implement our protocol and compare it to the state of the art sorting protocols and demonstrate an order of magnitude improvement in running times and communication for lists of size of 2^{20}.
We also extend our protocol to work for lists sampled from arbitrary distributions. In particular, when the lists are (close to) identically distributed, we achieve the same efficiency as uniform lists. This immediately improve the performance of many crucial applications including PSI & Secure Join, thus illustrating the significance and applicability of our protocol in practice.

1 Introduction

Secure merge refers to a class of cryptographic protocols that as input takes two secret-shared, sorted lists and outputs secret shares of a single combined, sorted list. These protocols should achieve this without leaking any information about the underlying lists apart from their lengths. Secure merge is closely related to the more general setting of secure sorting where there is no precondition on input list(s) being sorted. Both sorting and merging have numerous applications for privacy-preserving data processing. Just to name a few, machine learning techniques such as decision trees [19] need to sort the input features to identify candidate splits, protocols for Oblivious RAM [7], Private Set Intersection (PSI) [15] and secure join [4] all require sorting and/or merging as a building block.

In the plaintext setting merging two sorted lists of size n requires just $\mathcal{O}(n)$ time while sorting n elements requires $\mathcal{O}(n \log n)$ time in the comparison model. In the secret shared setting, the primary research effort has gone into inventing protocols for the more general problem of sorting. Initial efforts required $\mathcal{O}\left(n \log^2 n\right)$ time and were based on generic circuit-based MPC applied to the Batcher's bitonic sort circuit. However, since then several advances have enabled the overhead to be reduced to $\mathcal{O}(n \log n)$ cryptographic operations and communication, which is likely optimal. More recently, attention has turned to realizing comparable efficiency improvements in the case of secure merge, i.e. $\mathcal{O}(n)$ time & communication. Very recently, [10] achieved this with the so-called shuffle-then-merge paradigm. This technique makes sure of additively homomorphic encryption to emulate the classic merge sort algorithm. At each step the parties obliviously move the next smallest element from the input two lists to the output list. Unfortunately, their technique requires $\mathcal{O}(n)$ rounds of interactions which for many applications makes the protocol impractical. In particular, practical protocols should ideally require $\mathcal{O}(\log n)$ rounds of interactions for fewer.

From a theoretical perspective, this limitation was addressed by [11] which proposed a different paradigm for merging. The resulting protocol achieves $\mathcal{O}(n)$ computation and communication while requiring $\mathcal{O}(\log \log n)$ rounds of interaction. Unfortunately, we believe that this protocol would be inefficient in practice.

1.1 Applications

One of the most prominent applications of secure merge and sorting is private set interaction and secure joins. In its most basic formulation, two parties each hold a set X, Y respectively and want to identify the intersection $X \cap Y$. When $X \cap Y$ is output in plaintext, many extremely efficient protocols exist such as [21,9,16,6,28,29,17,30,26,27,31,12] with most of them not requiring sorting/merging. However, more advanced use cases require the secret sharing of $X \cap Y$ to be the output so that some additional computation can be performed in the intersection. A natural extension is to also allow associated values for each $x \in X, y \in Y$ to be included as part of the secret shared output.

In this way, a large set of SQL-like join queries can be performed on secret shared database tables where X, Y are the join keys and the associated values are the corresponding rows of the table. One of the current state of art protocols [24,4] for achieving this functionality builds on secure sorting as a building block. In particular, given the two input tables the first operation is to sort them into a combined table based on the join keys. Given this, it is relatively easy to generate a wide variety of joins such as inner, left, full, and union.

A large number of useful applications can be built on SQL-like joins. These range from privacy-preserving fraud detection [24], ad-conversation rates [16], data deduplication [20], oblivious RAM [7], graph algorithms [25] and many more. Essentially all large data processing tasks make use of some form of joins as a basic building block. Therefore, any improvement to the underlying join techniques translates to improvements to these end applications.

A very immediate optimization to the sorting-based join protocols is to have the input parties pre-sort these tables based on the join key and perform a secure merge instead. Unfortunately, at the time of the writing of the state of art protocol [4], the most concretely efficient merge protocol is to use a generic sorting protocol with $\mathcal{O}(n \log n)$ overhead. In this work, we aim to address this gap in performance.

1.2 Our Contributions

We propose a new secure merge protocol tailored for input lists where the marginal of each list is *uniformly* distributed. This restriction on the distribution of the inputs list in turn allows our protocols to be significantly more efficient than prior works. We then extend this protocol to merge arbitrarily distributed lists. For the case of arbitrarily but (close to) identically distributed lists, we are able to do this with virtually no loss in performance.

In particular, our main protocol takes as input two secret shared lists X, Y and outputs a secret shared permutation π and secret shared sorted list Z such that $Z_i = (X||Y)_{\pi_i}$. Important contributions that our construction provides include:

1. A flexible framework that can be implemented in various settings.
2. Concretely efficient merge for (close to) identically distributed lists.
3. Extensible for other input distributions.
4. First near-linear time and constant round circuit PSI and secure join in the two-party setting.
5. Malicious or semi-honest security.
6. Secret shared inputs and outputs.

Importantly, we show that the requirement that the input lists follow a known distribution is compatible with many of the most compelling applications, for example, when joining two tables or performing PSI on join keys such as usernames or emails that follow a well-known distribution. In addition, we give a general characterization of how to apply our techniques to lists that have arbitrarily different distributions as well.

1.3 Related Work

Secure Sorting Sorting is a widely used building block for data processing tasks. There are several leading paradigms for secure sorting. The first is the so-called "shuffle-then-sort" technique [14,13] where the parties first obliviously shuffle the list so that no one knows the order. Any traditional comparison-based sorting algorithm can then be performed where each plaintext comparison is replaced with a secure comparison with private inputs but *public* output. That is, all parties know the plaintext result of all comparisons. This in turn allows the parties to reorder the shared according to the sorting algorithm being used. Sorting algorithms such as quick-sort with a secret-sharing type MPC gives expected $\mathcal{O}(\log n \log \ell)$ round complexity and $\mathcal{O}(n\ell \log \ell)$ computational/communication complexity, where ℓ is the bit length of the items being sorted.

More recently, [8] presented a new protocol for performing radix sort which does not require shuffling the input list. They present a sub-protocol for obliviously sorting single-bit elements. In particular, their protocol outputs a secret sharing of a permutation that would sort the single-bit elements. One can then compose this protocol so that you obliviously sort the most significant bit down to the least significant bit. Their protocol requires $\mathcal{O}(\ell)$ rounds and $\mathcal{O}(n\ell \log \ell)$ communication. Asymptotically this protocol is preferred when $\log n > \frac{\ell}{\log \ell}$ which is typically the case.

Secure Merging One of the first protocols designed for merge is [7]. Their protocol also follows the "shuffle-then-sort" [14,13] paradigm but with sorting replaced with one iteration of merge-sort. In particular, the parties first construct a special linked-list structure for each input list and then shuffle the linked-lists. The parties can then run an essentially plaintext merge sort algorithm on the shares where the parties obliviously select the next element to include by revealing the location that it was shuffled to. In particular, the parties maintain a secret shared version of the head of the two linked lists. The smaller head is merged into the final list and the index of its shuffled child is revealed. The parties then update this merged node with its child and the process is repeated.

The protocol of [7] was originally designed for the 4-server ORAM setting. More recently, [10] adapted this protocol to the two-party setting. Their protocol makes use of additively homomorphic encryption to perform the shuffle and the construction of the linked list. One fundamental limitation of this approach is that the nodes in the linked list must be revealed in an iterative manner. Revealing the node requires at least one round of interaction and therefore their protocol requires a total of $\mathcal{O}(n)$ rounds of interaction. For large lists, e.g. $n = 2^{20}$, this makes their protocol less time efficient than the previous sorting protocols when network latency is considered.

An alternative approach was given by [11] where merging is performed in two phases. They divide the sorted input lists into blocks of poly-logarithmic size blocks. These blocks are then merged together based on their first elements. Since there are few blocks, this can be done relatively efficiently. Given that the blocks are now in sorted order, what remains is to locally sort the "strays" that are out of order. Our work will also make use of this general strategy.

Recently, [5] put forth a protocol that takes $\mathcal{O}(n)$ computation and communication and $\mathcal{O}(\log \log n)$ rounds relying on the same assumptions, designing a protocol that is asymptotically optimal in terms of communication and computation. While they also build on the framework of [11], considering dividing into blocks, merging them, and reorganizing strays, their protocol is rather complicated and makes use of several cleverly designed protocols that are tailored to be efficient in different parameter regimes. They then show how to compose these different protocols to achieve the earlier stated parameters. Asymptotically, this result is rather intriguing, but the concrete efficiency of the protocol seems uncertain at best.

2 Preliminaries

2.1 Notation

For integers $m, n \in \mathbb{Z}$, we denote by $[m, n]$ the set of integers $\{m, m+1, m+2, \ldots, n\}$ and $[n]$ be the shorthand for $[1, n]$. We define a permutation of size $N \in \mathbb{N}$ as a bijective function $\pi : [N] \to [N]$. We use η to denote the statistical security parameter, e.g., $\eta = 40$. Parties are denoted as $P_1, P_2, \ldots$.

We denote the results of comparison operators using $\{\cdot\}$. For instance, $\{2 \overset{?}{>} 3\} = 1$ and $\{2 \overset{?}{=} 3\} = 0$.

We denote by $[\![x]\!]$ a secret sharing of the value x, and by $[\![x]\!]_i$ the share held by party P_i. We assume values come from a field $\mathbb{F}$.

Following [5], for any two sorted lists L_1 and L_2, let $\bigsqcup$ denote the "merge" of two lists (i.e., $\bigsqcup$ is functionally equivalent to (multi-)set union followed by sort): $L_1 \bigsqcup L_2 = \textsc{sort}(L_1 \cup L_2)$. We will reserve the notation L_1, L_2 for the sorted lists that are to be merged, and denote $L_i = \{\ell_{i,1}, \ldots, \ell_{i,|L_i|}\}$ for $i \in [2]$, with $|L_1| = n$ and $|L_2| = m$. For any sorted list L_i of size N, and for any $k|N$, let $\mathcal{M}_{i,k}$ denote the k "medians" of L_i . Namely, $\mathcal{M}_{i,k} = \{\ell_{i,j\cdot\frac{N}{k}}\}_{j=1}^{k}$. For any two lists $X = (x_1, \ldots, x_N)$ and $Y = (y_1, \ldots, y_N)$ of equal length N, the *zip* of X and Y is $X \bowtie Y = ((x_1, y_1), \ldots, (x_N, y_N))$. We assume that all list elements come from an appropriate field $\mathbb{F}$ and that the zero element $0 \in \mathbb{F}$ is reserved as a special *dummy* element.

2.2 Distributions

Definition 1 (Poisson). *A discrete random variable X is said to have a Poisson distribution with parameter $\mathbb{E}[X] = \lambda > 0$ if it has a probability mass function given by*

$$\Pr[X = x] = \frac{\lambda^x e^{-\lambda}}{x!}$$

Definition 2 (Normal). *A continuous random variable X is said to have a Normal distribution $\mathcal{N}(\mu, \sigma)$ with parameters $\mu, \sigma > 0$ if it has a probability density function given by*

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

2.3 Definitions and Tools for Concentration

Definition 3 (Lipschitz condition). *A function f is said to be c-Lipschitz is $f(x) - f(y) \leq c|x - y|$ for all x, y.*

Lemma 1. *If a function f is c-Lipschitz with respect to hamming distance, then $||D_i f||_\infty \leq c$ for all i.*

Lemma 2 (McDiarmid's Inequality [3]). *Let $X_1, X_2, \ldots, X_m$ be independent random variables where X_i is $\mathcal{X}_i$-valued for $i \in [m]$. Let $X = (X_1, X_2, \ldots, X_m)$. Assume $f : \mathcal{X}_1 \times \ldots \times \mathcal{X}_m \to \mathbb{R}$ is a measurable function such that $||D_i f||_\infty < \infty$ for all i. Then for all $\gamma > 0$,*

$$\Pr[|f(X) - \mathbb{E}[f(X)]| \geq \gamma] \leq 2e^{-\frac{2\gamma^2}{\sum_{i\in[m]} ||D_i f||_\infty^2}}$$

If f is c-Lipschitz with respect to hamming distance, then

$$\Pr[|f(X) - \mathbb{E}[f(X)]| \geq \gamma] \leq 2e^{-\frac{2\gamma^2}{mc^2}}$$

Lemma 3 ([1]). *If $X_1, X_2, \ldots X_n$ are independent Poisson random variables with parameters $\lambda_1, \lambda_2, \ldots \lambda_n$, then $X_1 + X_2 + \ldots + X_n$ is Poisson random variable with parameter $\lambda_1 + \lambda_2 + \ldots + \lambda_n$.*

Lemma 4 ([2]). *If X is a Poisson random variable with parameter $\mathbb{E}[X] = \lambda > 0$, then for all $x > 0$,*

$$\Pr[X \leq \lambda - x] \leq e^{-\frac{x^2}{2(\lambda+x)}}$$

Lemma 5 (Probability Integral Transform). *Let X be a random variable with a continuous distribution for which the cumulative distribution function (CDF) is F_X. Then the random variable Y defined as $Y := F_X(X)$ has a standard uniform distribution.*

Proof. Given any random continuous variable X, define $Y = F_X(X)$. Given $y \in [0, 1]$, if $F_X^{-1}(y)$ exists (i.e., if there exists a unique x such that $F_X(x) = y$), then:

$$\begin{aligned} F_Y(y) &= \Pr[Y \leq y] \\ &= \Pr[F_X(X) \leq y] \\ &= \Pr[X \leq F_X^{-1}(y)] \\ &= F_X(F_X^{-1}(y)) \\ &= y \end{aligned}$$

If $F_X^{-1}(y)$ does not exist, then it can be replaced in this proof by the function χ, where we define $\chi(0) = -\infty$, $\chi(1) = \infty$, and $\chi(y) \equiv \inf\{x : F_X(x) \geq y\}$ for $y \in (0, 1)$, with the same result that $F_Y(y) = y$. Thus, F_Y is just the CDF of a uniform random variable over $(0, 1)$, so that Y has a uniform distribution on the interval $[0, 1]$.

2.4 Definitions and Tools for Oblivious Merge

Details of many of the tools described in this section can be found in [5]. We provide a summary for the sake of completeness.

Primitive Functionalities Our protocols are realized with black box calls to the following "primitive" functionalities: $\Pi_{\mathbf{Open}}$, $\Pi_{\mathbf{Comp}}$, $\Pi_{\mathbf{Sel}}$, $\Pi_{\mathbf{Shuffle}}$. They act on shared values. We briefly describe these functionalities here and note that they can be realized using standard techniques.

- In $\Pi_{\mathbf{Open}}$, the parties allow each other to learn the shared value. See [23] or similar.
- In $\Pi_{\mathbf{Comp}}$, the parties learn secret shares of a bit denoting the result of a comparison operator $[\![\{x \overset{?}{>} y\}]\!]$, $[\![\{x \overset{?}{\geq} y\}]\!]$, $[\![\{x \overset{?}{=} y\}]\!]$, or $[\![\{x \overset{?}{\neq} y\}]\!]$. See [23] or similar.
- In $\Pi_{\mathbf{Sel}}$, the parties perform multiplication of two values, one a bit b that is shared as $[\![b]\!]$, and the other a value x that is shared as $[\![x]\!]$. Equivalently, the parties compute secret shares of the ternary operator $b?x : 0$. See [23] or similar.
- In $\Pi_{\mathbf{Shuffle}}$, the parties begin with secret shares of a given list, and end up with secret shares of the same list, in some totally unknown order.
    - We will occasionally also need to unpermute a list related to (perhaps a modified version of equal length) a list that was permuted by $\Pi_{\mathbf{Shuffle}}$. For this, we assume a variant of $\Pi_{\mathbf{Shuffle}}$, where the parties begin with secret shares of a given list, and end up with secret shares of the same list, in some totally unknown order, and secret shares of a handle $[\![\mathsf{handle}]\!]$. We also assume a protocol $\Pi_{\mathbf{Unshuffle}}$ where parties begin with secret shares of a handle $[\![\mathsf{handle}]\!]$ and secret shares of a list of size the same as the one that was shuffled to get $[\![\mathsf{handle}]\!]$, and end up with secret shares of the list unpermuted. See [8].

We also assume a generic circuit-based MPC functionality $\mathcal{F}_{\mathbf{MPC}}$ that can apply a circuit C to shared inputs to obtain shared outputs. One can assume the GMW protocol.

Tag-shuffle-reveal Paradigm We make use of the *tag-shuffle-reveal* paradigm. In the tag-shuffle-reveal paradigm, each element of a list is (obliviously) tagged with some label. This label can be (a secret sharing of) its current index, or it can be the result of some multiparty computation, for example, a bit representing the output of a comparison against another value. Then, after shuffling the list, the tag or some part of the tag is opened, and the list entries are rearranged accordingly. Because the shuffle step ensures that the tags are randomly ordered, the only requirement to ensure security is to ensure that the set of values the opened tags take on does not depend on the underlying data. This paradigm is used to realize extraction functionalities, described ahead.

Extraction Functionalities Our protocols are realized with black box calls to the "extraction" functionalities $\Pi_{\textbf{Extract-Ord}}$ and $\Pi_{\textbf{Extract-Bin}}$ [5]. They act on secret shared values. We briefly describe these functionalities here and note that they can be realized using the tag-shuffle-reveal paradigm.

- In $\Pi_{\textbf{Extract-Ord}}$, the parties hold secret shares of a list with a certain number of marked elements, and end up with secret shares of a list of the marked elements in the order in which they appeared in the original list.
- In $\Pi_{\textbf{Extract-Bin}}$, the parties hold secret shares of a list with a certain number of marked elements, where the marking can be of many different types, and end up with secret shares of lists of the marked elements of each type in the order in which they appeared in the original list.

Trivial Secure Merge Functionalities Our protocols make use of the naïve secure merge protocol that simply performs all pariwise possible comparisons (securely) for terminating iterative/recursive processes when the reduced list sizes are sufficiently small. We call this protocol $\Pi_{\textbf{SM-ALL}}$. Another option would be to use any protocol that simply sorts the concatenation of the two lists. We call this protocol $\Pi_{\textbf{SM-Sort}}$.

3 Abstract Secure Merge

Suppose, Alice and Bob each have a private, sorted list of items L_1, L_2 with $|L_1| = n, |L_2| = m$ and jointly want to learn the combined, sorted list. We propose a recursive framework for Secure Merge ($\textsf{SM}(n,\ m)$) with the following high-level approach:

1. (*Base case* of $\textsf{SM}(n,\ m)$) If n, m are small enough, Secure Merge ($\textsf{SM}(n,\ m)$) via $\Pi_{\textbf{SM-ALL}}$ or $\Pi_{\textbf{SM-Sort}}$.
2. Partition input L_1 into k *blocks*, each of size $\frac{n}{k}$, for some $k|n$. Identify and store the last element of every block into a separate list of medians $\mathcal{M}_{1,k}$.
3. Securely merge the medians $\mathcal{M}_{1,k}$ with the list L_2 using a secure asymmetric merge protocol ($\textsf{asym-SM}(k,\ m)$) (see Section 3.1). The merge positions of the medians $\mathcal{M}_{1,k}$ divides L_2 into (potentially unequal sized) *chunks* and we end up with k smaller sub-problems. In each sub-problem, elements from a block of L_1 must be merged with its corresponding chunk in L_2.
4. Determine a suitable upper bound $B_{\max}$ on the size of all the chunks and pad each chunk[1] with dummies (upto $B_{\max}$) for obliviousness. Each padded sub-problem is *extracted* into its own instance and processed independently.
5. (*Recursive step*) Secure Merge ($\textsf{SM}(\frac{n}{k},\ B_{\max})$) each of the k blocks with its corresponding padded chunk.

[1] It is possible that we compute a different $B_{\max}$ for different chunks–more on this in Section 3.2.

The main idea of the framework is to divide the sorted list L_1 into k same-sized blocks and identify each block by its last element, called a *median* (which is an upper bound for all elements in the block). Next, we merge the block identifiers or medians with the input list L_2. This step demarcates the start and end positions in L_2 where elements of each block of L_1 will merge with L_2; we refer to each such *non-overlapping* sub-list of L_2 as a *chunk*. It is worth clarifying here for ease of reading that we refer to equal sized sub-divisions of a list (as we did with L_1) as *blocks*, and induced (potentially unequal sized) sub-divisions of a list (as we did with L_2) as *chunks*.

The goal is to have k sub-problems that can be solved recursively until the problem is small enough for the "all-pairs merge" (compare every pair of elements) using $\Pi_{\mathbf{SM\text{-}ALL}}$ to be efficient. However, the issue is that the sizes of each chunk may leak information about the input lists. Therefore, we need an extra step to select a suitable upper bound $B_{\max}$ on the sizes of the chunks to make the sub-problems oblivious (input-independent).

Observe that with no restrictions on the inputs L_1, L_2 it is difficult (actually, impossible) to find a reasonable upper bound on the chunk size. It might be the case that the first block identifier (last element of the first block) has a larger value than the last element of list L_2. In this skewed scenario, the upper bound on the chunk size is the size of the entire list L_2. Therefore, we work with a restricted class of inputs amenable to efficient sub-protocols for asymmetric merge and efficient upper bounds on the chunk size.

Uniform Inputs. In the rest of this section, we restrict ourselves to the case when the input lists L_1, L_2 are sampled from the uniform distribution[2]. We utilize this assumption to design an efficient merge protocol for the medians of one of the list with all the elements of the other list. Based on the uniform distribution, we can determine the expected *chunk* size. We can then analyze the tail distribution of the chunk sizes to bound their worst-case behavior and determine a reasonable upper bound $B_{\max}$ on chunk size. We discuss this aspect in Section 3.2.

3.1 Asymmetric Secure Merge with Uniform Lists (Steps 2–3)

We design an efficient **asym-SM**(k, m) protocol to merge a small list of k medians $\mathcal{M}_{1,k}$ of L_1 of size n with a significantly larger list L_2 of size m, where L_1, L_2 are sampled from the uniform distribution. In expectation, an element at index i from L_1 will merge with the other list L_2 at position $i \cdot \frac{m}{n}$. However, we must account for the strays where an element at position i merges with the other list elsewhere. We would like to compute some reasonable bound β such that the element at position i in L_1 merges with L_2 in a position within some range $\left[\frac{im}{n} - \frac{\beta}{2}, \frac{im}{n} + \frac{\beta}{2}\right]$ of indices with overwhelming probability. In the next section, we show using a balls-and-bins analysis that $\beta = \mathcal{O}\left(\sqrt{m \ln k}\right)$ suffices. Therefore, to merge a single element from list L_1 into list L_2 we need to make β

[2] Over some fixed support. Additionally, their joint distribution need not be uniform.

PARAMETERS:

- List of k medians $\mathcal{M}_{1,k}$ of a list L_1 of size n sampled from the uniform distribution
- Sorted list L_2 of size m sampled from the uniform distribution
- $\beta = \sqrt{2m((\eta+1)\ln 2 + \ln k)}$

PROTOCOL:

- Let $\mathcal{M}_{1,k} = \{m_1, \dots m_k\}$ and $L_2 = \{\ell_{2,1}, \dots, \ell_{2,m}\}$. For every $i \in [k]$:
  - Compare, using $\Pi_{\mathbf{Comp}}$, m_i with each element of the sub-list $\left\{\ell_{2,\frac{im}{k}-\frac{\beta}{2}}, \dots, \ell_{2,\frac{im}{k}+\frac{\beta}{2}}\right\}$ of L_2 and find the first index B_i where $M_i > \ell_{2,B_i}$.

Fig. 1: Outline of the Asymmetric Merge **asym-SM**$(k,\, m)$ protocol.

comparisons and require $\mathcal{O}\left(\sqrt{m \ln k}\right)$ communication and computation. Overall, as we want our merge protocol to have $\mathcal{O}(m)$ communication, we pick $k = \sqrt{\frac{m}{\ln m}}$ medians for our asymmetric merge.

We describe this outline of our asymmetric merge protocol in Figure 1 and perform a balls-and-bins analysis to estimate bounds on β in Section 3.1.

Choosing $\boldsymbol{\beta}$ In this section we will show via a balls-and-bins analysis how to find the number of comparisons β required to correctly merge any given element from $L_1 = \{\ell_{1,1}, \dots, \ell_{1,n}\}$ into $L_2 = \{\ell_{2,1}, \dots, \ell_{2,m}\}$. Let the elements of list L_1 define the divisions between bins and elements of list L_2 be balls that we throw uniformly into the bins. In the final merged list $L_1 \bigsqcup L_2$, we then look at the number of elements of L_2 that fall between $\ell_{1,i-1}$ and $\ell_{1,i}$ to be the number of balls that fall into the ith bin represented by $\ell_{1,i}$. Technically, there is one more bin for elements of L_2 that are larger than $\ell_{1,n}$, but we ignore this for ease of presentation, and note that it does not significantly change any of our analyses.

In expectation, we know that we have $\frac{m}{n}$ balls that fall into the ith bin. Now, we will compute concentration bounds for each bin, to learn the range β of the number of balls that fall into a particular bin with overwhelming negligible probability. Therefore, a ball can be at most $\pm\frac{\beta}{2}$ bins away from its expected position. In the context of merging lists, it means that an element $\ell_{1,i}$ of L_1 can fall behind by merging with L_2 at $\ell_{2,\frac{im}{n}-\frac{\beta}{2}}$ or be ahead by merging with L_2 at $\ell_{2,\frac{im}{n}+\frac{\beta}{2}}$. Note that when applied to the k medians of L_1, this translates to the sub-lists of L_2 compared with in Figure 1.

For our analysis, we will use the McDiarmid's Inequality (Lemma 2). Consider the experiment of throwing m balls into n bins uniformly and independently at random. Define random variables: $(P_1, P_2, \dots, P_m)$, where P_i is the bin where the ith ball lands. These variables are *independent*. Define random variables:

$(Q_1, Q_2, \ldots, Q_n)$, where Q_j is the number of balls that fall before and including in bin j.

For each bin j, $\mathbb{E}[Q_j] = j \cdot \frac{m}{n}$. Let f be the function such that the number of balls that fall before and including in bin j, $Q_j = f(P_1, P_2, \ldots, P_m)$, is a function of how each of the balls fell into the bins. Observe also that this function f is 1-Lipschitz. Simply put, if you change the value of one of the random variables $\{P_1, P_2, \ldots, P_m\}$, that is, move exactly one ball from one bin to another, then Q_j also changes by *at most* 1. Using McDiarmid's inequality, for all $j \in [n]$,

$$\Pr\left[\left|Q_j - j \cdot \frac{m}{n}\right| \geq \gamma\right] \leq 2e^{-\frac{2\gamma^2}{m}}$$

From this inequality, we can determine the probability that Q_j deviates from the expectation by more than γ, and we will finally set $\beta = 2 \cdot \gamma$. We are interested in finding the smallest possible γ, such that the probability of any of the k medians $\mathcal{M}_{1,k}$ of L_1 deviating from their expected positions by more than γ is negligible. So, using a union bound, we equate k times the right side with $2^{-\eta}$, where η is the statistical security parameter. This gives us

$$\gamma = \sqrt{\frac{m}{2}((\eta + 1)\ln 2 + \ln k)}$$

Effectively, our range size is $\beta = 2 \cdot \gamma = \mathcal{O}\left(\sqrt{m \ln k}\right)$.

Therefore, in order to merge the k medians $\mathcal{M}_{1,k}$ of L_1 with L_2, we need $k \cdot \mathcal{O}\left(\sqrt{m \ln k}\right)$ comparisons. Setting $k = \sqrt{\frac{m}{\ln m}}$, this ends up being $\mathcal{O}(m)$ comparisons in total. We present the outline of this protocol in Figure 1. Note that the protocol only takes $\mathcal{O}(1)$ rounds.

3.2 Padding: Choosing $B_{\max}$ (Steps 4–5)

We now look at how to determine a reasonable upper bound $B_{\max}$ on the chunk sizes. Note that the chunk sizes would have been $\frac{m}{k}$ in expectation, but they may deviate from this. While we can find the chunk sizes explicitly, we cannot leak them as this destroys obliviousness as this information is input-dependent. To work around this, we pad each chunk with *dummy*[3] elements up to some size $B_{\max}$. In this section, we will first talk about how to set $B_{\max}$ so that we have no overflows with overwhelming probability (Section 3.2). Next, we will describe an empirical way of padding chunks while heuristically maintaining obliviousness and input-independence (Section 3.2).

Bounds via Poisson Approximation We will perform another balls-and-bins analysis, but with the perspectives flipped. Let the elements of list L_2 define the bin divisions and elements of list L_1 be balls that we throw uniformly into the bins. If we throw n balls into m bins, the random variables representing the

[3] We can assume we have the designated dummy element, 0.

number of balls in each bin are not independent. However, we can treat them like independent variables and use Poisson approximation to learn a slightly looser upper bound on the number of balls in a bin and use the Poisson approximation method to translate it to a bound for the event we are interested in [18,22]. The Poisson approximation method for analyzing an event in the balls-and-bins experiment proceeds as follows:

– Treat the balls being thrown into bins as independent Poisson random variables with parameter $\frac{n}{m}$.
– Compute the probability q of the event that we are interested in analyzing.
– Apply a penalty factor of $e\sqrt{n}$ for the approximation itself, that is, the probability of the actual event $p \leq e\sqrt{n} \cdot q$.
– The penalty factor can be reduced to 2 if q is monotonic.

The event that we are interested in analyzing is the following: How many contiguous bins B do we need to find t balls with overwhelming probability? Note that to find the size of a chunk in L_2 that corresponds to a block of size $\frac{n}{k}$ in L_1 via our balls-in-bins analysis, we must determine how many contiguous bins are needed in order to see a total of at least $\frac{n}{k}$ balls. This is why, looking ahead, we are interested in the case where $t = \frac{n}{k}$, but we perform the analysis in general. Also note that the advantage of looking at contiguous bins as opposed to a single one and hoping to get strong concentration bounds is well-founded as even though a single bin may be off by a lot from its expectation, a contiguous set of bins may not be off by much more (and thus relative to the number of bins we consider, the average deviation is not much)–in other words, the "curve" is smooth and slow, not sharp.

Define Q_j to be the number of balls in the j'th bin with $\mathbb{E}[Q_j] = \frac{n}{m}$. Define R_j to be the number of balls in the contiguous bins indexed by $j, \ldots, j + B - 1$. Note that $\mathbb{E}[R_j] = B \cdot \frac{n}{m}$. If we assume that the Q_j's are independently Poisson, then the R_j's are Poissonas well, as $R_j = Q_j + \ldots + Q_{j+B-1}$ (Lemma 3).

Let $\lambda = \mathbb{E}[R_j] = B \cdot \frac{n}{m}$. From Lemma 4 and incorporating the penalty factor 2, we have that for all $t \leq \lambda$,

$$\Pr[R_j \leq t] = \Pr[R_j \leq \lambda - (\lambda - t)] \leq 2e^{-\frac{(\lambda-t)^2}{2(2\lambda-t)}}$$

We will obtain a similar bound for each of k sets of balls (corresponding to the k blocks of L_1) that we consider, and so the net failure probability, using a union bound, is at most k times the above bound. We would like for that bound to be negligible, that is, at most $2^{-\eta}$, where η is the statistical security parameter. Thus

$$e^{\frac{(\lambda-t)^2}{2(2\lambda-t)}} \geq e^{(\eta+1)\ln 2 + \ln k}$$

$$\lambda^2 - 2\lambda(t + 2(\eta+1)\ln 2 + 2\ln k) + 2t((\eta+1)\ln 2 + \ln k) + t^2 \geq 0$$

Let

$$\theta = 2(\eta+1)\ln 2 + 2\ln k$$

The roots of the above quadratic are

$$\begin{aligned}\lambda^-, \lambda^+ &= (t+\theta) \pm \sqrt{(t+\theta)^2 - t(t+\theta)} \\ &= (t+\theta) \pm \sqrt{\theta}\sqrt{t+\theta} \\ &= \sqrt{t+\theta}(\sqrt{t+\theta} \pm \sqrt{\theta})\end{aligned}$$

So, $\lambda \leq \lambda^-$ or $\lambda \geq \lambda^+$. We also need $\lambda - t \geq 0 \iff \lambda \geq t$. So

$$\lambda \in ([0, \lambda^-] \cup [\lambda^+, \infty)) \cap [t, \infty]$$

We have

$$\lambda^- = (t+\theta) - \sqrt{\theta}\sqrt{t+\theta} \leq t$$

and so $\lambda^- \leq t$. Also, $\lambda^+ \geq t$. Therefore

$$\lambda \in [\lambda^+, \infty)$$

Recalling that $\lambda = B \cdot \frac{n}{m}$, we have

$$B \geq \frac{m}{n}\sqrt{t+\theta}(\sqrt{t+\theta} + \sqrt{\theta})$$

Therefore, if we look at $B_{\max} = \frac{m}{n}\sqrt{t+\theta}(\sqrt{t+\theta} + \sqrt{\theta})$ bins, we will see t balls with overwhelming probability. In particular, the chunk sizes corresponding to each of the k blocks is of size at most $B_{\max}$ with overwhelming probability, where $t = \frac{n}{k}$, and hence padding chunks of L_2 up to a size of $B_{\max}$ with dummy elements suffices.

We tabulate the values of $B_{\max}$ for various choices of symmetric lists $m = n$ in Table 1. Somewhat predictably, we notice that padding to $B_{\max}$ introduces only a moderate overhead of dummies when $m = n$ is large, however, for moderate to small values of $m = n$, the overhead becomes significant, even crossing a 100%! This means that as we get into the lower stages and smaller sub-problems of our recursion, we will introduce a large fraction of dummies. This is not desirable. As a way to work around this, we investigate padding chunks in accordance with the distribution that the chunk sizes are expected to follow. We discuss this ahead in Section 3.2.

Bounds via Empirical Estimation In Section 3.2, we determined a worst-case upper bound $B_{\max}$ on the chunk size. However, it is unlikely that every or even many of the chunks get to be as large as $B_{\max}$. Our goal is then to pad the chunks while exploiting this observation. Really, what we would like to know is the probability distribution of the chunk sizes. If we had a handle on this, we could pad up the chunks corresponding to the blocks so that the set of all chunk sizes together jointly respects the probability distribution of the chunk sizes. The reason we would like to look at this as opposed to settling for the analysis from before is that from Table 1, for moderate to small values of $m = n$, the overhead introduced by padding becomes significant. This means

$m = n$	k	$B_{\max}$	$B_{\max} - \frac{m}{k}$	% $kB_{\max}$ over m
2^{28}	3719	74554	2375	3.29%
2^{24}	1004	17870	1160	6.94%
2^{20}	275	4395	582	15.26%
2^{16}	77	1162	311	36.53%
2^{12}	22	375	189	101.42%

Table 1: Values of $B_{\max}$ and the excess padding introduced as a result of the choice of $B_{\max}$; we have set $\eta = 40$.

that as we get further down our recursion, the number of dummies we introduce may even vastly outnumber the elements themselves, which would be damaging to concrete performance.

However, it seems technically difficult to analytically capture the joint distribution of the chunk sizes. Therefore, we resort to simulations and empirical estimations for the joint distribution of the chunk size. From our simulation results in Figure 2, we see that the joint distribution of the chunk sizes is very tightly approximation by a normal distribution with parameters

$$\hat{\mu} = \frac{m}{k}, \hat{\sigma} = \sqrt{\frac{m}{k}\left(1 - \frac{1}{k}\right)\left(1 + \frac{m}{n}\right)}$$

While we will not formally prove this claim, we note that it is in accordance with what we would expect if the chunk size were all drawn independently at random (parameters of a binomial or multinomial distribution), except for a "correction" factor of $1 + \frac{m}{n}$. This correction factor, we believe, accounts for abberations resulting from the lists that end up being sampled and how they impact chunk sizes. For instance, if n were really small compared to m, then the blocks of L_1 would spread somewhat non-uniformly across L_2 with reasonable probability. On the other hand, if $n \gg m$, in particular, if $n \to \infty$, we expect a distribution that exactly follows the balls-and-bins experiment, namely, a binomial or multinomial distribution. It is also worth noting that the results in Figure 2 do show a behavior closer to a binomial or multinomial distribution than a normal distribution. In particular, the distribution is skewed and the normal approximation falls just as short as it does when we try to approximate a binomial distribution using a normal distribution. However, as expected, for larger values, this approximation quality improves and so, with reasonable slack, we can use this normal distribution to approximate the joint distribution of the chunk sizes.

To further concretize the padding schemes that we can utilize, we look at the distribution of chunk sizes and compare it with the behavior of the normal distribution $\mathcal{N}(\hat{\mu}, \hat{\sigma})$. We performed thorough simulations, some of the results of which appear in Table 2. We considered various values of $m = n$ and studied the distribution of the chunk sizes that fell within $2\hat{\sigma}$ of $\hat{\mu}$. Accounting for the slight deviations for smaller values of $m = n$ on account of the skewedness of the

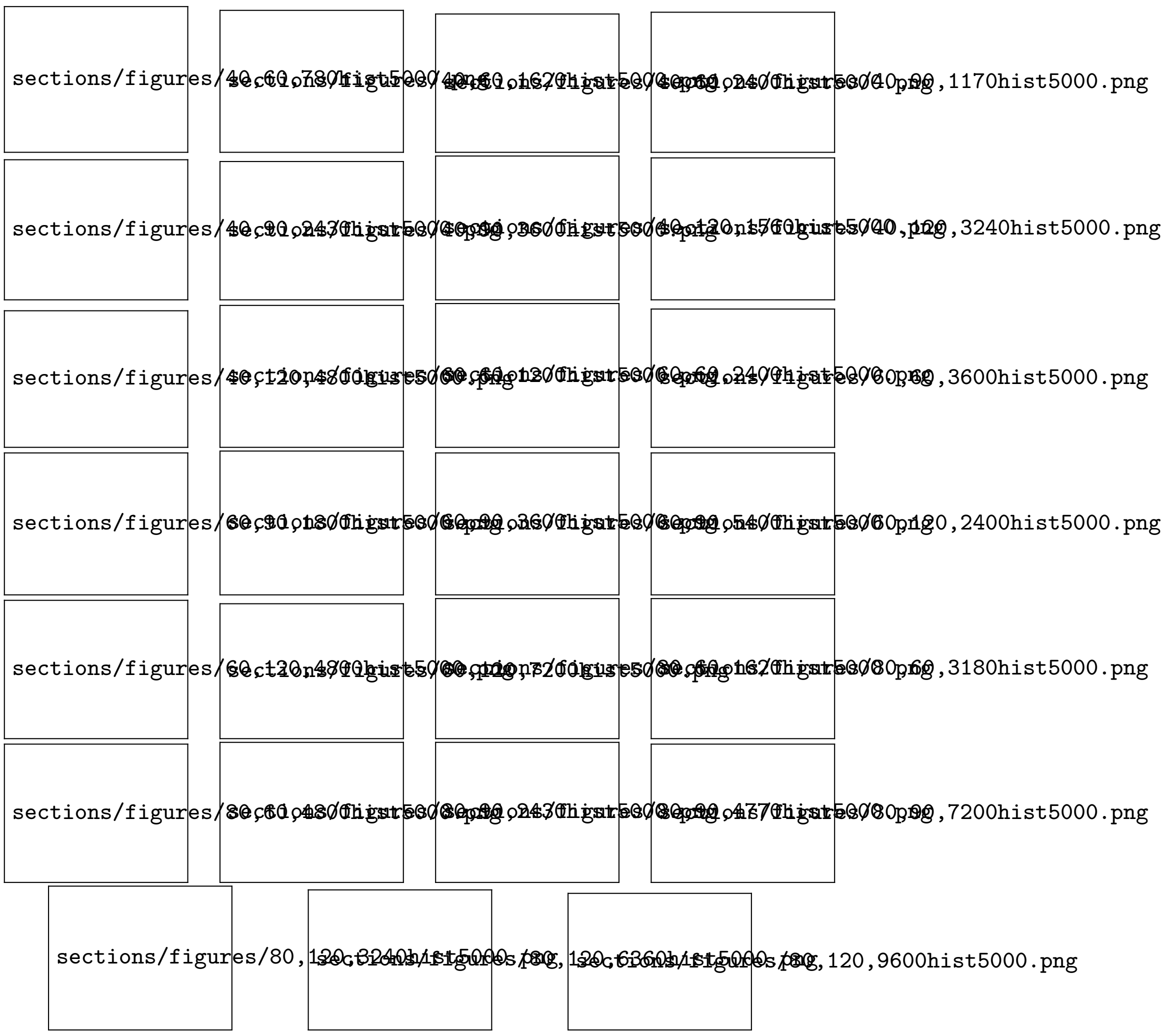

Fig. 2: Empirical joint distribution of the chunk sizes for various values of $n \geq m$, and k (similar results are observed for $n < m$), averaged over 5000 iterations. The observed values of the mean and standard deviations of the chunk sizes μ and σ match $\hat{\mu}$ and $\hat{\sigma}$ as defined before near exactly. The red line plots the probability density function of $\mathcal{N}(\hat{\mu}, \hat{\sigma})$, and it envelopes and approximates the observed empirical distribution near exactly.

$m = n$	k	$B_{\max}$	$\hat{\mu}$	$\hat{\sigma}$	$[0, \hat{\mu} - 2\hat{\sigma}]$	$(\hat{\mu} - 2\hat{\sigma}, \hat{\mu} - \hat{\sigma}]$	$(\hat{\mu} - \hat{\sigma}, \hat{\mu}]$	$(\hat{\mu}, \hat{\mu} + \hat{\sigma}]$	$(\hat{\mu} + \hat{\sigma}, \hat{\mu} + 2\hat{\sigma}]$	$(\hat{\mu} + 2\hat{\sigma}, B_{\max}]$	% Padding
2^{28}	3719	74554	72179	380	2.55%	13.42%	33.85%	33.96%	13.98%	2.24%	0.3%
2^{24}	1004	17870	16710	183	2.46%	13.6%	33.52%	34.18%	13.75%	2.49%	0.64%
2^{20}	275	4395	3813	87	2.17%	13.53%	34.48%	33.92%	13.4%	2.5%	1.33%
2^{16}	77	1162	851	41	1.93%	13.49%	34.79%	33.7%	13.39%	2.7%	2.9%
2^{12}	22	375	186	19	1.64%	13.42%	34.75%	33.38%	13.99%	2.82%	6.64%
2^{8}	7	173	36	8	1.35%	14.25%	31.35%	35.03%	14.59%	3.44%	17.42%
N/A	N/A	N/A	N/A	N/A	**2.2%**	**13.6%**	**34.1%**	**34.1%**	**13.6%**	**2.2%**	N/A

Table 2: Distribution of the chunk sizes that fall into the intervals $[0, \hat{\mu} - 2\hat{\sigma}]$, $(\hat{\mu} - 2\hat{\sigma}, \hat{\mu} - \hat{\sigma}]$, $(\hat{\mu} - \hat{\sigma}, \hat{\mu}]$, $(\hat{\mu}, \hat{\mu} + \hat{\sigma}]$, $(\hat{\mu} + \hat{\sigma}, \hat{\mu} + 2\hat{\sigma}]$, and $(\hat{\mu} + 2\hat{\sigma}, B_{\max}]$, as well as the overhead of padding based on these intervals, when averaged over 5000 iterations; we have set $\eta = 40$. The last row is a reference that lists the distribution of the probability mass for the normal distribution $\mathcal{N}(\hat{\mu}, \hat{\sigma})$.

$m = n$	$\frac{\text{bin width}}{\hat{\sigma}}$	# Bins	% Padding
2^{16}	2	1	5.27%
2^{16}	2	2	4.74%
2^{16}	1	3	2.38%
2^{16}	1	4	2.34%
2^{12}	2	1	11.32%
2^{12}	2	2	9.87%
2^{12}	1	3	4.99%
2^{12}	1	4	4.82%
2^{8}	2	1	28.18%
2^{8}	2	2	20.26%
2^{8}	1	3	9.92%
2^{8}	1	4	9.25%

Table 3: Overhead of padding based on varying number of intervals (bins) around $\hat{\mu}$ of varying sizes in terms of $\hat{\sigma}$, when averaged over 5000 iterations; we have set $\eta = 40$. As an example, Table 2 uses $\frac{\text{bin width}}{\hat{\sigma}} = 1$ and # Bins = 2. Further, $\frac{\text{bin width}}{\hat{\sigma}} = 2$ and # Bins = 2 uses the intervals $[0, \hat{\mu} - 4\hat{\sigma}]$, $(\hat{\mu} - 4\hat{\sigma}, \hat{\mu} - 2\hat{\sigma}]$, ..., $(\hat{\mu} + 2\hat{\sigma}, \hat{\mu} + 4\hat{\sigma}]$, and $(\hat{\mu} + 4\hat{\sigma}, B_{\max}]$, and $\frac{\text{bin width}}{\hat{\sigma}} = 1$ and # Bins = 3 uses the intervals $[0, \hat{\mu} - 3\hat{\sigma}]$, $(\hat{\mu} - 3\hat{\sigma}, \hat{\mu} - 2\hat{\sigma}]$, ..., $(\hat{\mu} + 2\hat{\sigma}, \hat{\mu} + 3\hat{\sigma}]$, and $(\hat{\mu} + 3\hat{\sigma}, B_{\max}]$.

real distribution, the distribution of the chunk sizes does very closely resemble $\mathcal{N}(\hat{\mu}, \hat{\sigma})$, which is to be expected given our results from Figure 2. If we pad chunk whose sizes are in the intervals up to the upper-bound of the interval, our overall overhead as a result of padding, i.e., the number of dummies we introduce, comes down drastically compared to the numbers we had in Table 1. This confirms our intuition that it is indeed a very small number of chunks whose sizes are large or close to $B_{\max}$. The point is that our empirical calculations give us a fairly strong handle on just how many of the chunks will end up having a large size. One can now evaluate whether these percentages (or some slightly noised version of them) can be considered public given the sizes of the lists, and hence whether padding according to them leaks any information. We believe that given the tight concentrations we see, it is indeed reasonable to pad up based on these intervals.

If one is more conservative or less conservative, we could vary the number and sizes of intervals we would like to consider. Indeed, the most conservative setting is the theoretical analysis from before that has a single interval size $[0, B_{\max}]$. We consider some other choices of intervals one could consider based on $\hat{\mu}$ and $\hat{\sigma}$ in Table 3. The results here are also as expected: (i) the padding is lesser for larger lists, presumably because of concentration; (ii) for a given list size and a given interval width, more intervals amounts to lesser padding; (iii) for a given list size and a given number of intervals, wider intervals result in more padding. These observations make a lot of sense in the context of an event that concentrates so well. Indeed, if we use more intervals, or we use less-wide intervals, we are leaking more information in a sense with respect to the particular instance of the random lists we are working with. Thus, it makes sense that they also end up needing lesser padding. And so, these statistics provide a way for one to choose precisely how much padding they are willing to handle and then assess whether the corresponding settings of intervals and their sizes constitute acceptable leakage in that context.

Regardless of how conservative one decides to be, if we at all pad adaptively, we are somewhat guaranteed that only one stage in the recursion introduces a lot of padding, approximately 20%, while all the others will introduce very little, say 0-5%. This means that, even for lists of size 2^{256}, our net padding will only be around 50%! This is significant compared to our results from Table 1 (also, Table 1 showed more than 100% padding for 2^{12} while we can now achieve less than 20% padding even for 2^8). This provides support for using adaptive padding schemes similar to the ones we suggest here in order to achieve concrete efficiency in practice, and all our other statistics provide ample evidence as to why such a strategy would result in minimal privacy loss.

For our full construction, we will assume the use of a padding scheme Pad which takes in chunk sizes and outputs a padded chunk size. Pad could be as simple as just outputting $B_{\max}$, or doing something more interesting like padding based on intervals like we have just discussed. Since the chunk sizes will not be revealed prior to padding, Pad will have to be invoked on secret shared data, which, looking ahead, we will do by invoking $\mathcal{F}_{\mathbf{MPC}}$.

3.3 Distribution of Inputs to a Recursive Call

One important detail we have not addressed is the following. Our entire analysis and choice of parameters hinges on the fact that the input lists are sampled uniformly. Is this true of the sub-problems we create when we recurse? Well, yes and no. Indeed, any block and chunk that we identify to merge do in fact satisfy our assumption of uniformity, however, we potentially pad our chunks with dummies. Thus, the sub-problems themselves do not satisfy the assumption. This is actually an issue as the windows where we will now have to search for the medians of one list in the other are now off on account of the dummies in the list. If we know how many dummies there were, we can remedy this by observing that it is in fact possible to quantify how much the expected position of where a median from the first list merges with the second list deviates on account of the dummies. Unfortunately, we cannot reveal the number of dummies.

To remedy this, we make the following observation. The maximum shift in the position we are looking for is exactly the number of dummies. Therefore, if we have a reasonable upper bound on the number of dummies in the list, we can simply expand our search window by that bound, and the guarantees we obtained from our analyses for the first level of the recursion will continue to apply for other levels too. The remaining piece of the puzzle is then to have an upper bound on the number of dummies we introduce, and in this regard, all the statistics we collected in the previous sections come in handy. We have fairly tight characterizations on the padding, i.e., number of dummies, that we introduce for the various padding schemes that we have proposed. We can use those estimates (and perhaps relax them appropriately) to serve as an upper bound on the number of dummies with high probability.

We formally denote this by a parameter $\alpha \geq 0$ in our full protocol. We assume that we have knowledge of the parameter α as a function of n, m, the level of the recursion, and the padding scheme that we use. For the outermost invocation of our protocol, $\alpha = 0$, and for further levels, α can be computed as a function of the sizes of the sub-problems.

4 Our Secure Merge Protocol for Uniform Lists

In this section, we will describe our protocol for uniform lists in complete detail. Before that, we look at the $\Pi_{\textbf{Extract-Bin}}$ algorithm, which mostly follows [5], but we make some changes to make it suitable for our purposes.

4.1 Adapting $\Pi_{\textbf{Extract-Bin}}$

We look at the stable bin extraction algorithm $\Pi_{\textbf{Extract-Bin}}$ from [5], where the parties hold:

- secret shares of a list A of elements marked with entries from $[0, k]$, with a certain number t'_j of contiguous elements marked by j for $j \in [k]$,
- an upper bound t such that $t'_j \leq t$
- secret shares of a list C that contains the starting indices in A of the block of elements marked by j for $j \in [k]$, and
- secret shares of a list T' that contains t'_j for $j \in [k]$

and end up with secret shares of lists B_j for $j \in [k]$ of elements marked j in the order in which they appeared in L, padded with dummy elements to make each list have a size of t.

We would like to adapt the protocol in the following respects:

- elements in A are marked with entries from $\{0, r_1, \ldots, r_k\}$ as opposed to $[0, k]$, where the parties hold $\{r_1, \ldots, r_k\}$,
- the list C contains the starting indices in A of the block of elements marked by j for $\{r_1, \ldots, r_k\}$,
- each block has its own upper bound t_j, i.e., $t'_j \leq t_j$ for $j \in [k]$, where the parties hold t_j for $j \in [k]$, and
- the list T' contains t'_j for $\{r_1, \ldots, r_k\}$

These adaptations can be performed using standard techniques and the ones outlined in [5]. We provide the complete protocol description in Figure 3 for completeness. Correctness, security, and efficiency of the protocol follow from the analyses in [5].

One final comment is that we have an upper bound on the number of dummies in any chunk, we can use this to make $\Pi_{\textbf{Extract-Bin}}$ more efficient in that Step 1 needn't append lists of size t_j, but rather, just lists as large as the upper bound on the number of dummies in each chunk. We however present the protocol without this modification and note that the protocol description can be modified fairly trivially to take this into account.

INPUT:

- Parties hold a list T of size k of elements of the form (t_j, r_j) for $j \in [k]$.
- Parties hold secret shares of a list A of size m of elements of the form (a_i, ι_i), with $\iota_i \in \{0, r_1, \ldots, r_k\}$, and $t'_j \leq t_j$ elements with $\iota_i = r_j$ for $j \in [k]$. We are guaranteed that all elements with $\iota_i = r_j$ are in a contiguous block, for $j \in [k]$.
- Parties hold secret shares of a pair of lists (C, T'), with the property that for each $j \in [k]$, c_j is the index i of the first element of A with $\iota_i = r_j$, and t'_j is the number of elements of A with $\iota_i = r_j$. If for any $j \in [k]$, no such element ι_i exists, $t'_j = c_j = 0$.

OUTPUT:

- Parties hold secret shares of a sequence of lists B_j for $j \in [k]$, where each list B_j has length t_j and contains all elements a_i with $\iota_i = r_j$, in the same order that they occur in A, followed by dummy elements as necessary.

PROTOCOL:

1. Append $\sum_{j\in[k]} t_j$ elements to A, where for each $j \in [k]$ and each $i \in [m + \sum_{p\in[j-1]} t_p + 1, m + \sum_{p\in[j]} t_p]$, we define
$$(a_i, \iota_i) = \left(0, r_j \cdot \left\{ i - \left(n + \sum_{p\in[j-1]} t_p \right) \overset{?}{>} t'_j \right\}\right)$$
This can be done, aside from local computations, by invoking $\Pi_{\mathbf{Comp}}$ and $\Pi_{\mathbf{Sel}}$ for each $j \in [k]$.
2. Generate the shared list $A' = A \bowtie K$, where K is the list of elements κ_i defined as
$$\kappa_i = \begin{cases} i & \text{if } i \in [m] \\ i \bmod t_j & \text{if } i \in \left[m + \sum_{p\in[j-1]} t_p + 1, m + \sum_{p\in[j]} t_p\right] \text{ for } j \in [k] \end{cases}$$
This can be done locally by P_1 setting their share equal to κ_i and all other parties setting their share equal to 0.
3. Invoke $\Pi_{\mathbf{Shuffle}}$ to compute $\hat{A}' = \Pi_{\mathbf{Shuffle}}(A')$.
4. Let $\hat{A}' = \hat{A} \bowtie \hat{K}$ where $\hat{A}$ has elements of the form $(\hat{a}_i, \hat{\iota}_i)$ and $\hat{K}$ has elements $\hat{\kappa}_i$. Open all values $\hat{\iota}_i$ by invoking $\Pi_{\mathbf{Open}}$. Delete the ith entries of $\hat{A}$ and $\hat{K}$ for every i such that $\hat{\iota}_i = 0$. Let the lengths of $\hat{A}$ and $\hat{K}$ after this be N.
5. For $i \in [N]$, let $j \in [k]$ by such that $\hat{\iota}_i = r_j$. Update the shared list $\hat{K}$ of elements $\hat{\kappa}_i$ as
$$\hat{\kappa}_i = \left(\hat{\kappa}_i - (c_j - 1) \cdot \{ \hat{a}_i \overset{?}{\neq} 0 \} \right) \bmod t_j$$
This can be done, aside from local computations, by invoking $\Pi_{\mathbf{Comp}}$ and $\Pi_{\mathbf{Sel}}$ for each $i \in [N]$.
6. Open all the values $\hat{\kappa}_i$ by invoking $\Pi_{\mathbf{Open}}$.
7. Intialize B_j as a list of length t_j for $j \in [k]$.
8. Set the pth element of B_j equal to $\hat{a}_i$ where $i \in [N]$ is the (unique) index with $\hat{\iota}_i = r_j$ and $\hat{\kappa}_i = p \bmod t_j$, for $j \in [k]$.

Fig. 3: Protocol $\Pi_{\mathbf{Extract\text{-}Bin}}$.

4.2 Complete Protocol Description

We now describe our protocol in complete detail, which is a concretization of the abstract protocol described in Section 3. The protocol can be found in Figures 4 and 5.

Without loss of generality, we assume that the first element of L_2 is a dummy element 0, and thus every element of L_1 is greater than the first element of L_2. This is just for ease of presentation and can easily be arranged by prepending L_2 with a dummy element before running our protocol and then stripping it off at the end of the protocol.

Correctness We will now argue the correctness of our protocol. Step 1 is the base case of the recursion which is correct from the correctness of $\Pi_{\textbf{SM-ALL}}$ or $\Pi_{\textbf{SM-Sor}}$. In Step 2, we compute the parameters k and γ as estimated in Section 3.1 and they suffice per the analyses therein. Step 3 picks random values $\delta_1, \ldots, \delta_{k+1}$ that will be used to define the labels for the recursive sub-problems that we will create. Note that the k blocks of L_1 may partition L_2 into $k+1$ chunks, k of which will each merge with one of the blocks of L_1, and the last of which is a trailing chunk that need just be appended at the end. Looking ahead, the k sub-problems will be labeled as $\overline{r}_1, \ldots, \overline{r}_k$, where $\overline{r}_j = \sum_{p\in[j]} \delta_p$ for $j \in [k]$. The trailing chunk will be labeled $\overline{r}_{k+1} = \sum_{p\in[k+1]} \delta_p$ (this computation happens in Step 7). In Step 3, we compute indicators $\mathsf{pos}_{j,j'}$ of where the jth median of L_1 merges with L_2 for $j \in [k]$. Specifically, if $\mathsf{pos}_{j,j'} \neq 0$, it stores the number of elements of L_2 that are smaller than the jth median of L_1. Simultaneously and similarly, we also compute $\mathsf{tag}_{j,j'}$ which when non-zero is assigned a random value δ_{j+1}.

Step 4 is where we aggregate the indicators to compute pos_j which is the number of elements of L_2 that are smaller than the jth median of L_1. Once we have this, we can compute the list $\overline{T}'$ chunk sizes csize_j of L_2 by looking at the difference sequence of pos_j for $j \in [k]$. The size of the trailing chunk is computed by csize_{k+1}. Step 6 then pads the chunk sizes for obliviousness. Details of how to do this can be found in Section 3.2. Step 7 computes the labels $\overline{T}$ for each sub-problem as described before, but also opens and reveals the label $\overline{r}_{k+1}$ of the trailing chunk which does not participate in any sub-problem. Step 8 computes the starting indices $\overline{C}$ of each of the chunks, zeroing them out if the chunk is empty (this is in preparation for invoking $\Pi_{\textbf{Extract-Bin}}$).

One crucial aspect that creates some additional work in our protocol is the fact that even after padding, our chunks may be of unequal sizes (this is to avoid "overpadding" with dummies, particularly later in the recursion, cf. Section 3.2). This means that we cannot reveal padded chunk sizes in correspondence with the order in which the chunks appear, but rather just the global set of chunk sizes. For this reason, we will permute the chunks, i.e., the sub-problems, and unpermute them once we recursively solve the sub-problems. For this, in Step 9, we divide L_1 into the blocks, one for each of the k sub-problems. In Step 10, we finally permute all the parameters needed for setting up our sub-problems, and hold on to the handle handle so that we can unpermute the sub-problems at the end.

We now have to mark L_2 so that we can divide it into its padded chunks by invoking $\Pi_{\textbf{Extract-Bin}}$. This is done in Steps 11-12. We basically use the $\mathsf{tag}_{j,j'}$ indicators from before and compute a prefix-sum to propagate the labels across the entire list. This is also why the labels end up being prefix sums of the δs and not the δs themselves. The reason we do things this way is because the linear operations can be done locally without any communication. Finally, in Step 13, we invoke $\Pi_{\textbf{Extract-Bin}}$ to obtain the chunks of L_2, pair them up with the blocks of L_1, and recursively solve the k sub-problems of blocks of L_1 with chunks of L_2. Once that is done, we unpermute the results of each sub-problem (after adding in the trailing chunk in the trailth place) to get the result of the merge of the lists. At this point, the dummies we have introduced while padding chunks still remain in the lists. However, we can only remove dummies at the very end of the protocol, so we don't remove them in any of the recursive calls on the sub-problems. In the end, we remove them using $\Pi_{\textbf{Extract-Ord}}$. The correctness of our protocol is immediate from the argument above and the correctness of all of the underlying protocols that we invoke.

Security The security of our protocol follows from the security of the underlying protocols that we invoke, and the fact that the values we open are input-independent. Indeed, in Step 7, the label of the trailing chunk is revealed but that will be revealed in correspondence with all the labels when we don't invoke a recursive call on a sub-problem involving it. In Step 10, we reveal the permuted list of padded chunk sizes and their labels. This is input-independent from the analysis in Section 3.2. These are the only values we reveal and all other operations are input-independent, and therefore our protocol is secure.

Efficiency Let $N = \max\{n, m\}$. Since the sizes of the sub-problems are all approximately $\frac{N}{k} = \sqrt{N \ln N}$, the recursion proceeds to a maximum depth of $\mathcal{O}(\ln\ln N)$. This holds even if we account for growth on account of padding at every stage of the recursion since our worst-case padding is smaller than $\mathcal{O}(N^c)$ for $c < 1$. Each stage of the recursion takes $\mathcal{O}(1)$ rounds and $\mathcal{O}(N)$ communication and computation. Therefore, our protocol takes $\mathcal{O}(\ln\ln N)$ rounds and $\mathcal{O}(N \ln\ln N)$ communication and computation in total.

Theorem 1. *There exists a secure merge protocol for lists of size N that takes $\mathcal{O}(N \ln\ln N)$ computation and communication and $\mathcal{O}(\ln\ln N)$ rounds and relies only on black-box access to the secure functionalities for open, comparison, selection, and shuffles.*

Recently, [5] have put forth a protocol that takes $\mathcal{O}(N)$ computation and communication and $\mathcal{O}(\ln\ln N)$ rounds relying on the same assumptions. While this is a great improvement asymptotically, we would be surprised if their protocol were to be concretely efficient. In particular, we believe that the complexities of their protocol $\Pi_{\mathbf{SSM}\text{-}\log\log}$ that has the same asymptotics as ours might involve some steep constants. It is interesting to note that we can compose our protocol with their main protocol by replacing $\Pi_{\mathbf{SSM}\text{-}\log\log}$ with our protocol to obtain a result similar to theirs asymptotically. Although this would only apply once again to uniform lists, it would likely be a lot more concretely efficient. Also, it would be interesting to see how the rest of their protocols could be made more concretely efficient, in general, or even when tailored to uniform lists.

While we have stated the requirement for the lists to be uniform, we note that much of the analysis centers on one crucial bound: *how far does an element have to look forward or behind in the other list to find its position in the merged list, i.e., is there some guarantee on the search window for any given element?* This in turn translates to how balanced chunks will be were we to take the divide-and-conquer approach as we do in this work. If the setting or the assumptions are strong enough to yield non-trivial bounds of this form, we can either directly use or suitably adapt our protocol to cater to lists from such a distribution as well. We thus note that our protocol thus applies to applications where lists may not be truly random, but they share certain characteristics with random lists that suffice for our protocol.

5 Going Beyond Uniform Lists

In this section, we describe how to perform a distributional secure merge of lists that are not uniformly distributed, but rather distributed according to some distribution. We begin by considering the case where the lists are identically distributed and move on to handle the case of arbitrary different distributions.

5.1 Identically Distributed Lists

We begin by noting that our protocol for uniform lists can be directly applied to identically distributed lists from an arbitrary distribution. The key idea is to leverage the so-called probability integral transform (see Lemma 5). The main idea is the following: from any random variable X, one can derive a uniformly distributed random variable Y (simply using the CDF of X). Crucially, this transformation is order-preserving! That is, if we have sorted lists, L_1 and L_2, which consists of samples from some distribution D, we can, in linear time and communication and $\mathcal{O}(1)$ rounds, create equivalent lists L'_1 and L'_2 of samples from the uniform distribution such that (1) L'_1 and L'_2 are sorted, and (2) the permutation that merges L'_1 and L'_2 also merges L_1 and L_2. This is can be done if the description of the distribution D is (1) publicly known, or (2) known via a secret-sharing, or (3) unknown as well, but the support of D is known; in this case, we would empirically estimate D using a scan of the lists themselves (the quality of the estimation would of course depend on the length of the lists in relation to the size of the support).

Input:

– Parties hold $\alpha \geq 0$ and secret shares of sorted lists L_1 and L_2 of sizes n and m respectively, where the first element of L_2 is 0.

Output:

– Parties hold secret shares of the merged list $L_1 \bigsqcup L_2$ of size $n+m$.

Protocol:

1. if n, m are small enough, invoke either $\Pi_{\mathbf{SM\text{-}ALL}}$ or $\Pi_{\mathbf{SM\text{-}Sort}}$ on L_1 and L_2 and return the result.
2. Choose k such that $k|n$ and $k \approx \sqrt{\frac{m}{\ln m}}$. Generate the shared list $L_3 = \mathcal{M}_{1,k}$ of the k medians of L_1. In the rest of the protocol, any value that is used as an index is assumed to be rounded to the nearest integer, and we avoid explicitly indicating a rounding in our description for readability. We also assume that comparisons with $\ell_{2,p}$ are omitted when $p < 1$ or $p > m$. Compute
$$\gamma = \sqrt{\frac{m}{2}((\eta+1)\ln 2 + \ln k)}$$
3. Pick shared random (from the underlying field $\mathbb{F}$) values $\delta_1, \ldots, \delta_{k+1}$. For $j \in [k]$ and $j' \in \left[-\gamma - \frac{j\alpha m}{k}, \gamma\right]$, compute the shared values $\mathsf{pos}_{j,j'}$ and $\mathsf{tag}_{j,j'}$ defined by
$$\mathsf{pos}_{j,j'} = \left(\frac{jm}{k} + j'\right) \cdot \left\{\ell_{2,\frac{jm}{k}+j'} \overset{?}{<} \ell_{3,j} \overset{?}{<} \ell_{2,\frac{jm}{k}+j'+1}\right\};$$
$$\mathsf{tag}_{j,j'} = \delta_{j+1} \cdot \left\{\ell_{2,\frac{jm}{k}+j'} \overset{?}{<} \ell_{3,j} \overset{?}{<} \ell_{2,\frac{jm}{k}+j'+1}\right\}$$
This can be done, aside from local computations, by invoking $\Pi_{\mathbf{Comp}}$ and $\Pi_{\mathbf{Sel}}$ for each $j \in [k]$ and $j' \in \left[-\gamma - \frac{\alpha m}{k}, \gamma\right]$.
4. For $j \in [k]$, compute the shared values $\mathsf{pos}_j = \sum_{j' \in \left[-\gamma - \frac{\alpha m}{k}, \gamma\right]} \mathsf{pos}_{j,j'}$.
5. Compute the shared values $\mathsf{csize}_j = \mathsf{pos}_j - \mathsf{pos}_{j-1}$ for $j \in [k]$, where $\mathsf{pos}_0 = 0$, and $\mathsf{csize}_{k+1} = m - \mathsf{pos}_k$. Let $\overline{T}'$ be the list with entries $\{\mathsf{csize}_1, \ldots, \mathsf{csize}_{k+1}\}$.
6. Compute the shared list $\overline{T}''$ using the padding scheme Pad that pads each entry of $\overline{T}'$. This can be done by invoking $\mathcal{F}_{\mathbf{MPC}}$ on each entry of $\overline{T}'$. Let the entries of $\overline{T}''$ be $\overline{t}''_1, \ldots, \overline{t}''_{k+1}$.
7. Compute the shared list $\overline{R}$ with entries $\overline{r}_j$ for $j \in [k+1]$ defined as $r_j = \sum_{p \in [j]} \delta_p$. Define $\overline{T} = \overline{T}'' \bowtie \overline{R}$. Additionally, open $\overline{r}_{k+1}$ by invoking $\Pi_{\mathbf{Open}}$.
8. Compute the shared list $\overline{C}$ with entries $\overline{c}_j$ for $j \in [k+1]$ defined as $\overline{c}_j = (\mathsf{pos}_{j-1} + 1) \cdot \{\mathsf{csize}_j \overset{?}{>} 0\}$, where $\mathsf{pos}_0 = 0$. This can be done, aside from local computations, by invoking $\Pi_{\mathbf{Comp}}$ and $\Pi_{\mathbf{Sel}}$.
9. Divide L_1 into k equally sized blocks $\overline{A}_j$ for $j \in [k]$. For the forthcoming shuffle, we think of L_1 as a list of size k where each block of L_1 is a macroelement.
10. Shuffle the entries of the lists $L_1, \overline{T}, \overline{C}, \overline{T}'$ by invoking $\Pi_{\mathbf{Shuffle}}((L_1, \overline{T}, \overline{C}, \overline{T}'))$. Let the shuffled lists be L'_1, T, C, T' and let the handle returned by $\Pi_{\mathbf{Shuffle}}$ be handle. Parse L'_1 as k blocks A_j for $j \in [k]$. Open the list T by invoking $\Pi_{\mathbf{Open}}$. Let trail be the unique $j \in [k+1]$ such that $\overline{r}_{k+1} = r_j$ where $(\cdot, r_j)$ is the jth entry in the list T.

Fig. 4: Our protocol for securely merging two random lists (Part 1 of 2).

INPUT:

– Parties hold $\alpha \geq 0$ and secret shares of sorted lists L_1 and L_2 of sizes n and m respectively, where the first element of L_2 is 0.

OUTPUT:

– Parties hold secret shares of the merged list $L_1 \bigsqcup L_2$ of size $n + m$.

PROTOCOL (CONTD.):

1. Compute the shared list Lab of length m with entries lab_i for $i \in [m]$ defined as
$$\mathsf{lab}_i = \sum_{\substack{j \in [k], j' \in [-\gamma, \gamma] \\ \frac{jm}{k} + j' = i}} \mathsf{tag}_{j,j'}$$
Update lab_m as $\mathsf{lab}_m = \delta_{k+1} \cdot \{\mathsf{lab}_m \overset{?}{=} 0\}$. This can be done, aside from local computations, by invoking $\Pi_{\mathbf{Comp}}$ and $\Pi_{\mathbf{Sel}}$.
2. Let $\mathsf{lab}_0 = 0$. Compute the shared list I with entries ι_i for $i \in [m]$ defined as
$$\iota_i = \delta_1 + \sum_{p \in [i]} \mathsf{lab}_{p-1}$$
3. Define $A = L_2 \bowtie I$. Invoke $\Pi_{\mathbf{Extract\text{-}Bin}}(T, A, C, T')$ to obtain the shared lists B_j for $j \in [k+1]$.
4. Define the k sub-problems (A_j, B_j) for $j \in [k+1] \setminus \{\mathsf{trail}\}$. Recursively invoke this protocol on each of the sub-problems with appropriate choices of the parameter α to obtain lists P_j for $j \in [k+1] \setminus \{\mathsf{trail}\}$. Set $P_{\mathsf{trail}} = B_{\mathsf{trail}}$. Define $P = \{P_1, \ldots, P_{k+1}\}$. For the forthcoming unshuffle, we think of P as a list of size $k+1$ where each P_j is a macroelement for $j \in [k+1]$.
5. Invoke $\Pi_{\mathbf{Unshuffle}}(P, \mathsf{handle})$ to unpermute P and obtain the list $\overline{P} = \{\overline{P}_1, \ldots, \overline{P}_{k+1}\}$.
6. If this was not the parent recursive call of the protocol, return $\overline{P}$. Otherwise, invoke $\Pi_{\mathbf{Extract\text{-}Ord}}(\overline{P})$ to remove dummy elements and return the final result $L_1 \bigsqcup L_2$. Note in this final invocation of $\Pi_{\mathbf{Extract\text{-}Ord}}$ that we think of $\overline{P}$ not as a list of size $k+1$, but where each element of each $\overline{P}_j$ is seen as a separate element, for $j \in [k+1]$.

Fig. 5: Our protocol for securely merging two random lists (Part 2 of 2).

5.2 Arbitrarily Distributed Lists

The case where the lists are distributed according to arbitrarily different distributions is a lot more challenging. Recall that the reason why we are able to merge uniform lists well is because elements close to one another in the support of the distribution must be close to one another in the lists as well. In case of arbitrarily but identically distributed lists, the probability integral transform essentially stretches and squishes the distribution to provide the guarantee that elements close to one another in the two lists must be close to one another in their positions in the lists as well.

A first idea that does not work is the following. Suppose the lists L_1 and L_2 of lengths n and m are distributed according to the distributions D_1 and D_2 respectively. Consider the convex combination of their distributions given by $D = \frac{nD_1+mD_2}{n+m}$. In the limit, drawing $n+m$ samples from D is the same as drawing n samples from D_1 and m samples from D_2. So, one consider whether we could think of L_1 and L_2 as coming from the distribution D and reduce to the case of identically distributed lists. This is however not true. While in the limit, the $n+m$ elements of L_1 and L_2 do in fact come from D, as individual lists, they come from D_1 and D_2 respectively. We could perhaps try to salvage this by trying to introduce approximately dummy elements from D_2 into L_1 and D_1 into L_2 so that their distributions match that of D, but the lists are no longer sorted!

In a sense, this is not surprising as if the lists could be arbitrarily distributed, we could land up with a “worst-case” merge, i.e., the bound β on the size of the “window” that we would have to consider could be quite large in general. Thus, in general, our approach could get as inefficient as performing a merge on worst-case lists. Nonetheless, we can still use the the fact that our lists are distributed according to some distributions to design a protocol that would be much better in scenarios where the distributions are not too dissimilar.

We take a second look at the convex combination D of the distributions. Effectively, we will make use of the probability integral transform to bucketize the lists into “uniform” buckets. Essentially, we examine D and determine thresholds that split D into several quantiles. Once we have obtain these thresholds, we use them to divide the lists L_1 and L_2 into buckets. To guarantee security, we can pad the buckets. If the descriptions of D_1 and D_2 are publicly known, we can use them to come up with padding thresholds distributionally as well like we did in Section 3.2. Note that these buckets could be quite different in size. However, since they span equally-sized quantiles of D, considering each bucket as a single element, they are uniformly distributed over D. Now, thinking of the buckets as elements, we can run over protocol for uniform lists which determines the buckets in one list that the elements in a bucket of the list must consider for their merge. There is a lot of room to tweak concrete performance. One can consider varying the number of quantiles. The smaller the number of quantiles, the smaller the window-size one needs to consider in terms of the number of buckets, but the worse the merging of buckets. The larger the number of quantiles, the larger the window-size one needs to consider in terms of the number of buckets, but the better the merging of buckets. One can also play with how the buckets are merged. We could use a trivial protocol such as all-pair comparisons, or a sort-based merge, or any of the other close-to linear time merge protocols, depending on the bucket sizes considered.

6 Evaluation

We implement our protocol for uniform lists and compare it to the state of art radix sorting protocol of [8] in the honest majority three-party semi-honest setting. The primary reason for choosing this setting is to enable us to compare with a high-performance sorting protocol. We note that our protocol could be implemented in other settings that have the required building blocks, e.g. two party. For our protocol, we make use of two levels of recursion regardless of the size of the input lists. For the padding scheme we ensure that there is statistically negligible leakage with parameter $\sigma = 40$. We implement the base case using the "shuffle-then-sort" paradigm [14,13]. We implemented both protocols using C++. We performed benchmarks on a single 8-core i7 laptop communicating over TCP via localhost. Network latency was sub-millisecond. We note that this low network latency disproportionally benefits [8] due to having a larger round complexity. Each party was given a single computation thread to run on. Input elements being sorted are 32 bits long.

Protocol	$m = n$	Time (ms)	Communication (MB)
[ours]	2^{12}	101	2.9
[ours]	2^{16}	251	47.3
[ours]	2^{20}	3,166	779.6
[8]	2^{12}	116	23.0
[8]	2^{16}	1,220	367.1
[8]	2^{20}	27,123	5,887.5

Table 4: Comparison of performance overheads (running time and total communication) of our secure merge protocol and generic radix sorting protocol [8]. Input elements being sorted are 32 bits long. Each of the two lists has n items, i.e., [8] is sorting $2n$ items.

Table 4 contains the performance results. For the largest input size tested of $n = 2^{20}$, we observe that our protocol is an order of magnitude faster, requiring just 3.2 seconds compared to 27 seconds to perform generic sorting [8]. Similarly, our protocol requires just 780MB of communication compared to 5,888MB for [8]. As expected, the main overhead of our protocol is the secure comparisons and oblivious shuffles. For smaller input lists we observe that the difference between our protocols decreases, with our protocol for $n = 2^{16}$ being $4.8\times$ faster and our protocol for $n = 2^{12}$ being just $1.2\times$ faster. However, we note that in a network with realistic latency, our protocol would likely outperform [8] by a larger margin.

References

1. Sums of independent poisson random variables are poisson random variables. https://llc.stat.purdue.edu/2014/41600/notes/prob1805.pdf (2014)
2. Tail bounds for poisson random variables. http://www.cs.columbia.edu/~ccanonne/files/misc/2017-poissonconcentration.pdf (2017)
3. Mcdiarmid's inequality. https://www.cs.columbia.edu/~djhsu/coms4995-s20/lectures/mcdiarmid-notes.pdf (2020)
4. Badrinarayanan, S., Das, S., Garimella, G., Raghuraman, S., Rindal, P.: Secret-shared joins with multiplicity from aggregation trees. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. pp. 209–222. ACM (2022)
5. Blunk, M., Bunn, P., Dittmer, S., Lu, S., Ostrovsky, R.: Secure merge in linear time and o(log log N) rounds. IACR Cryptol. ePrint Arch. p. 590 (2022)
6. Buddhavarapu, P., Knox, A., Mohassel, P., Sengupta, S., Taubeneck, E., Vlaskin, V.: Private matching for compute. IACR Cryptol. ePrint Arch. **2020**, 599 (2020)
7. Chan, T.H., Katz, J., Nayak, K., Polychroniadou, A., Shi, E.: More is less: Perfectly secure oblivious algorithms in the multi-server setting. In: Peyrin, T., Galbraith, S.D. (eds.) Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11274, pp. 158–188. Springer (2018)
8. Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Kiribuchi, N., Pinkas, B.: An efficient secure three-party sorting protocol with an honest majority. IACR Cryptol. ePrint Arch. p. 695 (2019)
9. Cristofaro, E.D., Tsudik, G.: Practical private set intersection protocols with linear complexity. In: Financial Cryptography. Lecture Notes in Computer Science, vol. 6052, pp. 143–159. Springer (2010)
10. Falk, B.H., Nema, R., Ostrovsky, R.: A linear-time 2-party secure merge protocol. In: Dolev, S., Katz, J., Meisels, A. (eds.) Cyber Security, Cryptology, and Machine Learning - 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30 - July 1, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13301, pp. 408–427. Springer (2022)
11. Falk, B.H., Ostrovsky, R.: Secure merge with o(n log log n) secure operations. In: Tessaro, S. (ed.) 2nd Conference on Information-Theoretic Cryptography, ITC 2021, July 23-26, 2021, Virtual Conference. LIPIcs, vol. 199, pp. 7:1–7:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
12. Garimella, G., Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: Oblivious key-value stores and amplification for private set intersection. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12826, pp. 395–425. Springer (2021)
13. Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. IACR Cryptol. ePrint Arch. p. 121 (2014)
14. Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: Kwon, T., Lee, M., Kwon, D. (eds.) Information Security and Cryptology - ICISC 2012

- 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7839, pp. 202–216. Springer (2012)

15. Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012. The Internet Society (2012), `https://www.ndss-symposium.org/ndss2012/private-set-intersection-are-garbled-circuits-better-custom-protocols`
16. Ion, M., Kreuter, B., Nergiz, A.E., Patel, S., Saxena, S., Seth, K., Raykova, M., Shanahan, D., Yung, M.: On deploying secure computing: Private intersection-sum-with-cardinality. In: EuroS&P. pp. 370–389. IEEE (2020)
17. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. IACR Cryptol. ePrint Arch. p. 799 (2016), `http://eprint.iacr.org/2016/799`
18. Kübler, R.: The ‘balls into bins’ process and its poisson approximation. `https://www.cantorsparadise.com/the-balls-into-bins-process-and-its-poisson-approximation-e38d11bdf283` (2021)
19. jie Lu, W., Huang, Z., Zhang, Q., Wang, Y., Hong, C.: Squirrel: A scalable secure two-party computation framework for training gradient boosting decision tree. Cryptology ePrint Archive, Paper 2023/527 (2023), `https://eprint.iacr.org/2023/527`, `https://eprint.iacr.org/2023/527`
20. Lucani, D.E., Nielsen, L., Orlandi, C., Pagnin, E., Vestergaard, R.: Secure generalized deduplication via multi-key revealing encryption. In: Galdi, C., Kolesnikov, V. (eds.) Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12238, pp. 298–318. Springer (2020)
21. Meadows, C.A.: A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In: IEEE Symposium on Security and Privacy. pp. 134–137. IEEE Computer Society (1986)
22. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press (2005)
23. Mohassel, P., Rindal, P.: Aby3: A mixed protocol framework for machine learning. Cryptology ePrint Archive, Paper 2018/403 (2018), `https://eprint.iacr.org/2018/403`, `https://eprint.iacr.org/2018/403`
24. Mohassel, P., Rindal, P., Rosulek, M.: Fast database joins and PSI for secret shared data. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020. pp. 1271–1287. ACM (2020)
25. Nayak, K., Wang, X.S., Ioannidis, S., Weinsberg, U., Taft, N., Shi, E.: Graphsc: Parallel secure computation made easy. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 377–394. IEEE Computer Society (2015)
26. Orrù, M., Orsini, E., Scholl, P.: Actively secure 1-out-of-n OT extension with application to private set intersection. In: CT-RSA. Lecture Notes in Computer Science, vol. 10159, pp. 381–396. Springer (2017)
27. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: Spot-light: Lightweight private set intersection from sparse OT extension. In: CRYPTO (3). Lecture Notes in Computer Science, vol. 11694, pp. 401–431. Springer (2019)

28. Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: Private set intersection using permutation-based hashing. In: USENIX Security Symposium. pp. 515–530. USENIX Association (2015)
29. Pinkas, B., Schneider, T., Zohner, M.: Faster private set intersection based on OT extension. In: USENIX Security Symposium. pp. 797–812. USENIX Association (2014)
30. Rindal, P., Rosulek, M.: Malicious-secure private set intersection via dual execution. In: ACM Conference on Computer and Communications Security. pp. 1229–1242. ACM (2017)
31. Rindal, P., Schoppmann, P.: VOLE-PSI: fast OPRF and circuit-psi from vector-ole. In: Canteaut, A., Standaert, F. (eds.) Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12697, pp. 901–930. Springer (2021)