Logstar: Efficient Linear* Time Secure Merge

Suvradip Chakraborty[1], Stanislav Peceny[2]**, Srinivasan Raghuraman[3], and
Peter Rindal[1]

[1] Visa Research
[2] Georgia Institute of Technology
[3] Visa Research and MIT

Abstract. Secure merge considers the problem of combining two sorted lists into a single sorted secret-shared list. Merge is a fundamental building block for many real-world applications. For example, secure merge can implement a large number of SQL-like database joins, which are essential for almost any data processing task such as privacy-preserving fraud detection, ad conversion rates, data deduplication, and many more.

We present two constructions with communication bandwidth and rounds tradeoff. Logstar, our bandwidth-optimized construction, takes inspiration from Falk and Ostrovsky (ITC, 2021) and runs in $O(n \log^* n)$ time and communication with $O(\log n)$ rounds. In particular, for all conceivable n, the $\log^* n$ factor will be equal to the constant 2, and therefore we achieve a near-linear running time. Median, our rounds-optimized construction, builds on the classic parallel medians-based insecure merge approach of Valiant (SIAM J. Comput., 1975), later explored in the secure setting by Blunk et al. (2022), and requires $O(n \log^c n)$, $c \approx 1.71$, communication with $O(\log \log n)$ rounds.

We introduce two additional constructions that merge input lists of different sizes. SquareRootMerge merges lists of sizes $n^{\frac{1}{2}}$ and n and runs in $O(n)$ time and communication with $O(\log n)$ rounds. CubeRootMerge is closely inspired by Blunk et al.'s (2022) construction and merges lists of sizes $n^{\frac{1}{3}}$ and n. It runs in $O(n)$ time and communication with $O(1)$ rounds.

We optimize our constructions for *concrete efficiency*. Today, concretely efficient secure merge protocols rely on standard techniques such as Batcher's merging network or generic sorting. These approaches require an $O(n \log n)$ size circuit of $O(\log n)$ depth. Despite significant research thrust, no work has been able to reduce their concrete costs. Our constructions are the first to be more efficient by improving their asymptotics and maintaining small constants. We analytically benchmark against these constructions and show that Logstar reduces bandwidth costs $\approx 2.0\times$ and Median reduces rounds $\approx 1.5\times$.

* Almost linear.

** Part of this work was done while the author was an intern at Visa Research.

1 Introduction

Secure Multi-Party Computation (MPC) is an area of cryptography that enables parties to compute on private data without revealing it to counterparties. Traditionally, MPC techniques first compile functions into Boolean or arithmetic circuits and then evaluate them gate by gate. The advantage of these generic MPC techniques is that they can evaluate *arbitrary* functions. Much research effort has been put into optimizing them. For example, [KS08,ZRE15] reduce the costs of individual circuit gates; [HKP20,HKP21] reduce costs in circuits with complex control flow. [YPHK23] recently introduced a novel approach to generic MPC, which escapes the traditional circuit model, and further improves on programs with complex control flow. Despite these significant improvements, the techniques remain cost-prohibitive for many applications.

Special-purpose MPC techniques address this trade-off by focusing on efficient evaluation of *specific* functions. Many works, e.g. [BCG+18,APR+24], evaluate machine learning functions but also other functions such as secure sorting [AHI+22]. These technique are tailored to a specific functionality or building block, but in turn are practically efficient.

In this work, we consider a secure merge problem, where two sorted and secret-shared lists are combined to produce a sorted output. Secure merge has found many applications spanning database operations, joins, and more. We include a necessarily non-exhaustive list in Section 1.1. More efficient secure merge will lead to improvement in all such applications. Moreover, we believe that an efficient merge protocol will open up many new MPC-specific applications that were previously prohibitively expensive. Going even further, we optimize the overall cost of the merge by proposing specialized protocols for various network settings, offering tradeoffs between bandwidth and latency.

Our Setting. Our protocols implement the functionality $\mathcal{F}_{\mathsf{merge}}(X, Y)$ (see Figure 1), which takes as input two sorted secret-shared lists X and Y. The output is a secret-shared permutation π such that applying π to the concatenated lists $\pi(X||Y)$ results in the sorted list $X \bigsqcup Y$. It is worth noting that a protocol for non-secret-shared input lists can be easily obtained by first secret-sharing X, Y. We typically think of our protocols as being for 2 parties, but it naturally supports any number of parties. Our contribution can be viewed as a circuit for secure merge, whose gates can be traditional operations like addition, multiplication, but also more complex operations such as permutations, extraction, etc. Given n-party protocols for these gates, our techniques can be implemented in the n-party setting.

$\mathcal{F}_{\mathsf{merge}}$ **Functionality**

INPUT: Two sorted secret-shared lists $[\![X]\!]$ and $[\![Y]\!]$.
OUTPUT: Secret-shared permutation $[\![\pi]\!]$ such that $\pi(X||Y)$ forms a sorted list $X \bigsqcup Y$.

Fig. 1: $\mathcal{F}_{\mathsf{merge}}$ is the functionality that our protocols implement.

Our goal is to optimize for concrete efficiency. Today, concretely efficient secure merge techniques use generic MPC primitives. For example, evaluating Batcher's merging network [Bat68] with GMW incurs $O(n \log n)$ time and communication with $O(\log n)$ rounds. Another technique relies on sorting. The idea of the state-of-the-art shuffle-then-sort paradigm [HKI+13] is that if the data is shuffled, then standard sorting algorithms can reveal the result of each secure comparison and move data based on the result without compromising security. This paradigm incurs the same asymptotics as evaluating Batcher's network with GMW. Alternative efficient sorting techniques that do not rely on the initial shuffle are based on radix sort [HICT14,CHI+19]. However, they outperform shuffle-then-sort only for some parameter regimes. While these techniques have small constants, their runtime complexity is unsatisfactory as (insecure) merging is an easier problem than (insecure) sorting. It is well-known that sorting a list of length n in plaintext requires $O(n \log n)$ comparisons, while merging two already sorted lists requires only $O(n)$. I.e., as [BBD+22] noted, by taking advantage of the ordering on the two lists, merge can outperform sorting by a $O(\log n)$ factor. Thus, our work hopes to find a secure merge that is not limited by the asymptotics of the generic techniques, but retains their small constants.

Concretely, the most efficient secure merge prior to our work is attained by evaluating Batcher's network. Despite many works on secure merge, e.g. [FO21,FNO22,BBD+22], none have managed to concretely outperform it. They have intriguing runtime asymptotics, but they are complex and incur large constants [BBD+22] or have (close to) linear round complexity [FO21,FNO22]. Linear round complexity plainly precludes adoption for all but very small lists. [BBD+22] does not discuss concrete performance; we estimate their cost in Section 8 and show that their merge is concretely slower than existing techniques. Our main merging protocol reduces their bandwidth/rounds by $\approx 8.3\times/\approx 3.1\times$, respectively.

We introduce two symmetric constructions ($|X| = |Y| = n$) with tradeoff between communication bandwidth and communication rounds. Both have better asymptotics than the generic techniques and small constants. Our first bandwidth optimized construction Logstar has almost linear bandwidth $O(n \log^* n)$ with $O(\log n)$ rounds. Note that $\log^*$ is a small constant for all feasible list lengths n (e.g. for $n = 2^{65536}$, $\log^* n = 5$). Our second rounds optimized construction Median incurs $O(n \log^c n)$, $c \approx 1.71$, bandwidth but uses only $O(\log \log n)$ rounds. In both constructions, communication bandwidth and computation have the same asymptotics.

Along the way, we modify and present in our notation [BBD+22]'s asymmetric merge ($|X| \neq |Y|$), which we call CubeRootMerge. CubeRootMerge is a subprotocol of Median and merges lists of length $n^{\frac{1}{3}}$ and n. It runs in $O(n)$ time and communication and uses $O(1)$ rounds. Separately, we design and present SquareRootMerge, which merges lists of length $n^{\frac{1}{2}}$ and n. It runs in $O(n)$ times and communication and uses $O(\log n)$ rounds.

In Section 1.1, we provide the motivation for secure merge, and in Section 1.2, we summarize our contributions.

1.1 Applications of Secure Merge

Secure Sort. Secure sort reduces to secure merge when each party holds a private list (i.e. the list is not secret-shared). Each party separately invokes an insecure local sort algorithm on their list so that the more costly secure interactive operations are required only for merging the lists. Note that merge is an easier problem than sort. Intuitively, this is because in merge the input lists are already sorted. In contrast, sort cannot make any assumptions on the input. In this paper, we show that our merge is more efficient than existing secure merge/sort protocols.

Database Joins. We can use our merge to reduce the costs of secure SQL-like database joins. State-of-the-art constructions [BDG$^+$22,AHK$^+$23] rely on secure sort although the inputs are sorted. For many applications, it is possible to replace sort with our merge for better efficiency.

Database Queries. In some applications, e.g. to query order statistics, databases need to be ordered. When inserting new entries, it is more efficient to securely merge them in with our protocols and maintain the database sorted rather than execute secure sort.

GROUP BY Statement. Recall the SQL GROUP BY statement groups database rows with the same values into summary rows. E.g., it can answer queries such as 'Find the number of clients by country'. It is often used in conjunction with aggregate functions such as COUNT, MAX, AVG to answer useful questions about the database. We can clearly use sort to order the database based on the row values. We then follow up with MPC to compute the aggregate functions. With our merge, we can replace the initial sort to reduce costs.

Decision Trees. Merge is a necessary tool to construct decision trees. Parties first sort their datasets. Once their datasets are sorted, they retrieve k medians, which determine the predicate to use at each decision node. To improve performance, we can use our protocol instead of sort to merge their locally sorted datasets.

1.2 Our Contributions

We design two highly efficient secure merge constructions:

- Logstar**: Our Bandwidth-optimized Construction.** Our first construction uses $O(n \log^* n)$ communication and computation with $O(\log n)$ rounds. I.e., we reduce the *bandwidth and work* of the state-of-the-art concretely efficient approaches from $O(n \log n)$ to almost linear $O(n \log^* n)$. This construction relies on some key ideas of [FO21] and to efficiently implement uses [BDG$^+$22]'s so called aggregation trees.
- Median**: Our Rounds-optimized Construction.** Our second construction uses $O(n \log^c n)$, $c \approx 1.71$, communication and computation with $O(\log \log n)$ rounds. I.e., we reduce the *rounds* of the state-of-the-art concretely efficient approaches from $O(\log n)$ to $O(\log \log n)$. This construction relies on a

medians-based approach first introduced by Valiant [Val75] in the insecure setting (see Section 3.2) and then explored in [BBD+22] in the secure setting (see Section 3.3). In our construction, we importantly rely on [BBD+22]'s block alignment lemma (see Section 6.3) and use some of their subprotocols.

Along the way, we present CubeRootMerge, which merges two lists of sizes $n^{\frac{1}{3}}$ and n. This protocol is closely inspired by [BBD+22]'s protocol, but is modified and expressed in our notation. As the original protocol, it runs in $O(n)$ time and communications and $O(1)$ rounds.

Additionally, we design SquareRootMerge, which merges two lists of length $n^{\frac{1}{2}}$ and n, respectively. We believe this technique to be of independent interest. It runs in $O(n)$ time and communication and $O(\log n)$ rounds. This approach also relies on [Val75,BBD+22]'s medians-based technique and uses [BDG+22]'s aggregation trees.

Note that all our protocols improve the asymptotics of the state-of-the-art concretely efficient approaches while maintaining their small constants, resulting in total concrete improvement. Also, our protocols work for any size inputs. The main concern is how the input size impacts the asymptotic running times. They simply get the best asymptotics for these sizes.

We analytically benchmark our protocols against the state-of-the-art Batcher's network merge protocol (see Section 2). For $n = 2^{20}$ and $\ell = 128$-bit list elements, we estimate Logstar reduces bandwidth $\approx 1.4\times$ without significantly increasing the number of rounds (i.e., Π-Logstar uses 155 rounds vs. state of the art uses 154). Note that for a larger input size of $n = 2^{30}$, the improvement in bandwidth increases to $\approx 2.0\times$, while the number of rounds increases to less than $1.1\times$ compared to the state of the art. Median introduces a tradeoff between bandwidth and rounds; for $n = 2^{20}$ and $\ell = 128$-bit list elements, it reduces the number of rounds $\approx 1.3\times$ (and $\approx 1.5\times$ for $n = 2^{30}$) at the cost of increasing bandwidth $\approx 2.5\times$ (and $\approx 2.2\times$ for $n = 2^{30}$). Hence, Median is useful on networks with high bandwidth but constrained latency. For $n = 2^{20}$ and $\ell = 128$, SquareRootMerge reduces bandwidth $\approx 2.5\times$ and rounds $\approx 2.6\times$; CubeRootMerge reduces bandwidth $\approx 5.0\times$ and rounds $\approx 4.3\times$. The $(n^{\frac{1}{3}}, n)$-merge of [BBD+22] has similar performance, but they do not give concrete estimates. We do not consider the performance of CubeRootMerge as our improvement.

2 Related Work

We review related work, focusing on works that optimize both insecure and secure merge. Secure merge can be viewed as a special case of secure sort when the input lists are already sorted. I.e., any sort is also a merge protocol. For that reason, we first review works that focus on sort before getting into merge. When reviewing secure sort, we keep in mind that even in plaintext any comparison-based sorting protocol requires $O(n \log n)$ comparisons, whereas merge needs only $O(n)$ comparisons. Hence, sort is a harder problem than merge in the insecure setting. In the secure setting, concretely efficient merge protocols today evaluate Batcher's network or run a secure sort. Because of our contributions that get much closer to the plaintext merge costs, secure sort protocols are no longer competitive with secure merge.

2.1 Secure Sort

A common way to get secure sort is to implement a sorting network with a generic MPC protocol such as GMW. Asymptotically, the fastest network is the AKS network [AKS83] requiring $O(n \log n)$ comparisons. While asymptotically optimal, the network is practically prohibitive as the hidden constants are enormous. In contrast, Batcher's sorting network [Bat68] requires $n \log^2 n$ comparisons but is practically efficient.

The reason why sorting networks are often used over other traditional sorts such as mergesort and quicksort is that the data movement in these sorts is input-dependent, and hence not oblivious. [HKI+13] introduced a *shuffle-then-sort* paradigm, which observes that many traditional sorts can be made oblivious by first securely shuffling the inputs. I.e., after the shuffle, it is secure to sort with a traditional $O(n \log n)$ sorting algorithm. One must still compute each comparison of the sort under MPC, but the result of the comparison can be revealed. The parties then reorder the data corresponding to the comparison inputs based on the comparison output. As secure shuffle can be implemented in $O(n)$ [PRRS24], the entire sort takes $O(n \log n)$.

Alternative secure sorts include radix sort [HICT14,CHI+19], which outperforms shuffle-then-sort for some combination of list length n and the bitlength ℓ. Zig-zag sort [Goo14] runs in $O(n \log n)$ with small constants, but its depth is $O(n \log n)$. Randomized shellsort [Goo10] also runs in $O(n \log n)$ with small constants, but is only correct with high probability.

2.2 Secure Merge

We now present works that explicitly solve secure merge. We start with techniques that use generic MPC to implement secure merge, and follow up with techniques based on the shuffle-then-sort paradigm (i.e. shuffle-then-merge). Then we discuss works that stress asymptotic guarantees. In Figure 2 we compare our asymptotic performance with state-of-the-art works.

Secure Merge via Generic MPC [BBD+22]. We can pick any merging algorithm representable as a Boolean circuit and evaluate it with a garbled circuit (GC) or GMW. It is well-known that such circuit will have size at least $O(n \log n)$ and depth $O(\log n)$. For example, we can use Batcher's merging network to obtain such circuit. By using GMW to evaluate this circuit, we will incur $O(n \log n)$ communication/computation and $O(\log\ n)$ rounds. We consider this the most performant merge technique prior to our work. If we use GC instead, we will

Algorithm	# Comparison	Depth	Constants	
			# Cmp.	Depth
[BBD+22]'s Sub*	$O(n \log\log n)$	$O(\log\log n)$	25.5	19.2
[BBD+22]'s Full◇	$O(n)$	$O(\log\log n)$	120.8	55.3
(Shuffled) Quick-Sort	$O(n \log n)$	$O(\log n)$	1.4	1.4
Batcher's Network	$O(n \log n)$	$O(\log n)$	1.0	1.0
Logstar	$O(n \log^* n)$	$O(\log n)$	2.9	1.7
Median	$O(n \log^c n)$, $c \approx 1.71$	$O(\log\log n)$	0.3	3.9

Fig. 2: This table compares the asymptotics of our techniques Logstar and Median with state-of-the-art approaches. Depth refers to the number of sequential comparisons. The Constants column estimates the size of the constants hidden by the asymptotics for $n = 2^{20}$ and $c = \log_{3/2}(2) \approx 1.71$. * [BBD+22]'s symmetric merge subprotocol $\mathit{\Pi}_{\mathsf{SSM}\text{-}\mathrm{loglog}n}$ (Step 1 in Section 3.3). ◇ [BBD+22]'s full symmetric merge protocol (Step 3 in Section 3.3).

get $O(1)$ rounds but will incur computational security parameter κ blowup in communication and computation, i.e. $O(\kappa n \log n)$.

Note that we can also get a $O(1)$ round protocol by using fully homomorphic encryption (FHE). In this case, the communication is proportional to the size of the encryptions of one party's list, i.e. $O(n)$. As the computation must remain input-independent, however, one of the parties will need to execute a circuit under FHE with $O(n \log n)$ comparisons. As the circuit has depth $O(\log\ n)$, this approach further requires bootstrapping, and hence is practically expensive.

Secure Merge via Shuffle-then-merge. While the shuffle-then-sort approach was originally designed for secure sort, similar techniques have been developed for secure merge [CKN+18,FNO22]. We refer to them as shuffle-then-merge. In this setting, the approach is more subtle as the input lists are presorted and the merge needs to process them in sorted order (otherwise this technique reduces to secure sort).

In these techniques, the parties first construct a special linked list structure for each input list and then shuffle the linked lists. The parties then essentially run a plaintext merge sort algorithm. They maintain a secret-shared version of the head of the two linked lists. At each step, the smaller head is placed into the merged list and the index of its shuffled child is revealed. The parties then update the head with its child and the process is repeated.

While these techniques run in only $O(n)$ time and communication, the plaintext merge emulation is sequential, and thus takes $O(n)$ rounds. This is plainly prohibitive for most applications. As a result, standard shuffle-then-sort is in most settings more practical than shuffle-then-merge.

Secure Merge with Strong Asymptotics. We now discuss secure merge protocols that emphasize *asymptotic* improvements [FO21,FNO22,BBD+22]. [FO21] introduced a protocol that runs in $O(n \log\log n)$ time and communication and almost linear rounds. Such round complexity is prohibitive in most settings. Our Logstar uses some of their ideas but is significantly different and requires only $O(\log n)$ rounds (see Section 4.1). [FNO22] gives a $O(n)$ time and communication protocol, but also requires $O(n)$ rounds. Recently, [BBD+22] introduced a protocol that runs in $O(n)$ time and communication and $O(\log\log n)$ rounds. This approach, like Median, is based on [Val75]'s medians-based approach (see Section 3.2) and mixes several merge protocols with different properties such that they get their desired asymptotics. While asymptotically intriguing, the protocol is complex and has high constants. As discussed in Section 1, we show that our Logstar reduces their bandwidth/rounds by $\approx 8.3\times/\approx 3.1\times$, respectively. We note that one of [BBD+22]'s subprotocols that implements their full linear time merge, can be viewed as a standalone merge of arbitrary length lists. This protocol corresponds to Step 1 in our high level description of [BBD+22] (see Section 3.3) and runs in $O(n \log\log n)$ time with $O(\log\log n)$ rounds. Concretely, our Logstar reduces bandwidth over this subprotocol by $\approx 7.5\times$ and rounds by $\approx 1.2\times$. See Section 8 for a more detailed comparison.

3 Preliminaries

3.1 Notation and Assumptions

- We use 0-based indexing.
- $[n]$ denotes the sequence of integers $0, \ldots, n-1$. $[n, l]$ denotes $n, \ldots, l-1$.
- We denote lists as $X = X_0, X_1, \ldots, X_{n-1}$.
- n denotes list length. Sometimes we express list length as a function of n, e.g. $n^{\frac{1}{2}}$.
- We index lists with subscripts. E.g., X_0 is the first entry of X.
- We denote sublists as $X_{[a,b]} = X_a, \ldots, X_{b-1}$.
- We denote merge with $\bigsqcup$. E.g., $X \bigsqcup Y$ is the result of merging X and Y, where $|X \bigsqcup Y| = |X| + |Y|$.
- We concatenate two lists with $||$, e.g., $X||Y$.
- We associate variables with list elements with a . followed by the variable name. E.g., X_i.IsReal denotes if X_i is a real or a dummy element.
- We denote the list of k medians of X as $X'_i = X_{(i+1)\frac{n}{k}-1}$, $\forall i \in [k]$.
- We negate a bit b with $\neg$, e.g., $\neg b$.
- We work with additive secret shares. We use the shorthand $[\![X]\!]$ to denote a (uniform) sharing of array X. We mostly work with binary secret shares, but sometimes we require arithmetic secret shares for prefix sum. In those cases, we implicitly convert between the shares using standard techniques [DSZ15,MR18].
- We denote a protocol that implements functionality Func as Π-Func. E.g., Π-Sort implements Sort (securely sorts two secret-shared lists).

Throughout this paper, we treat the length of the list elements as *constant*. This is reflected in our asymptotic cost computations. Note that this approach was taken by previous merge papers [FO21,FNO22,BBD+22].

3.2 [Val75]'s Insecure Merge

Insecure merge is well-researched. The one most pertinent to our techniques for secure merge is [Val75]'s *medians-based approach*. [BBD+22] was the first to explore the application of this approach in the secure setting, see Section 3.3 for a more detailed description. [Val75]'s work is designed to solve merge on multiprocessor machines. As it is so closely related to our approach for secure merge, we review it below.

Let X and Y be the input lists such that $|X| = |Y| = n$. The merge works as follows:

1. Select $k = n^{\frac{1}{2}}$ medians X' of X. Repeat for Y. The medians X' and Y' split the lists into same-size blocks.
2. Compare all X'_i with all Y'_i. This requires n comparisons and tells us into which block of Y each median X'_i needs to be inserted.
3. Now we compare each X'_i with all elements in the block of Y into which it needs to be inserted. This also requires n comparisons. At this point, we have identified where each X'_i goes in Y. This effectively splits the merge into $n^{\frac{1}{2}}$ merge subproblems. The first input is a block from X of size $n^{\frac{1}{2}}$; the latter is a chunk from Y of variable size. Both the block of X and the chunk of Y have values between two consecutive medians X'_i and X'_{i+1}.
4. We now recursively merge the subproblems.

3.3 [BBD+22]'s Secure Merge

[BBD+22] puts forth a rather fascinating set of protocols that manage to collectively achieve a secure symmetric ($|X| = |Y| = n$) merge with $O(n)$ work and in $O(\log \log n)$ rounds (among other things). Asymptotically, this is probably the best that one can hope for, as discussed in [BBD+22]. For completeness, we include a high-level description of how their protocols work.

Step 0: Asymmetric $(n^{\frac{1}{3}}, n)$*-merge with* $O(n)$ *work and in* $O(1)$ *rounds.* This is the protocol described in Figures 8 and 9 and Sections 6.3, 7.1., and 7.2 of [BBD+22]. In this work, we consider a modified version of this protocol, which we call Π-CubeRootMerge. At a high level, the protocol proceeds as follows. Divide the list Y of length n into $n^{\frac{2}{3}}$ blocks of size $n^{\frac{1}{3}}$. Then, by comparing every pair of elements from X of length $n^{\frac{1}{3}}$ with the $n^{\frac{2}{3}}$ *medians* of Y (i.e., the last element of each block), we can determine which blocks of Y merge with elements of X in $O(n)$ work and in $O(1)$ rounds. Then, we can obliviously extract all the blocks of Y that merge with elements of X in $O(n)$ work and in $O(1)$ rounds. Note that there can only be $n^{\frac{1}{3}}$ of them as $|X| = n^{\frac{1}{3}}$ resulting in a total of at most $n^{\frac{1}{3}} \cdot n^{\frac{1}{3}} = n^{\frac{2}{3}}$ elements from Y as each block of Y is of length $n^{\frac{1}{3}}$. Now, by comparing every pair of elements from X with the $\leq n^{\frac{2}{3}}$ extracted elements of Y, we can determine the final positions of every element of X and Y in the final merged list in $O(n)$ work and in $O(1)$ rounds. Thus, in total, we have an asymmetric $(n^{\frac{1}{3}}, n)$-merge with $O(n)$ work and in $O(1)$ rounds.

Achieving $O(n)$ work and $O(1)$ rounds is clearly the best one can hope for, and one might imagine that this was only possible since one of the lists was quite small in comparison to the other. Somewhat counter intuitively, [BBD+22] show that they can bootstrap this protocol to construct an asymmetric (n^{α}, n)-merge with $O(n)$ work and in $O(1)$ rounds, for any constant $\alpha < 1$. However, the underlying constants grow rather quickly with α, as $\Omega\left(2^{\frac{2}{(1-\alpha)^3}}\right)$ and $\Omega\left(\frac{1}{(1-\alpha)^3}\right)$, respectively. This makes the case of $\alpha = \frac{1}{3}$ compelling as it is also practically interesting. For more concrete details, we refer to Section 6.2.

Step 1: Symmetric merge with $O(n \log\log n)$ work and in $O(\log\log n)$ rounds. This is the protocol described in Figure 7 and Sections 6.1 and 6.2 of [BBD+22]. It is a divide-and-conquer style protocol that repeatedly reduces a problem of size t to several subproblems of size $t^{\frac{2}{3}}$. At a high level, the protocol proceeds as follows. We begin by dividing the lists X and Y of length n into $n^{\frac{1}{3}}$ blocks of size $n^{\frac{2}{3}}$. Then, do a "prepared merge"[1] of the $n^{\frac{1}{3}}$ *medians* of X and all of Y, and the $n^{\frac{1}{3}}$ *medians* of Y and all of X. These merges are both asymmetric $(n^{\frac{1}{3}}, n)$-merges, and thus can be performed with $O(n)$ work and in $O(1)$ rounds. This is a common tool used by [BBD+22] and our work. There are several things one can learn from such prepared merges, for instance, *which* blocks of X (resp. Y) merge with *how many* elements of Y (resp. X), with *which* block of X (resp. Y) does each element of Y (resp. X) merge, *how many complete blocks* of X (resp. Y) appear before each element of Y (resp. X), etc.

Now, consider the various positions in list X where the $n^{\frac{1}{3}}$ medians of Y landed *along with* the medians of X itself ($2n^{\frac{1}{3}}$ positions in total). Note that we know these positions thanks to the "prepared merge" of the $n^{\frac{1}{3}}$ *medians* of Y and X. These $2n^{\frac{1}{3}}$ medians divide X into $2n^{\frac{1}{3}}$ *chunks*. The reason we call them chunks and not blocks is that they may be of varying lengths. The crucial observation is that no chunk can be larger than $n^{\frac{2}{3}}$. This is easy to see as if we consider just the $n^{\frac{1}{3}}$ medians of X, the resulting chunk are all blocks of size $n^{\frac{2}{3}}$; the $n^{\frac{1}{3}}$ medians of Y can only further split these blocks into smaller chunks. We proceed to extract the $2n^{\frac{1}{3}}$ chunks of X, each padded up with dummy elements to $n^{\frac{2}{3}}$ for obliviousness. This can be done within the tag-shuffle-reveal paradigm using suitable extraction protocols with $O(n)$ work and in $O(1)$ rounds (see [BBD+22] for details). Let the chunks be $C_1, \ldots, C_{2n^{\frac{1}{3}}}$.

Similarly, we consider the positions in list Y where the $2n^{\frac{1}{3}}$ medians of X and Y land. The $2n^{\frac{1}{3}}$ medians divide X into $2n^{\frac{1}{3}}$ *chunks*, which we extract as before with $O(n)$ work and in $O(1)$ rounds. Let the chunks be $C'_1, \ldots, C'_{2n^{\frac{1}{3}}}$. Next, observe that the result of the merge of X and Y is simply the concatenation of the merges of C_i and C'_i for $i = 1, \ldots, 2n^{\frac{1}{3}}$ (aside from removing all the dummy elements we may have added). For an illustration of this, we refer to Figure 6

[1] A "prepared merge" determines the final indices of all elements in the result of a merge without actually performing the merge itself. This can be done in asymptotically the same amount of work and in asymptotically the same number of rounds as the merge itself.

in Section 6.1 of [BBD$^+$22]. This means that we have essentially been able to reduce the problem of merging X to Y to $2n^{\frac{1}{3}}$ subproblems of size $n^{\frac{2}{3}}$ of merging C_i and C'_i for $i = 1, \ldots, 2n^{\frac{1}{3}}$ with $O(n)$ work and in $O(1)$ rounds.

At this point, we have everything in place to simply recurse. After recursing $O(\log\log n)$ times, the sizes of the subproblems will be $O(1)$ and they can be merged with $O(1)$ work and in $O(1)$ rounds each. After this, we will have to extract and remove the added dummy elements, but we will come back to this. Overall, this protocol as is would run in $O(\log\log n)$ rounds. However, it would require more work than we would like. This is because one problem of size n results in up to $2n^{\frac{1}{3}}$ subproblems of size $n^{\frac{2}{3}}$. Proceeding this way, total number of elements we are handling doubles on every recursion and so we would end up with at most $2^{O(\log\log n)} \cdot n = \omega(n \log\log n)$ subproblems of size $O(1)$ and hence the protocol would need $\omega(n \log\log n)$ work. To get around this, [BBD$^+$22] uses a counting argument to bound the number of real, i.e., non-dummy subproblems. Specifically, they show that there are at most $4n^{1-\left(\frac{2}{3}\right)^d} = O(n)$ non-dummy subproblems at the dth level of the recursion, for $d = \log_{\frac{3}{2}} \log n - O(1)$. Thus, prior to every recursion, we simply extract up to $4n^{1-\left(\frac{2}{3}\right)^d} = O(n)$ non-dummy subproblems and recurse on them. This can be done once again with $O(n)$ work and in $O(1)$ rounds.

Overall, all operations in a single recursive step can be performed with $O(n)$ work and in $O(1)$ rounds. Since there are $O(\log\log n)$ recursions, this takes $O(n \log\log n)$ work and $O(\log\log n)$ rounds. In the end, we remove the remaining dummy elements with $O(n)$ work and in $O(1)$ rounds. Thus, in total, we have a symmetric merge with $O(n \log\log n)$ work and in $O(\log\log n)$ rounds. This protocol is already close to the final goal. Indeed, the number of rounds is what we were hoping for. The rest of the structure of [BBD$^+$22]'s final protocol makes use of this protocol on slightly smaller problems to shave off the $\log\log n$ factor in the work.

Step 2: Asymmetric $(\frac{n}{\log\log n}, n)$ merge with $O(n)$ work and in $O(\log\log n)$ rounds. This is the protocol described in Figure 5 and Sections 5.3 and 5.4 of [BBD$^+$22]. At a high level, the protocol proceeds as follows. We begin by dividing the list Y of length n into $k = \frac{n}{\log\log n}$ blocks of size $\frac{n}{k} = \log\log n$. Then, we do a "prepared merge" of X and the $k = \frac{n}{\log\log n}$ *medians* of Y. For this, we invoke the symmetric merge we designed in Step 1 with $O(k \log\log k) = O(n)$ work and in $O(\log\log k) = O(\log\log n)$ rounds. Based on the information obtained from the "prepared merge", we obliviously extract all the blocks of Y that merge with at most $\frac{n}{k} = \log\log n$ elements from X. We also extract the corresponding chunks of at most $\frac{n}{k} = \log\log n$ elements of X for each of the previously extracted blocks of Y. Together, these constitute up to $k = \frac{n}{\log\log n}$ symmetric merges of size $\frac{n}{k} = \log\log n$. [BBD$^+$22] invokes the protocol of [FNO22] on all these instances in parallel. Since each of them would take $O(\frac{n}{k}) = O(\log\log n)$ work and rounds, these symmetric merges can be performed with $k \cdot O(\frac{n}{k}) = O(n)$ work and in $O(\frac{n}{k}) = O(\log\log n)$ rounds. What remains to be handled are the blocks of Y that merge with more than $\frac{n}{k} = \log\log n$ elements from X. Since X is of length

$k = \frac{n}{\log\log n}$, there can only be less than $\frac{k}{\frac{n}{k}} = \frac{k^2}{n}$ such blocks of Y resulting in a total of less than $\frac{n}{k} \cdot \frac{k^2}{n} = k = \frac{n}{\log\log n}$ elements of Y. We obliviously extract all such elements of Y with $O(n)$ work and in $O(1)$ rounds. We also extract the corresponding elements of X that would merge with the previously extracted elements of Y. Then, we invoke the symmetric merge we designed in Step 1 to merge the up to $k = \frac{n}{\log\log n}$ elements of X with the up to $k = \frac{n}{\log\log n}$ elements of Y with $O\left(k \log\log k\right) = O(n)$ work and in $O(\log\log k) = O(\log\log n)$ rounds. At this point, we have enough information for every element of X and Y to determine their final indices in the result of the merge (see [BBD+22] for details). Thus, in total, we have an asymmetric $(\frac{n}{\log\log n}, n)$ merge with $O(n)$ work and in $O(\log\log n)$ rounds. This protocol is again close to the final goal, in a different way than before. Now, the work and number of rounds is what we were hoping for, but this is an asymmetric merge where one of the lists is very slightly smaller than the other. The final step of [BBD+22] makes use of this protocol to overcome this final limitation.

Step 3: Symmetric merge with $O(n)$ work and in $O(\log\log n)$ rounds. This is the protocol described in Figure 4 and Sections 5.1 and 5.2 of [BBD+22]. At a high level, the protocol proceeds as follows. It makes use of the idea of *lossless alignment* which we also use in our protocol Π-Median. The core lemma in this regard is due to [BBD+22] (Lemma 10.1 in [BBD+22], restated as Lemma 1 in this work). It states that, for any k, if one were to find the k medians of X and Y, determine where they merge in the other list and insert dummy blocks of size $\frac{n}{k}$ at each of those spots, then the resulting lists, say X' and Y' (now each of length $2n$), will be *aligned*. I.e., they will have the same $2k$ medians, which themselves are the k medians of X and Y. This is easier to appreciate pictorially and for this we refer to Figure 3 in Section 4 of [BBD+22] and Figure 7 in this work. With this observation, we immediately have the following protocol. We begin by dividing the lists X and Y of length n into $k = \frac{n}{\log\log n}$ blocks of size $\frac{n}{k} = \log\log n$. Then, do a "prepared merge" of the $k = \frac{n}{\log\log n}$ *medians* of X and Y, and the $k = \frac{n}{\log\log n}$ *medians* of Y and X. These merges are both asymmetric $(\frac{n}{\log\log n}, n)$-merges, and thus can be performed with $O(n)$ work and in $O(\log\log n)$ rounds by invoking the asymmetric merge we designed in Step 2. We now have the positions where we would like to insert the dummy elements needed to losslessly align the lists. The dummy elements can be inserted within the tag-shuffle-reveal paradigm in $O(n)$ work and in $O(1)$ rounds (see [BBD+22] or our description of Π-AlignLists using DuplicateMedians in Figure 9 for details). Once the lists have been losslessly aligned, we have $2k = \frac{n}{\log\log n}$ subproblems of symmetric merges of size $\frac{n}{k} = \log\log n$. [BBD+22] invokes the protocol of [FNO22] on all these instances in parallel. Since each of them would take $O(\frac{n}{k}) = O(\log\log n)$ work and rounds, these symmetric merges can be performed with $2k \cdot O(\frac{n}{k}) = O(n)$ work and in $O(\frac{n}{k}) = O(\log\log n)$ rounds. In the end, we remove the dummy elements with $O(n)$ work and in $O(1)$ rounds. Thus, in total, we *finally* have a symmetric merge with $O(n)$ work and in $O(\log\log n)$ rounds.

3.4 Subprotocols and Subprocedures

In this section, we present subprotocols and subprocedures used by our merge protocols. Subprotocols are *interactive*, i.e., they require interaction under MPC. Subprocedures are *local*, i.e., the parties can run them independently. For more complex subprotocols and subprocedures, we include a dedicated figure and explain in text. The simple constructions we present only in text.

Our protocols rely on the following interactive secure protocols:

- $[\![X]\!] \leftarrow \mathsf{\Pi\text{-}Shuffle}([\![X]\!])$ is a key tool for some of our protocols. It takes as input a secret-shared list $[\![X]\!]$, shuffles it according to a *random* permutation (unknown to parties) and returns fresh secret-shares of the permuted list to each party. There are efficient shuffle implementations. E.g., the work of [PRRS24] runs in $O(n)$ time and communication and $O(1)$ rounds.
- $[\![X]\!] \leftarrow \mathsf{\Pi\text{-}Unshuffle}([\![X]\!], [\![\theta]\!])$ undoes $\mathsf{\Pi\text{-}Shuffle}$. To implement $\mathsf{\Pi\text{-}Unshuffle}$, $\mathsf{\Pi\text{-}Shuffle}$ optionally outputs a secret-shared permutation $[\![\theta]\!]$ that remembers the original order of the input list. $\mathsf{\Pi\text{-}Unshuffle}$ then uses $[\![\theta]\!]$ to place the list elements in their original order.
- $[\![\pi]\!] \leftarrow \mathsf{\Pi\text{-}MergeInv}([\![X]\!], [\![Y]\!])$ receives as inputs two secret-shared lists $[\![X]\!]$, $[\![Y]\!]$ of sizes n_0 and n_1. The output is an *inverse* secret-shared permutation π such that $\pi^{-1}(X||Y)$ is merged. Unless we specify otherwise, we assume implementation by evaluating Batcher's merge network with GMW in $O(n \log n)$ time and communication and $O(\log n)$ rounds.
- $[\![\pi]\!] \leftarrow \mathsf{\Pi\text{-}AllPairsMergeInv}([\![X]\!], [\![Y]\!])$ (see Figure 3 for details) is similar to $\mathsf{\Pi\text{-}MergeInv}$ but runs in $O(n_0 \cdot n_1)$ time and communication as it performs secure comparisons between all pairs of elements in X and Y. Its round complexity is $O(1)$.

  The protocol is straightforward. In step 1, we compute $n_0 \cdot n_1$ secure comparisons between all elements of X and Y. This is the only step that requires interaction and outputs secret-sharing of $n_0 \cdot n_1$ bits. By straightforward use of local sums on these bits (steps 2-3), we obtain the counts and output them in step 4.

  Optionally, the input lists can include dummies. Depending on the values of the dummies, this can break the condition that the input lists are sorted. In turn, this can break some of our protocols. As a result, we provide another $\mathsf{\Pi\text{-}AllPairsMergeInv}$ that can handle dummy values. More specifically, the $\mathsf{\Pi\text{-}AllPairsMergeInv}$ additionally takes as input bitvectors $[\![X.\mathsf{IsReal}]\!]$, $[\![Y.\mathsf{IsReal}]\!]$, which indicate whether an element of X is real or a dummy, and outputs a permutation $[\![\pi]\!]$ that places all dummies at the end.

  We now explain the protocol. As in $\mathsf{\Pi\text{-}AllPairsMergeInv}$, we first perform secure comparison between all X and Y (step 1). In steps 2-4, we compute inIdx, which denotes the number of real elements before each X_i in X (and similarly for Y). In steps 5-6, we compute the final index of all real elements in $X \bigsqcup Y$. Now we just need to ensure all dummies are placed behind the real elements. As we did for the real elements, we compute the final index of all dummy elements. The first dummy starts at the end of all real elements (step 7). Then it increases by 1 with each dummy in X (step 8) and similarly with each dummy in Y (steps 9-10). At this point, we hold a dummy index and a real index for each element of X and Y. We need to obliviously select one of them depending on the IsReal bit. We do that in steps 11 for X and 12 for Y. We now hold an inverse permutation and return it (step 13).

Π-AllPairsMergeInv Protocol

Input: Secret-shared lists $\llbracket X \rrbracket$ and $\llbracket Y \rrbracket$ such that $|X| = n_0$ and $|Y| = n_1$. The lists may optionally include dummies. Dummies are denoted as $X_i.\mathsf{isReal} = 0, Y_i.\mathsf{isReal} = 0$.
Output: Secret-shared inverse permutation $\llbracket \pi \rrbracket$ such that $\pi^{-1}(X||Y)$ is merged with all dummies at the end.

```
Π-AllPairsMergeInv($\llbracket X \rrbracket, \llbracket Y \rrbracket$) :

 1. $\llbracket e_{i,j} \rrbracket := \llbracket X_i \rrbracket > \llbracket Y_j \rrbracket \ \forall i \in [n_0],\ j \in [n_1]$
 2. $\llbracket c_{X,i} \rrbracket := \Sigma_{j=0}^{n_1-1} \llbracket e_{i,j} \rrbracket$
 3. $\llbracket c_{Y,i} \rrbracket := \Sigma_{j=0}^{n_0-1} \neg \llbracket e_{j,i} \rrbracket$
 4. return $(\llbracket c_X \rrbracket + [n_0]) || (\llbracket c_Y \rrbracket + [n_1])$

Π-AllPairsMergeInv($\llbracket X \rrbracket, \llbracket Y \rrbracket, \llbracket X.\mathsf{isReal} \rrbracket, \llbracket Y.\mathsf{isReal} \rrbracket$) :

 1. $\llbracket e_{i,j} \rrbracket := \llbracket X_i \rrbracket > \llbracket Y_j \rrbracket \ \forall i \in [n_0],\ j \in [n_1]$
 2. $\llbracket X_0.\mathsf{inIdx} \rrbracket := 0, \llbracket Y_0.\mathsf{inIdx} \rrbracket := 0.$
 3. for $i \in [1, n_0]$ : $\llbracket X_i.\mathsf{inIdx} \rrbracket := \llbracket X_{i-1}.\mathsf{inIdx} \rrbracket + \llbracket X_i.\mathsf{isReal} \rrbracket$
 4. for $i \in [1, n_1]$ : $\llbracket Y_i.\mathsf{inIdx} \rrbracket := \llbracket Y_{i-1}.\mathsf{inIdx} \rrbracket + \llbracket Y_i.\mathsf{isReal} \rrbracket$
 5. $\llbracket X_i.\mathsf{realIdx} \rrbracket := \llbracket X_i.\mathsf{inIdx} \rrbracket + \Sigma_{j=0}^{n_1-1} \llbracket Y_j.\mathsf{isReal} \rrbracket \llbracket e_{i,j} \rrbracket$
 6. $\llbracket Y_i.\mathsf{realIdx} \rrbracket := \llbracket Y_i.\mathsf{inIdx} \rrbracket + \Sigma_{j=0}^{n_0-1} \llbracket X_j.\mathsf{isReal} \rrbracket \, (\neg \llbracket e_{j,i} \rrbracket)$
 7. $\llbracket X_0.\mathsf{dummyIdx} \rrbracket := \sum_{i \in [n_0]} \llbracket X_i.\mathsf{isReal} \rrbracket + \sum_{i \in [n_1]} \llbracket Y_i.\mathsf{isReal} \rrbracket$
 8. for $i \in [1, n_0]$ : $\llbracket X_i.\mathsf{dummyIdx} \rrbracket := \llbracket X_{i-1}.\mathsf{dummyIdx} \rrbracket + (\neg \llbracket X_i.\mathsf{isReal} \rrbracket)$
 9. $\llbracket Y_0.\mathsf{dummyIdx} \rrbracket := \llbracket X_{n_0-1}.\mathsf{dummyIdx} \rrbracket + (\neg \llbracket Y_0.\mathsf{isReal} \rrbracket)$
10. for $i \in [1, n_1]$ : $\llbracket Y_i.\mathsf{dummyIdx} \rrbracket := \llbracket Y_{i-1}.\mathsf{dummyIdx} \rrbracket + (\neg \llbracket Y_i.\mathsf{isReal} \rrbracket)$
11. $\llbracket \pi_{[n_0]} \rrbracket \quad := \llbracket X.\mathsf{isReal} \rrbracket \cdot (\llbracket X.\mathsf{realIdx} \rrbracket - \llbracket X.\mathsf{dummyIdx} \rrbracket) + \llbracket X.\mathsf{dummyIdx} \rrbracket$
12. $\llbracket \pi_{[n_1]+n_0} \rrbracket := \llbracket Y.\mathsf{isReal} \rrbracket \cdot (\llbracket Y.\mathsf{realIdx} \rrbracket - \llbracket Y.\mathsf{dummyIdx} \rrbracket) + \llbracket Y.\mathsf{dummyIdx} \rrbracket$
13. return $\llbracket \pi \rrbracket$
```

Fig. 3: Π-AllPairsMergeInv merges input lists of sizes n_0, n_1 respectively. It runs in $O(n_0 \cdot n_1)$ communication and $O(1)$ rounds.

- $\llbracket X \rrbracket \leftarrow$ Π-Permute($\llbracket X \rrbracket, \llbracket \pi \rrbracket$) rearranges a secret-shared input list according to a secret-shared permutation. More specifically, it takes as input a secret-shared list $\llbracket X \rrbracket$ of size n and a secret-shared permutation $\llbracket \pi \rrbracket : [n] \rightarrow [n]$. It permutes $\llbracket X \rrbracket$ according to $\llbracket \pi \rrbracket$ and returns the secret-shared result. Π-Permute runs in linear time and communication and $O(1)$ rounds.

- $[\![X]\!] \leftarrow \mathsf{\Pi\text{-}PermuteInv}([\![X]\!],[\![\pi]\!])$ is similar to $\mathsf{\Pi\text{-}Permute}$ but rearranges the input list according to *inverse* permutation. In our construction, we extensively use inverse permutations as they are convenient for some of our constructions.
- $[\![\pi]\!] \leftarrow \mathsf{\Pi\text{-}Inv}([\![\pi]\!])$ takes as input a secret-shared *inverse* permutation and turns it into a secret-shared permutation.
- $[\![X]\!] \leftarrow \mathsf{\Pi\text{-}ExtractOrdered}([\![X]\!])$ receives as input a secret-shared list $[\![X]\!]$ that is interspersed with dummy elements. A bit $[\![X_i.\mathsf{IsReal}]\!]$ indicates if $[\![X_i]\!]$ is a dummy or not. $\mathsf{\Pi\text{-}ExtractOrdered}$ extracts and outputs only the $[\![X_i]\!]$ s.t. $X_i.\mathsf{IsReal} = 1$. A key feature of $\mathsf{\Pi\text{-}ExtractOrdered}$ is that the output elements retain the *order* of the input list. We use [PRRS24]'s efficient protocol that runs in $O(n)$ time and communication and uses $O(1)$ rounds.
- $[\![X]\!] \leftarrow \mathsf{\Pi\text{-}ExtractOrderedPad}([\![X]\!], c)$ is similar to $\mathsf{\Pi\text{-}ExtractOrdered}$. Sometimes we do not know the true number of non-dummies. Hence, we also pass an upper bound c on the number of non-dummies as input. We output a secret-shared list of all non-dummies padded to c elements. We again use [PRRS24]'s protocol.
- $[\![X]\!] \leftarrow \mathsf{\Pi\text{-}UnextractOrdered}([\![X]\!],[\![\theta]\!])$ undoes the protocols $\mathsf{\Pi\text{-}ExtractOrdered}$ and $\mathsf{\Pi\text{-}ExtractOrderedPad}$. To implement it, the extraction protocols optionally output a secret-shared permutation $[\![\theta]\!]$ that remembers which elements were extracted. Then, $\mathsf{\Pi\text{-}UnextractOrdered}$ uses this permutation to place the extracted elements back into the original positions in a list.
- $\mathsf{\Pi\text{-}MPC}$ refers to a set of protocols that implement common circuit MPC operations that we collectively refer to as MPC. These include (but are not limited to) interactive AND gates $[\![c]\!] := [\![a]\!] \wedge [\![b]\!]$, comparison gates $[\![c]\!] := [\![a]\!] < [\![b]\!]$, but also local subprocedures such as XOR gates $[\![c]\!] := [\![a]\!] \oplus [\![b]\!]$. In our protocols, we use simple operators such as $\wedge$, $<$, and $\oplus$ to invoke these operations. MPC is used extensively in our proofs.
- $[\![B']\!] := \mathsf{\Pi\text{-}AggregationTree}([\![B]\!],[\![c]\!], f)$ [BDG$^+$22] is the aggregation tree protocol that we extensively use in our constructions to efficiently distribute information between blocks. For inputs, the aggregation tree receives a list of blocks B, a list of control bits c such that $|B| = |c| = n$, and a flag $f \in \{\mathsf{prefix}, \mathsf{suffix}\}$. It then outputs another list of blocks B' of the same size as B, c. Conceptually, c partitions B into sublists. The output list B' is obtained by copying the first item of each sublist into the rest of the sublist. In more detail, if $f = \mathsf{prefix}$, then $B'_i := B_i$ when $c_i = 0$ else $B'_i = B'_{i-1}$. For example, when $B = \{B_0, B_1, B_2, B_3, B_4\}$ and $c = \{0, 0, 1, 1, 0\}$, then $B' = \{B_0, B_1, B_1, B_1, B_4\}$. If $f = \mathsf{suffix}$, the output is as expected. For example, for the same c, $B' = \{B_0, B_3, B_3, B_3, B_4\}$. $\mathsf{\Pi\text{-}AggregationTree}$ runs in linear time with $O(\log n)$ rounds.

Our protocols also rely on the following local subprocedures:

- $[\![X']\!] \leftarrow \mathsf{ComputeMedians}([\![X]\!], k)$ is parameterized by integer k, the number of medians. It takes as input a secret-shared list $[\![X]\!]$ of length n. The output is a secret-shared list of k medians $[\![X']\!]$ s.t. $[\![X'_i]\!] = [\![X_{(i+1)\frac{n}{k}-1}]\!]$.

- $\llbracket X' \rrbracket \leftarrow \mathsf{DuplicateMedians}(\llbracket X' \rrbracket, \frac{n}{k})$ takes as input a secret-shared list of medians $\llbracket X' \rrbracket$ output by ComputeMedians and duplicates each median $\frac{n}{k}$ times.
- $\llbracket \pi' \rrbracket \leftarrow \mathsf{UpdateInvPermutation}(\llbracket \pi \rrbracket, k, n)$ (see Figure 4) takes as input a secret-shared inverse permutation π that merges $X'||Y$ (in our case initially $[k+n] \to [k+n]$) and adjusts it to account for the fact that medians were duplicated in DuplicateMedians, resulting in a secret-shared permutation $[2n] \to [2n]$.

  The subprocedure consists of 4 steps. Step 1 initializes an empty inverse permutation $\pi' : [2n] \to [2n]$, steps 2-3 fill in this permutation, and step 4 outputs it. We now look at steps 2-3 more closely. Step 2 sets entries of π' corresponding to the first list of size k with each entry copied $\frac{n}{k}$ times. Hence, it sets $\pi'_{[n]}$. Step 3 then sets the remaining entries $\pi'_{[n,2n]}$ corresponding to the second list of size n. The idea in both steps is simple: add the number of elements from the *other* list preceding the current element (this can be derived from the input permutation π) with the number of elements in the *current* list preceding the current element.

UpdateInvPermutation **Subprocedure**

Input: List sizes k and n such that $k < n$ and a secret-shared permutation $\llbracket \pi \rrbracket : [k+n] \to [k+n]$ that merges the two lists.
Output: Secret-shared permutation $\llbracket \pi' \rrbracket : [2n] \to [2n]$ that merges the two lists if each element of the first list of size k is copied $\frac{n}{k}$ times.

```
UpdateInvPermutation($\llbracket \pi \rrbracket, k, n$) :

1. $\llbracket \pi' \rrbracket : [2n] \to [2n]$
2. $\llbracket \pi'_{i\frac{n}{k}+j} \rrbracket := \llbracket \pi_i \rrbracket - i + i \cdot \frac{n}{k} + j,\ \forall i \in [k], j \in \left[\frac{n}{k}\right]$
3. $\llbracket \pi'_{n+i} \rrbracket := (\llbracket \pi_{k+i} \rrbracket - i)\frac{n}{k} + i,\ \forall i \in [n]$
4. return $\llbracket \pi' \rrbracket$
```

Fig. 4: UpdateInvPermutation computes an inverse permutation $[2n] \to [2n]$ that merges two lists of sizes k and n if each element of the first list is copied $\frac{n}{k}$ times. This is a local operation and does not require interaction.

3.5 Simulator for Our Protocols

Simulating our protocols is straightforward and follows naturally from a simple composition argument. All of our merge protocols can be viewed as circuits composed of both interactive and non-interactive gates. The circuits include more complex gates/subprotocols, such as Permute and ExtractOrdered, in addition to the standard Π-MPC circuit gates (AND, XOR, and secure comparison). All gates work with secret shares. I.e., they receive a secret-sharing of the inputs and output a secret-sharing of the outputs. They have already been proven simulatable in other works. Thus, the simulator, given the adversary's input and output shares, simply goes through the circuit gate-by-gate. The simulator for our merge protocol invokes the simulators for each gate/subprotocol. These simulators do not require knowledge of the underlying plaintext values.

4 Technical Overview

We now explain our constructions at a high level. We start with our *symmetric* constructions, which take as input two secret-shared lists X and Y of size n and output a secret-shared permutation $[2n] \rightarrow [2n]$ such that $\pi(X||Y)$ is merged. In Section 4.1 we introduce Logstar, our bandwidth-optimized construction, and then continue in Section 4.2 with Median, our rounds-optimized construction. We then present SquareRootMerge (Section 4.3), our *asymmetric* construction for merging lists of length $n^{\frac{1}{2}}$ and n. Sections 5, 6, and 7 present our constructions in formal detail.

Our goal is to optimize for *concrete efficiency*. Asymptotically, Logstar runs in $O(n \log^* n)$ time and communication and uses $O(\log n)$ rounds; Median runs in $O(n \log^c n)$ time and communication and uses $O(\log \log n)$ rounds, where $c = \log_{\frac{3}{2}} 2 \approx 1.71$; SquareRootMerge runs in $O(n)$ time and communication and uses $O(\log n)$ rounds.

4.1 Logstar High Level Explanation

At a high level, the Π-Logstar construction divides the lists X, Y into blocks, invokes Π-MergeInv to merge the lists of blocks, and then recurses on adjacent blocks. The approach was first used in [FO21] and can be summarized as follows: divide $X||Y$ into blocks of size $b := O(\log n)$ (to be precise, [FO21] used polylog n) and merge these $k := 2\frac{n}{b}$ blocks using the first element as the block value (each block is treated as a unit). Since there are $k = O(\frac{n}{\log n})$ blocks, this can be done by even a naive $k \log k$ time and $\log k$ rounds sort protocol. After the block merge reorders the blocks, the hope is that the individual items are close to their final merged positions. [FO21] proceeds by performing a sliding window merge sort which sequentially sorts adjacent blocks together, i.e., merge-sort blocks $(i, i+1)$, then $(i+1, i+2)$, etc. Due to the sequential nature of this sort, most items will be carried along to their final position. [FO21] proved that the sliding window merge correctly places all but a small number of so called *strays*, which can be extracted and merged separately. Unfortunately for [FO21], sequentially merging adjacent blocks introduces a near-linear round complexity overhead. Note that [BBD$^+$22] states that [FO21] has $\log n$ round complexity, which we confirmed to be incorrect.

Our approach deviates significantly in how we merge adjacent blocks together and handle the strays. In particular, after reordering the blocks, Π-Logstar makes careful use of a so-called *aggregation tree* [BDG$^+$22] to efficiently distribute information between the blocks. This in turn allows Π-Logstar to merge adjacent blocks in parallel instead of sequentially. This combination of techniques allows our protocol to reduce the round complexity of [FO21] from near-linear to $O(\log n)$ while simultaneously reducing the running time from $O(n \log\log n)$ to $O(n \log^* n)$. In more detail, Π-Logstar proceeds as follows:

1. **[Block Merge]** Select the $k = \frac{n}{b} = O(\frac{n}{\log n})$ medians of X, Y, $X' :=$ Π-ComputeMedians$(X), Y' :=$ Π-ComputeMedians(Y) and compute a permutation that would merge them $\pi =$ Π-MergeInv(X', Y'). Then permute the b-sized blocks of X, Y by π, i.e., $\pi(B)$ where $B := B^X || B^Y$ are the blocks of $X||Y$.

That is, Π-ComputeMedians outputs the k medians which are merged with any efficient merging/sorting algorithm. This outputs a permutation $\pi : [2k] \to [2k]$ that merges $(X'||Y')$. The original lists are divided into blocks of size $b = O(\log n)$ and permuted by π. Because the initial lists were sorted, most elements in B are now close to their final position. For example, it is easy to see that the location of the X', Y' medians in B will be at most $b - 1$ positions from their final position. However, it is possible for some elements to be far from their final position when many blocks from the other list merge into the middle of a block. An example of this can be seen in Figure 5.

The example considers two lists X (red) and Y (blue), where the first two blocks of X are $[1, 10, 15, 16, 22, 45, 51]$ and $[61, 62, 63, 64, 65, 66, 70]$, and the first four blocks of Y are $[11, 12, 13, 14, 17, 18, 19]$, $[21, 23, 24, 25, 26, 27, 29]$, followed by $[31, 32, 33, 34, 37, 38, 39]$ and $[41, 42, 43, 44, 67, 67, 68]$. When the blocks are merged using their first elements as the block values, they get reordered as the first block of X, the first four blocks of Y, and then the second block of X. We label these reordered blocks as $B_0, \ldots, B_5$. As the figure shows, block B_0 has elements that merge with the next 4 blocks, $B_1, \ldots, B_4$. If an element of a block has a value that falls within the range of the next blocks, we call it a *stray*. We make the following observations about strays:

- **Observation 1:** If a block B_i has strays, then the next block B_{i+1} must come from the other list (i.e., if B_i is from X and has strays, then B_{i+1} is from Y, or vice versa). This means that a block B_i can only contain a stray if it is a *transition block*, i.e., a block from one list that is followed by a block from the other list. In the example, B_0 and B_4 are the transition blocks and they are the only blocks with strays.
- **Observation 2:** If a block B_i has strays, then all blocks following B_i until the next transition block will come from the other list. In the example, blocks $B_1, \ldots, B_4$ are all from Y.
- **Observation 3:** All strays in B_i can only belong to $B_{i+1}, \ldots, B_j$, where j is the next transition point, i.e., all strays from block B_i will be distributed among blocks until the next transition point j. Similarly, $B_{i+1}, \ldots, B_j$ only contain strays from B_i. In the example, the strays from block B_0 are distributed among the blocks $B_1, \ldots, B_4$, and the blocks $B_1, \ldots, B_4$ only contain strays from block B_0.

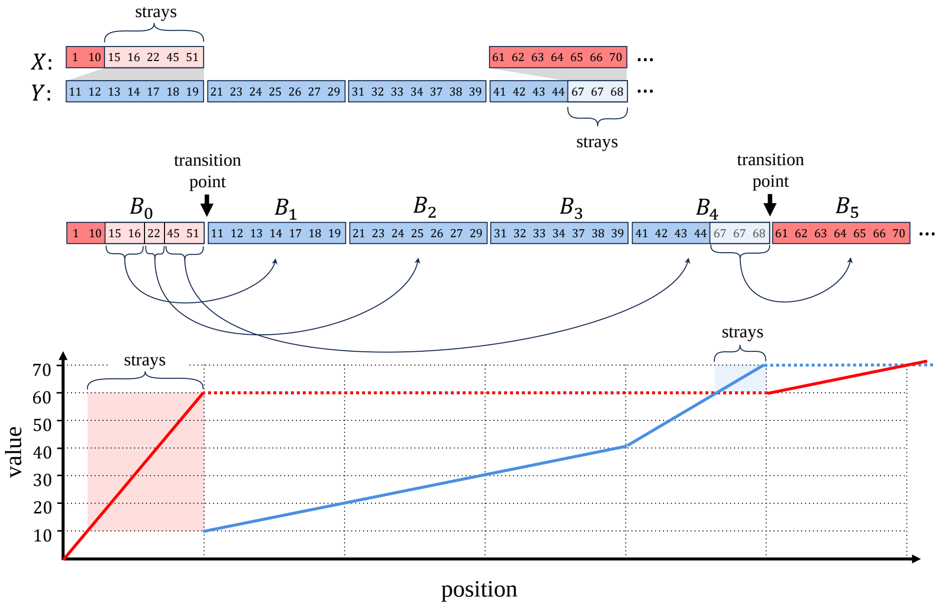

Fig. 5: This figure shows a segment of a plausible state for the merge after blocks are merged. Note that there are two transition blocks B_0 (first block of X) and B_4 (fourth block of Y). By observation 1, they are the only blocks that can have strays. By observation 2, all blocks starting with B_1 until the next transition block come from the list Y. By observation 3, all strays in B_0 belong to blocks $B_1, \ldots, B_4$ from Y.

Based on these observations we arrive at our high level strategy: map the strays of each block to the subsequent blocks that they *belong to* and then recursively merge these strays with those blocks. This is visualized in Figure 6 and detailed in the following steps:

2. **[Duplicate]** We obliviously duplicate each transition block B_i onto the next streak of blocks from the other lists, e.g., Figure 6 duplicates B_0 onto blocks $B_1, ..., B_4$. For each B_j, let S_j denote the duplicated block that is associated with it, e.g., $S_1 = \ldots = S_4 = B_0$ in Figure 6. Recall from our observations that S_i contains all strays that belong to B_i. Naively duplicating the blocks would require $O(n)$ rounds. However, [BDG+22] introduced the so called aggregation tree protocol Π-AggregationTree that requires linear time and $O(\log n)$ rounds.
3. **[Extract]** We next extract from S_i only the strays that belong to B_i. This can be done trivially in $O(n)$ time and $O(1)$ rounds by selecting all elements of S_i in the range $[B_{i,0}, B_{i+1,0})$, i.e., the first element of B_i and the first element of the next block B_{i+1}. The remaining elements of S_i are marked as

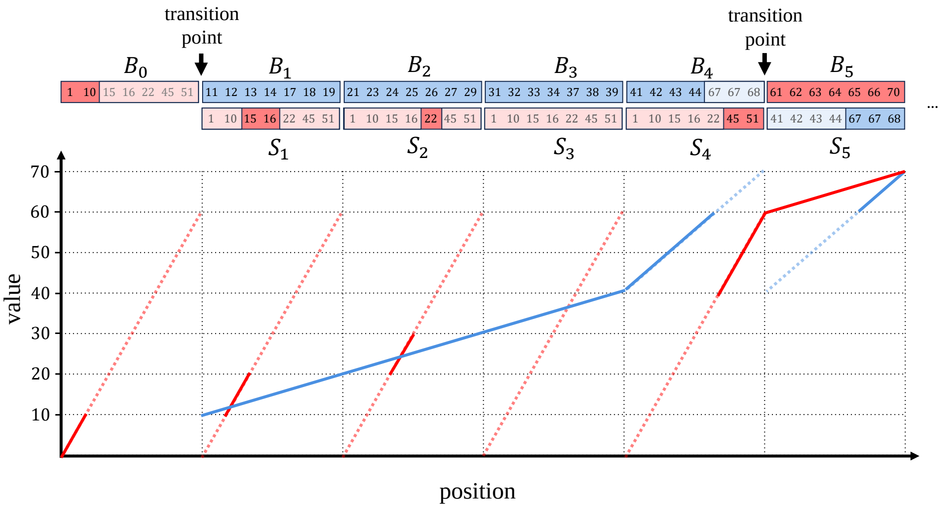

Fig. 6: In this figure we give a visual aid for steps 2 and 3 of Π-Logstar. We build on the example in Figure 5. We show the blocks $S_1, \ldots, S_5$ copied into $B_1, \ldots, B_5$. We then highlight the parts of the copied blocks with strays belonging to the corresponding block. The rest of the copied blocks are marked off as dummies.

dummies such that S_i retains $\log n$ elements. We similarly extract from B_i only the elements that are smaller than $B_{i+1,0}$.

After step 3, we hold $2k$ blocks B_i and their corresponding blocks of strays (and dummies) S_i. Both are of size $b = O(\log n)$. We now finish our merge:

4. **[Recurse]** Recursively call steps 1-3 on each (B_i, S_i) until they are of constant size, at which point we merge with another protocol. As the blocks reduce to log size at each recursion, we need $O(\log^* n)$ recursive steps to get to a constant. In practice, for any feasible list size, we never need to do more than 2 recursive calls before using a naïve protocol for the base case.
5. **[Reconstruct]** We now concatenate outputs from step 4. Note that at each of the $\log^* n$ recursive steps, we double the list size by inserting S_i next to each block B_i. We also double the list size by merging X and Y together. I.e., the list is now of length $O(n \cdot 2^{\log^* n})$. We need to remove all the extra (all but $2n$) dummy elements from the merged list. We use an in-order extraction technique from [PRRS24] (see Section 3).

We now explain at a high level why Π-Logstar runs in $O(n \cdot 2^{\log^* n})$ time and communication and uses $O(\log n)$ rounds.

Time and Communication. Recall that in step 4 we extract block S_i for each block B_i and then recursively merge them. Thus, we effectively double the size of the list $X||Y$ (step 1) in each recursive call. We also double once to merge $X||Y$. As we recurse $O(\log^* n)$ times, the list is of size $O(n \cdot 2^{\log^* n})$ after the last recursive call. Each recursive call is linear time and communication, and thus the resulting complexity is $O(n \cdot 2^{\log^* n})$.

Rounds. We use aggregation trees to copy blocks S_i in step 4, which require $\log n$ rounds (the remaing steps run in $O(1)$ rounds). We execute step 4 at each of the $\log^* n$ recursive calls. While a loose analysis would result in $O(\log^* n \cdot \log n)$ rounds, the round complexity gets smaller in each recursive call as the size of the blocks reduces exponentially each time. Specifically, the number of rounds required in the ith level of the recursion is $c \cdot \log^{(i)} n$, where c is a fixed constant and $\log^{(i)}$ is i iterations of the log function. It is easy to see (inductively) that $\log^{(i)} n \leq \frac{\log n}{2^{i-1}}$. Therefore, the round complexity is

$$c \cdot \sum_{i \geq 1} \log^{(i)} n \leq c \cdot \sum_{i \geq 1} \frac{\log n}{2^{i-1}} = O(\log n)$$

Getting to $n \log^* n$. We now describe how to optimize $\mathsf{\Pi\text{-}Logstar}$ to get time and communication down from $O(n \cdot 2^{\log^* n})$ to $O(n \log^* n)$. Intuitively, we need to prevent the list size from doubling every recursive call while still performing roughly linear work in every recursive step. The key idea is the following: When we duplicate a block containing strays out onto the blocks where the strays may belong, there are several copies of the block that are created, but for any given stray element, only one of them is used (the element will be turned into a dummy in all other copies). This means that there are going to be several merges involving completely dummy blocks in the later stages of the recursion. If we prevent recursing on such subproblems, we will prevent this perpetual doubling of the size of the list at every recursive step. Care must be taken to prune these subproblems obliviously, but we show how to do it in Section 5.3.

4.2 Median High Level Explanation

In this construction we build on the insecure medians-based approach of [Val75] (see Section 3.2), first explored in the secure context by [BBD+22]. Recall [Val75]'s approach selects evenly-spaced k-medians of X, i.e. it computes $X' = \mathsf{ComputeMedians}(X, k) = \{X_0, X_b, \ldots, X_{n-b}\}$, where $b = \frac{n}{k}$. The positions of these medians will partition X into k evenly sized *blocks*, e.g. the first block $\{X_0, \ldots, X_{b-1}\}$. Similarly, these medians $X' = \mathsf{ComputeMedians}(X, k)$ will partition Y into variable-sized *chunks* with the ith chunk falling in the range $[X_{ib}, X_{ib+b})$, see (1) in Figure 7.

The challenge is that in secure computation, the size of the variable-sized chunks must remain secret as we cannot leak the number of elements of Y lying between successive medians of X. The first step of this approach is to *obliviously align* the subproblems as suggested in [BBD+22]'s lemma (see Section 6.3). Conceptually, this means we will insert n dummy elements into the X, Y lists such

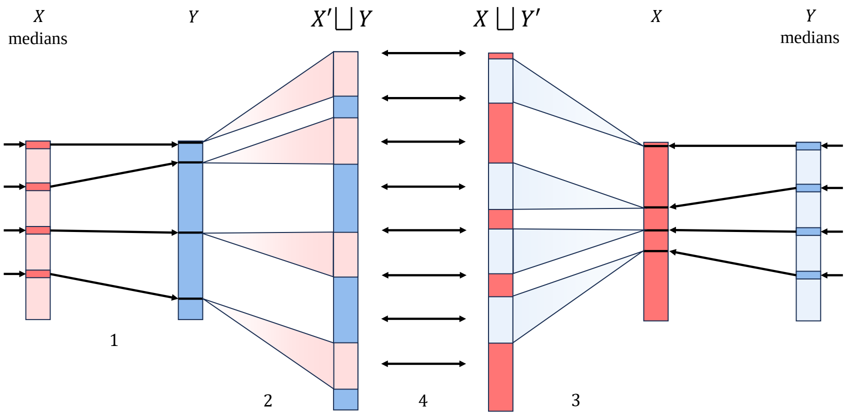

Fig. 7: This figure shows steps 1-4 of Median (c.f. Section 4.2). In (1), we retrieve medians X' of X and compute where they map in Y (i.e., we merge them). (2) copies each element of X' such that $|X'| = |Y|$. (3) repeats steps 1-2 with the lists switched. In (4), the lists are aligned and can be recursively merged. Step 5 is a simple concatenation of the outputs from step 4, and hence we do not display it in the figure.

that the ith block of X will start at the same position as the ith chunk of Y. Once aligned, we can define aligned subproblems and recursively solve them.

We now discuss this in more detail. An example of this approach can be seen in Figure 7. Let $k = n^{\frac{1}{3}}$ be the number of medians. The block size will be $b = n^{\frac{2}{3}}$.

1. [(k,n)**-merge**] The first step is to determine where the k medians of X map to in Y. We can do this in $O(n)$ time and communication and $O(1)$ rounds by using the highly-efficient Π-CubeRootMerge (a modified protocol of [BBD+22]) that merges lists of $k = n^{\frac{1}{3}}$ and n elements. However, instead of actually merging the $X' = \mathsf{ComputeMedians}(X,k), Y$ lists, this protocol, as defined in our notation, outputs a secret-shared permutation $\pi : [k+n] \rightarrow [k+n]$ such that $\pi(X'||Y)$ is merged.
2. [**Permute**] Now, instead of merging $(X'||Y)$ via the permutation π, we modify π by invoking UpdateInvPermutation to obliviously merge b dummies where each $x \in X'$ would go. After invoking this modified permutation on $(X'||Y)$, we have a list $X' \bigsqcup Y$ that has all the n elements of Y merged with b copies of the medians $X' = \mathsf{ComputeMedians}(X,k)$, i.e., containing a total of $n = kb$ dummies.
3. [**Repeat**] We repeat steps 1-2 for the other list. I.e., we take medians of Y and merge (b copies of) them with X. After this step, we hold two lists $X' \bigsqcup Y, X \bigsqcup Y'$ of length $2n$ that are *aligned* (see Lemma 1 in Section 6.3). In particular, the $2k$ medians of the two lists coincide and they are in fact the medians X' and Y'. Thus, we have the guarantee that the medians $Y' = \mathsf{ComputeMedians}(Y,k)$, (respectively $X' = \mathsf{ComputeMedians}(X,k)$) will

be in the correct location of $X \bigsqcup Y'$, (respectively $X' \bigsqcup Y$). This guarantee is not trivial to see but allows us to recurse in the next step.

4. **[Recurse]** We now split both lists into $2k$ blocks of length $\frac{n}{k} = n^{\frac{2}{3}}$ and merge them separately (i.e., we merge the first block from the first list with the first block from the second list, etc.). We recursively invoke steps 1-4. It suffices to make $O(\log\log n)$ calls (see bottom of this section for explanation) until our blocks are of constant size, at which point we use a naïve protocol for the base case.
5. **[Reconstruct]** We now concatenate outputs from step 4. Note that at each of the $O(\log\log n)$ recursive steps, we double the list size by inserting dummy medians to each list and then double it again when we merge the lists. I.e., the list is now of length $n \cdot 2^{O(\log\log n)} = n \cdot \text{polylog } n$. We need to remove all the extra (all but $2n$) dummy elements from the merged list. We use an *extraction* technique of [PRRS24,BBD$^+$22]. I.e., whenever we insert the copies of the medians in Step 2, we mark them as dummies. We now extract (in order) the elements not marked as dummies to get the merged list.

We note this this protocol shares some similarities to Protocol 1 of [BBD$^+$22] in that both protocols use Lemma 1 to first align the lists. However, apart from this alignment, our protocols deviate significantly. Our approach, as described above, uses relatively large blocks ($n^{\frac{2}{3}}$ size) and a recursive structure to merge the lists. In contrast, [BBD$^+$22] uses extremely small blocks ($\log\log n$ size), which can directly be merged using [FNO22], with no recursion. However, this approach requires a $O(n)$ time subprotocol for merging $k = n/\log\log n$ medians with a size n list. [FNO22] achieved the desired asymptotics through a series of complex (and intriguing) protocols. However, the complexity of their subprotocols results in a significantly higher round complexity as detailed in Section 8.

We now explain at a high level why Median runs in $O(n \log^c n)$ time and communication and uses $O(\log\log n)$ rounds.

Rounds. We require $O(\log\log n)$ recursive calls to make the subproblems constant size. This is because the subproblem sizes in each recursive call reduces in size by a factor of $\frac{2}{3}$ in the exponent. Therefore, the size of the subproblems in the ith recursive call is $n^{\left(\frac{2}{3}\right)^i}$. Each recursive call requires $O(1)$ rounds. The value of i required for this to be a constant is $c\log\log n = O(\log\log n)$, where $c = \log_{\frac{3}{2}} 2$. Hence, we use a total of $O(\log\log n)$ rounds.

Time and Communication. At first look, each recursive step has a linear cost (in the length of the list) and we have $O(\log\log n)$ steps. Thus, we would expect $O(n\log\log n)$ time and communication. But the total size of the subproblems doubles at each step, resulting in $O(n \cdot \text{polylog } n)$. Specifically, since we make at most $c\log\log n$ recursive calls and each recursive step has a linear cost (in the length of the list), the time and communication is

$$O\left(\sum_{i=0}^{c\log\log n-1} 2^i \cdot n\right) = O\left(n \cdot 2^{c\log\log n}\right) = O(n\log^c n)$$

where $c = \log_{\frac{3}{2}} 2$ as before.

4.3 SquareRootMerge High Level Explanation

This protocol is designed for input lists of sizes $n^{\frac{1}{2}}$ and n. Like Π-Median, Π-SquareRootMerge is closely inspired by Valiant's [Val75] plaintext medians-based approach and the secure merge of [BBD$^+$22]. Unlike Π-Median, it is not a recursive protocol. At a high level, we split Y into evenly spaced blocks, find and extract for each X_i a block Y_j into which X_i goes, compute the position of X_i in that block, and then extrapolate the position to the full merged list. We now present in more detail.

- **[Find]** We first split Y into evenly-spaced same-size blocks. More specifically, we find $k = n^{\frac{1}{2}}$ medians of Y with ComputeMedians. These medians partition Y into k blocks of size k. We next find into which block of Y each element of X goes. This can be done in linear time by performing $k^2 = n$ secure comparisons.
- **[Extract]** Now that we found the blocks, we want to extract them for each X_i. The challenge is that multiple X_i can go into the same block Y_j. This information must remain oblivious. Our solution once again relies on [BDG$^+$22]'s aggregation trees. First, we extract for each X_i a single block. This block is either a real block Y_j[2] or a dummy all-zero block D_i. We extract D_i if and only if some previous $X_{<i}$ already extracted the real block Y_j. In other words, we extract Y_j only for the smallest X_i that goes into it; otherwise we extract D_i. This step is simple. We mark for each X_i either the Y_j or the D_i we want to extract, shuffle $Y||D$ together, reveal the marks in the clear, and then extract *in order* the marked blocks. The challenge now is to replace the dummy D_i with the real Y_j. This is where we use *prefix* aggregation trees. They replace all $D_{>i}$ following the closest previous Y_j with Y_j. Note that all $D_{>i}$ between the Y_j and the next Y_{j+1} should be replaced with Y_j. This is because the lists are ordered, and hence all elements of X between the extracted Y_j and Y_{i+j} go into the same block Y_j.
- **[Find]** Now that we extracted the correct block Y_j for each X_i, we find where X_i goes in that block. This can be done once again in linear time with $k^2 = n$ secure comparisons.
- **[Extrapolate]** We now extrapolate to find out where all elements of X and Y go in $X \bigsqcup Y$. For X this is simple. We know (1) the position of X_i in its block Y_j, (2) i elements of X come before X_i, and through straightforward housekeeping we learn, and (3) the number of blocks of Y before Y_j. Summing these will yield the final positions of all X_i. For Y, this is more complex as we need to consolidate the positions of different X_i across possibly multiple copies of the same Y_j. We can do that once again with aggregation trees. We use a *suffix* aggregation tree to compute a list that is non-zero only at the indices in Y_j where elements of X are inserted. I.e., the non-zero entries

[2] Note that j can differ from i.

can be viewed as offsets resulting from X_i being merged into the block Y_j. Importantly, these offsets include all elements of X that go into Y_j. At this point, we know where all elements of X go in the extracted blocks. Now, if we can place the extracted blocks with offsets into their initial position in Y and set the remaining unextracted offsets to zeros (i.e., no X_i goes in them), we can do a (1) simple prefix sum and (2) add in j, the number of $Y_{<j}$ to get the final positions of Y in $X \bigsqcup Y$. This can be simply achieved by reversing the block extraction process. I.e., we simply unshuffle the extracted blocks. Now that we know the final positions of all X_i and Y_i in $X \bigsqcup Y$, constructing the merging permutation is straightforward.

Rounds. The only building block that does not run in $O(1)$ rounds is an aggregation tree, which runs in $O(\log n)$ rounds. We execute aggregation trees twice; Π-SquareRootMerge runs in $O(\log n)$ rounds.

Time and Communication. We perform secure comparisons to (1) find the block Y_j for each X_i and to (2) find where each X_i goes in Y_j. Both steps require $k^2 = n$ comparisons, and thus are linear. We also execute aggregation trees twice and (un)shuffle [PRRS24], which are both $O(n)$. The remaining primitives are straightforward. All building blocks are $O(n)$, hence Π-SquareRootMerge is $O(n)$.

5 Logstar: Our Bandwidth-optimized Construction

We are now ready to formally present Π-Logstar. In Section 5.1, we present the version of Π-Logstar that runs in $O(n \cdot 2^{\log^* n})$ time and communication (as opposed to $O(n \log^* n)$) and uses $O(\log n)$ rounds. This protocol is displayed in Figure 8. We prove that the protocol is correct and secure in Section 5.2. In Section 5.3, we show how to optimize Π-Logstar so that it runs in $O(n \log^* n)$ time and communication and still uses $O(\log n)$ rounds. We note that the difference between the two versions of our protocols is simply removing dummies. Dummies can only be removed after the second iteration, however, our recursion will typically terminate at this point making the two protocols effectively the same. We include the dummy removal process mainly to show how our protocol works asymptotically, or for very large lists such as 2^{30}.

5.1 Π-Logstar

We now explain the protocol described in Figure 8. Recall that we want to compute permutation π that merges two sorted lists X, Y, where $|X| = |Y| = n$. The protocol is parameterized by a block size $m = \log n$ and a number of blocks $k = \frac{n}{\log n}$. Note that $m \cdot k = n$.

Our description of Π-Logstar in Figure 8 consists of 3 protocols: Π-Logstar, Π-LogstarRecursive, and Π-ComputeMedians. Π-Logstar is the top-level protocol. In step 4 it invokes Π-LogstarRecursive which in turn invokes Π-ComputeMedians

```
Π-Logstar Protocol

INPUT: Secret-shared lists $[\![X]\!]$, $[\![Y]\!]$, s.t. $|X| = |Y| = n$. Let $k := \frac{n}{\log n}$, $m := \log n$.
OUTPUT: Secret-shared permutation $[\![\pi]\!]$, s.t. $\pi(X||Y)$ is sorted.

Π-Logstar($[\![X]\!], [\![Y]\!]$) :

1. $[\![X.\mathsf{ListId}]\!] := 0^n, [\![Y.\mathsf{ListId}]\!] := 1^n$
2. $[\![X.\mathsf{IsReal}]\!] := 1^n, [\![Y.\mathsf{IsReal}]\!] := 1^n$
3. $[\![X.\mathsf{Idx}]\!] := [n], [\![Y.\mathsf{Idx}]\!] := [n] + n$
4. $[\![I]\!] :=$ Π-LogstarRecursive($[\![X]\!], [\![Y]\!]$)
5. return Π-ExtractOrdered($[\![I]\!]$).Idx

Π-LogstarRecursive($[\![X]\!], [\![Y]\!]$) :

1. if $n \leq 10$ :
   a. $[\![\pi]\!] :=$ Π-MergeInv($[\![X]\!], [\![Y]\!]$)
   b. return Π-PermuteInv($[\![X.(\mathsf{Idx}, \mathsf{IsReal})]\!] || [\![Y.(\mathsf{Idx}, \mathsf{IsReal})]\!], [\![\pi]\!]$)
2. else :
   a. $[\![X']\!] :=$ Π-ComputeMedians($[\![X]\!]$), $[\![Y']\!] :=$ Π-ComputeMedians($[\![Y]\!]$)
   b. $[\![\pi]\!] :=$ Π-MergeInv($[\![X']\!], [\![Y']\!]$)
   c. parallel-for $i \in [k] : b := im, e := b + m,\ [\![B_i]\!] := [\![X_{[b,e]}]\!],\ [\![B_{i+k}]\!] := [\![Y_{[b,e]}]\!]$
   d. parallel-for $i \in [k] : [\![B_{i,0}.\mathsf{value}]\!] := [\![X'_i]\!], [\![B_{i+k,0}.\mathsf{value}]\!] := [\![Y'_i]\!]$
   e. $[\![B]\!] :=$ Π-Permute($[\![B]\!], [\![\pi]\!]$)
   f. $[\![c_0]\!] := 0, [\![c_i]\!] := \neg([\![B_i.\mathsf{ListId}]\!] \oplus [\![B_{i-1}.\mathsf{ListId}]\!]),\ i := [1, 2k]$
   g. $[\![S]\!] :=$ Π-AggregationTree($[\![B]\!] \gg 1, [\![c]\!]$, prefix)
   h. parallel-for $i \in [2k], j \in [m]$:
      i. $[\![B_{i,j}.\mathsf{IsReal}]\!] := [\![B_{i,j}.\mathsf{IsReal}]\!] \wedge ([\![B_{i,j}]\!] < [\![B_{i+1,0}]\!])$
      ii. $[\![S_{i,j}.\mathsf{IsReal}]\!] := [\![S_{i,j}.\mathsf{IsReal}]\!] \wedge ([\![S_{i,j}]\!] \geq [\![B_{i,0}]\!])\ \wedge\ ([\![S_{i,j}]\!] \leq [\![B_{i+1,0}]\!])$
   i. parallel-for $i \in [2k] : [\![I_i]\!] :=$ Π-LogstarRecursive($[\![S_i]\!], [\![B_i]\!]$)
   j. return $||_{i \in [2k]} [\![I_i]\!]$

Π-ComputeMedians($[\![X]\!]$) :

1. parallel-for $i \in [k]$:
   a. $b := im$
   b. $[\![f_{i,0}]\!] := [\![X_b.\mathsf{IsReal}]\!]$
   c. parallel-for $j \in [1, m] : [\![f_{i,j}]\!] := [\![X_{b+j}.\mathsf{IsReal}]\!] \wedge \neg[\![X_{b+j-1}.\mathsf{IsReal}]\!]$
   d. $[\![Z_i]\!] := 0$
   e. parallel-for $j \in [m] : [\![Z_i]\!] := [\![Z_i]\!] + [\![X_{b+j}]\!] \cdot [\![f_{i,j}]\!]$
2. return $[\![Z]\!]$
```

Fig. 8: Π-Logstar is the key protocol of our approach. It initializes parameters for its subprotocol Π-LogstarRecursive, which recursively computes the secure merge. We also define Π-ComputeMedians, a subprotocol of Π-LogstarRecursive.

(step 2a). Π-ComputeMedians is similar to ComputeMedians but is interactive. It computes k medians of a list when each m-sized block of the list starts with possibly multiple dummy elements. The median in each block is the first non-dummy element. Π-LogstarRecursive recursively computes a permutation that would merge the input lists. When Π-LogstarRecursive returns, its output is of size $O(n \cdot 2^{\log^* n})$.[3] It consists of the *indices* of the merged output of size $2n$ (i.e., the permutation π such that $\pi(X||Y)$ is merged); the rest are dummies interspersed between the merged output indices. Π-Logstar then extracts *in order* the merged indices, which maintains the ordering of the returned indices and removes all dummies, and outputs the result (step 5). Steps 1-3 initialize parameters for Π-LogstarRecursive. More specifically, step 1 saves from which list each element of X and Y come. Note that this is obvious at the start but will become important in the recursive procedure when we reorder blocks of X, Y according to medians. Step 2 sets bitvectors X.IsReal, Y.IsReal to all 1s to indicate the elements of the input lists X, Y are not dummies (0 will be used to indicate dummies when they are inserted later in the protocol). These bitvectors will be updated as we insert dummies in X and Y at each recursive step. They will enable us to obliviously remove all dummies once the recursive function returns control to Π-Logstar. In step 3, we save the initial indices X.Idx and Y.Idx. This will enable us to return a permutation when we merge the lists. We then pass the outputs of steps 1-3 as arguments to Π-LogstarRecursive.

Π-LogstarRecursive is more complicated. While theoretically we recurse a total of $O(\log^* n)$ times so that the last-level subproblems have constant size, in practice we execute the recursive procedures 2-3 times (for any array size) before reaching the base case (step 1) and merging with a standard protocol Π-MergeInv (step 1a), e.g. Batcher's network evaluated with GMW. Recall from Section 3.4 that Π-MergeInv returns the inverse permutation that would merge the current subproblem. We use this inverse permutation to reorder the indices and dummy bits in the current subproblem and return the result to the recursive caller (step 1b). The recursive step 2 first splits the merge into subproblems that can be recursively merged. The steps to achieve that correspond to steps 1-4 from Section 4.1. It then concatenates the outputs of all solved subproblems and returns the result to Π-Logstar (step 2j), which computes the final permutation. Note that this step corresponds to step 5 in the high-level description of Section 4.1, which concatenates the outputs from the base case. Step 5 is straightforward. We now formally explain how we implement the high-level steps 1-4:

1. **[Block Merge]** We first compute k medians $[\![X']\!]$, $[\![Y']\!]$ of $[\![X]\!]$, $[\![Y]\!]$ (step 2a). Note that this requires a secure protocol as we need to ensure the medians are not dummies. We compute a permutation π that would merge the medians by first calling Π-MergeInv (step 2b). We then split X and Y into k m-sized blocks (steps 2c), set the value of the first element of each block to equal the median (step 2d), and then permute the blocks with ComputePermutation according to π (step 2e).

[3] Each of the $O(\log^* n)$ recursive steps doubles the size of the output; merging doubles the size once more.

2. **[Duplicate]** We now duplicate transition blocks onto the next streak of blocks via aggregation trees. We first compute $2k$ control bits c, one per block, which indicate the start of a new streak of blocks (step 2f). Observe that $c = 0$ only when the corresponding neighboring blocks are from different lists. We can now invoke an aggregation tree with c and B (step 2g) and get S as output. S consists of $2k$ m-sized blocks, i.e., there is one block S_i for each B_i.
3. **[Extract]** Now, we would like to recursively merge each B_i with each S_i. The issue is that we duplicated some blocks multiple times. Hence, we need to ensure that each element is marked as non-dummy only once so that we extract each element only once after the recursive calls complete. We do that by ensuring that each $B_{i,j}.\mathsf{IsReal}$ and $S_{i,j}.\mathsf{IsReal}$ are 1 if and only if (1) they were 1 to start with, and (2) $B_{i,j}$ and $S_{i,j}$ are in the range of the median of the current block and the median of the next block (steps 2(2h)i-2(2h)ii). Note that in this step we use the result of step 2d, where we set the first value of each block to the non-dummy median.
4. **[Recurse]** Now we are ready to recursively merge B_i with S_i (step 2i), combine the subproblem results (step 2j), and return control to Π-Logstar.

Π-ComputeMedians. The remaining step is to describe how Π-ComputeMedians works (invoked in step 2a of Π-LogstarRecursive). Note that we cannot use the straightforward local ComputeMedians as the inputs can contain dummies at the beginning and end of each m-sized block. Π-ComputeMedians finds in each block the first non-dummy element and outputs it.

The protocol takes as input a secret-shared list X. It computes its k medians in step 1 and outputs them in step 2. We now look at step 1 in detail. Note the medians are computed in parallel. In step 1a, we compute the beginning index b of each block. Then we compute a one-hot vector f indicating which element of a block i contains the median. Recall it is the first non-dummy element. We start by setting $f_{i,0} := X_{im}.\mathsf{IsReal}$ (step 1b) and then compute the rest of f with simple ANDs. We want $f_{i,j} = 1$ only when the previous X_{b+j-1} is a dummy and the current X_{b+j} is real (step 1c). We then extract the element with $f_{i,j} = 1$. We take a dot product of the block starting at X_b and the bitvector f (step 1e) and add it into variable Z_i initialized to 0 (step 1d). Z_i represents the median. After computing Z_i for all k blocks, we return it (step 2).

This protocol relies on the fact that X only has at most a single contiguous block of non-dummy elements. This is true when we first invoke this protocol as X is the original list without any dummy elements. However, this invariant is maintained in all future calls. This is because when we mark parts of a block as dummy elements, they are either in the beginning, or in the end, but never from the middle. This preserves our invariant. After marking the dummy elements, we divide a block into further sub-blocks for the next recursive call, and it is easy to see that the sub-blocks also have this property.

5.2 Π-Logstar Proofs

Theorem 1 (Π-Logstar **correctness**). Π-Logstar *realizes the* $\mathcal{F}_{\mathsf{merge}}$ *functionality when* $|X| = |Y| = n$.

Proof. Correctness can be verified by inspection via the logic in Section 4.1. □

Theorem 2 (Π-Logstar **security**). Π-Logstar *is secure against semi-honest adversaries in the* ***sf****-hybrid model, where* ***sf*** *is the list of functionalities invoked by* Π-Logstar *(*MergeInv*,* PermuteInv*,* ComputeMedians*,* Permute*,* AggregationTree*,* ExtractOrdered*, and* MPC*).*

Proof. Simulation follows exactly that of Section 3.5, where the circuit consists of MergeInv, PermuteInv, ComputeMedians, Permute, AggregationTree, ExtractOrdered, and MPC gates. Hence, Π-Logstar is simulatable. □

5.3 Optimizing Π-Logstar: Getting to $O(n \log^* n)$

We now describe how to optimize Π-Logstar to get time and communication down from $O(n \cdot 2^{\log^* n})$ to $O(n \log^* n)$ while keeping the round complexity at $O(\log n)$. As described in Section 4.1, we need to prevent the list size from doubling every recursive call while still performing roughly linear work in every recursive step. The key observation in this regard is the following. Recall that for each block B_i, we create a duplicated block S_i that contains the potential strays that go into it from previous blocks. See the example from Section 4.1 (Figures 5 and 6). We looked at six blocks $B_0, \ldots, B_5$, where B_0 and B_5 were from X and $B_1, \ldots, B_4$ were from Y. In this case, B_0 potentially strays into $B_1, \ldots, B_4$, and B_4 potentially strays into B_5. So, we set $S_1 = \ldots = S_4 = B_0$ and $S_5 = B_4$. Then, we mark parts of S_is that aren't actual strays for B_i as dummies (and we also mark parts of B_i that do stray as dummies). Indeed, if we look at $S_1, \ldots, S_4$, the non-dummy elements are simply a subset of B_0. Importantly, each element of B_0 occurs as a non-dummy exactly once, either in B_0, or in one of $S_1, \ldots, S_4$. Thus, even though we have doubled the size of the list, we are only using every element once. This may seem obvious, and it is. And we also cannot leverage this observation immediately to cull down the sizes of B_is or S_is as there should be many dummies across them not to reveal information about X and Y.

It seems like we may be dead in the water, but the observation comes by peeking into what happens in the next recursive call. Indeed, since a lot (precisely half) of the B_is and S_is are dummies, this means that when we recurse on them, divide them up into smaller subproblems, many of those subproblems will have a trivial solution because one of the lists in the subproblem will entirely be dummies. So our approach will be to construct the subproblems for the next level of the recursion, filter some of them out, and then recurse on the rest. What we will show is that when we do this, our effective list size will still grow, but not quite double every time. In fact, it grows by a factor of $1+o(1)$ every time. Now, since the number of recursive calls we need is $\log^* n$, the effective size of the list at the every end, which also turns out to be the time and communication complexity of our protocol, will be $O(n \log^* n)$.

Consider the first recursive step of our protocol. We divide the lists X and Y of length n into a total of $2 \cdot \frac{n}{\log n}$ blocks B_i of size $\log n$. For each of these $2 \cdot \frac{n}{\log n}$ blocks, we construct a duplicated block S_i of size $\log n$. Next, we will look to recursively merge each B_i and S_i. When we do that, we divide each B_i and S_i to get a total of $2 \cdot \frac{\log n}{\log \log n}$ blocks $B'_{i,i'}$ (say) of size $\log \log n$. Thus across all i, we have a total of $2 \cdot \frac{n}{\log n} \cdot 2 \cdot \frac{\log n}{\log \log n} = 4 \cdot \frac{n}{\log \log n}$ blocks of size $\log \log n$. We know that across all of these blocks, we have exactly one copy of each of the elements of X and Y. Thus, the total number of non-dummy elements in these $4 \cdot \frac{n}{\log \log n}$ blocks of size $\log \log n$ is exactly $2n$. Clearly, all of the blocks cannot be completely filled with non-dummy elements. As expected, the blocks have a total of $4 \cdot \frac{n}{\log \log n} \cdot \log \log n = 4n$ elements, but only contain half, i.e., $2n$ non-dummy elements. The question we are now faced with is the following: How many of the $4 \cdot \frac{n}{\log \log n}$ blocks contain any non-dummy elements (more importantly, how many of them are completely filled with dummy elements)?

Trivially, all we can say is that they could all have some non-dummy elements, but we can in fact do better. Suppose we add an extra step where before dividing B_i and S_i into blocks of size $\log \log n$, we rearrange the elements in each B_i and S_i such that all the non-dummy elements in each of them appear right at the beginning and all the dummy elements appear at the end.[4] Note that this can be done in linear time and communication and $O(1)$ rounds using Π-ExtractOrdered. Once we do this, when we divide B_i and S_i into blocks of size $\log \log n$, they each potentially contribute some blocks completely filled with non-dummy elements, at most one block partially filled with non-dummy elements, and then some blocks completely filled with dummy elements. We can thus say that:

- at most $\frac{2n}{\log \log n}$ blocks are completely filled with non-dummy elements;
- at most $\frac{2n}{\log n}$ blocks are partially filled with non-dummy elements; and hence,
- at most $2n \left(\frac{1}{\log n} + \frac{1}{\log \log n} \right)$ blocks (among the $4 \cdot \frac{n}{\log \log n}$ blocks) contain non-dummy elements.

This means that of the $4 \cdot \frac{n}{\log \log n}$ subproblems we would have recursed on, we only need to recurse on at most $2n \left(\frac{1}{\log n} + \frac{1}{\log \log n} \right)$ of them. Observe that in linear time, we can extract $2n \left(\frac{1}{\log n} + \frac{1}{\log \log n} \right)$ such subproblems that will contain all of the non-dummy elements using linear time and communication and $O(1)$ rounds using Π-ExtractOrderedPad.

Thus, using a fixed linear (in n) amount of time and communication, we are able to turn $\frac{2n}{\log n}$ subproblems on lists of size $\log n$ into $2n \left(\frac{1}{\log n} + \frac{1}{\log \log n} \right)$ subproblems on lists of size $\log \log n$. If we continue for another recursive step,

[4] One can actually observe that blocks already possess the property that dummies are never interspersed to avoid having to rearrange the elements in the blocks and lose a factor of 2 elsewhere.

we will see that we will turn them into $2n\left(\frac{1}{\log n}+\frac{1}{\log\log n}+\frac{1}{\log\log\log n}\right)$ subproblems on lists of size $\log\log\log n$. In each transformation, we incur time and communication that is a fixed linear function of the effective length of the list at the point. After the first transformation, we have $\frac{2n}{\log n}$ subproblems on lists of size $\log n$ resulting in an effective list size of $2\cdot\frac{2n}{\log n}\cdot\log n = 4n$. After the second transformation, we have $2n\left(\frac{1}{\log n}+\frac{1}{\log\log n}\right)$ subproblems on lists of size $\log\log n$ resulting in an effective list size of $2\cdot 2n\left(\frac{1}{\log n}+\frac{1}{\log\log n}\right)\cdot\log\log n = 4n\left(1+\frac{\log\log n}{\log n}\right)$. After the third transformation, we would have an effective list size of $4n\left(1+\frac{\log\log\log n}{\log\log n}+\frac{\log\log n}{\log n}\right)$. Thus, the effective size of the list only grows by a factor of $1+o(1)$ every recursive step.

As we previously had, we only need $\log^* n$ recursive steps until the blocks of size $O(1)$. Since the transformations introduce only an additional $O(1)$ rounds in each recursive step, the asymptotic round complexity of this optimized version of Π-Logstar is the same as before, i.e., $O(\log n)$.

Let us now bound the time and communication of this optimized version of Π-Logstar. The total time and communication complexity of all the transformations is

$$O\left(n+n\left(1+\frac{\log\log n}{\log n}\right)+n\left(1+\frac{\log\log\log n}{\log\log n}+\frac{\log\log n}{\log n}\right)+\ldots\right)$$

Each of the terms above can be bounded by $O(n)$ yielding a total of $O(n\log^* n)$. To see why, consider the ith term, for $i\geq 2$. It is

$$n\left(1+\sum_{j=1}^{i-1}\frac{\log^{(j+1)} n}{\log^{(j)} n}\right)$$

where $\log^{(\cdot)}$ denotes iterated logarithms. The term only shows up if $\log^{(i)} n\geq 1$. This means that $\log^{(i-1)} n\geq 2$, $\log^{(i-2)} n\geq 2^2$, $\log^{(i-3)} n\geq 2^{2^2}$, and so on. Furthermore, each of the fractions $\frac{\log^{(j+1)} n}{\log^{(j)} n}$ are decreasing functions. Therefore, their maximum in our range of consideration will be at most

$$\frac{1}{2}+\frac{2}{2^2}+\frac{2^2}{2^{2^2}}+\ldots<1+\frac{1}{2}+\frac{1}{4}+\ldots=2$$

So each individual term is bounded by $3n=O(n)$ as required.

6 Median: Our Rounds-optimized Construction

We now present Π-Median in formal detail. The key algorithm is presented in Figure 9. We explain our main protocol Π-Median in Section 6.1. Π-Median closely depends on Π-CubeRootMerge, our modified presentation of [BBD+22]'s protocol. We present the protocol in Section 6.2. In Section 6.3, we demonstrate that our approach aligns all blocks. We prove Π-Median (and Π-CubeRootMerge) correct and secure in Section 6.4.

6.1 Π-Median

```
Π-Median Protocol

INPUT: Secret-shared lists $[\![X]\!]$, $[\![Y]\!]$, s.t. $|X| = |Y| = n$. Let $k := n^{\frac{1}{3}}$, $m := \frac{n}{k}$.
OUTPUT: Secret-shared permutation $[\![\pi]\!]$, s.t. $\pi(X||Y)$ is sorted.

Π-Median($[\![X]\!], [\![Y]\!]$) :
  1. $[\![X.\mathsf{ListId}]\!] := 0^n, [\![Y.\mathsf{ListId}]\!] := 1^n$
  2. $[\![X.\mathsf{IsReal}]\!] := 1^n, [\![Y.\mathsf{IsReal}]\!] := 1^n$
  3. $[\![X.\mathsf{Idx}]\!] := [n], [\![Y.\mathsf{Idx}]\!] := [n] + n$
  4. $[\![I]\!] :=$ Π-MedianRecursive($[\![X]\!], [\![Y]\!]$)
  5. return Π-ExtractOrdered($[\![I]\!]$).Idx

Π-MedianRecursive($[\![X]\!], [\![Y]\!]$) :
  1. if $n \leq 10$ :
     a. $[\![\pi]\!] :=$ Π-MergeInv($[\![X]\!], [\![Y]\!]$)
     b. return Π-PermuteInv($[\![X.(\mathsf{Idx}, \mathsf{IsReal})]\!] || [\![Y.(\mathsf{Idx}, \mathsf{IsReal})]\!], [\![\pi]\!]$)
  2. else :
     a. $[\![Z]\!] :=$ Π-AlignLists($[\![X]\!], [\![Y]\!]$)
     b. $[\![Z']\!] :=$ Π-AlignLists($[\![Y]\!], [\![X]\!]$)
     c. parallel-for $i = 0, \ldots, 2k-1$ :
        i. $s := i \cdot m$, $e := s + m$
        ii. $[\![I_{[2s,2e]}]\!] :=$ Π-MedianRecursive($[\![Z]\!]_{[s,e]}, [\![Z']\!]_{[s,e]}$)
     d. return $[\![I]\!]$

Π-AlignLists($[\![X]\!], [\![Y]\!]$) :
  1. $[\![X']\!] :=$ ComputeMedians($[\![X]\!], k$)
  2. $[\![\pi]\!] :=$ Π-CubeRootMergeInv($[\![X']\!], [\![Y]\!]$)
  3. $[\![X']\!] :=$ DuplicateMedians($[\![X']\!], m$)
  4. $[\![X'.\mathsf{IsReal}]\!] := 0^n$
  5. $[\![\pi]\!] :=$ UpdateInvPermutation($[\![\pi]\!], k, n$)
  6. return Π-PermuteInv($[\![X']\!] || [\![Y]\!], [\![\pi]\!]$)
```

Fig. 9: Π-Median is the main protocol of our Median approach. As Π-Logstar, it initializes parameters for its subprotocol Π-MedianRecursive, which recursively computes secure merge.

We now explain our main protocol in Figure 9. It shares many similarities with Π-Logstar, and hence we highlight the differences. Similarly to Π-Logstar, Π-Median in Figure 9 consists of 3 protocols: Π-Median, Π-MedianRecursive, and Π-AlignLists. The main functions Π-Logstar and Π-Median are almost identical (but the recursive steps are completely different). We highlight their differences:

- We invoke Π-MedianRecursive instead of Π-LogstarRecursive in step 4.

- When Π-MedianRecursive returns, the output is of size $n \cdot 2^{O(\log\log n)}$. Note that in Π-LogstarRecursive, the output is of size $O(n \cdot 2^{\log^* n})$ because it requires only $\log^* n$ recursive calls instead of $O(\log\log n)$. The merged output is still of size $2n$.

Unlike in Logstar, the recursive Π-MedianRecursive is relatively simple. The base case in step 1 is identical, and hence we focus on the recursive step 2. This step splits merge into smaller subproblems and recursively merges them. The challenge is to align all subproblems obliviously, i.e., ensure they are of the same size. We designed Π-AlignLists for this purpose (steps 2a-2b). After invoking Π-AlignLists with $(\llbracket X \rrbracket, \llbracket Y \rrbracket)$ and then with $(\llbracket Y \rrbracket, \llbracket X \rrbracket)$, we get Z and Z' such that $|Z| = |Z'| = 2n$ and they are aligned, i.e., we can split them into $2k$ $\frac{n}{k}$-sized blocks and merge the blocks from one list with corresponding blocks from the other list. We do that in step 2c. As in Π-Logstar, we take care that we invoke all Π-MedianRecursive *in parallel* for all subproblems not to incur unnecessary rounds. We obtain a list of indices corresponding to the final permutation interspersed with dummies and return it to Π-Median (step 2d).

The key challenge of Π-Median is to align the subproblems in Π-AlignLists. Let X and Y (such that $|X| = |Y| = n$) be the inputs to Π-AlignLists. In step 1 we compute $k = n^{\frac{1}{3}}$ medians X' of X. Then we invoke Π-CubeRootMerge (see Section 6.2) on X' and Y (step 2) to obtain an inverse permutation that would merge X' and Y. In step 3, we duplicate each median of X $m = \frac{n}{k}$ times (s.t. $|X'| = n$). We set the IsReal bits associated with X to all zeros to indicate all dummies (step 4). Essentially, to align blocks of X and Y, we are merging X' with Y, and hence X' is already included in the merge as part of X. In other words, each element of X should be marked as non-dummy only once. We update the permutation that merges X' and Y to account for the duplicated medians (step 5) and finish up the merge by invoking Π-Permute, which orders X' and Y according to the updated permutation (step 6). We return its output.

6.2 Π-CubeRootMerge

Π-CubeRootMerge is closely related to [BBD⁺22]'s $(n^{\frac{1}{3}}, n)$-merge. While it incurs similar costs to the original protocol, it has been modified and expressed in our notation, with step 4 implemented differently. We show the protocol in Figure 10. In this protocol, the size of the sorted input lists is imbalanced. It receives as input a secret-shared list $\llbracket X \rrbracket$ of size $n^{\frac{1}{3}}$ and another secret-shared list $\llbracket Y \rrbracket$ of size n. As in Π-Logstar, it outputs a secret-shared permutation that merges X and Y. Π-CubeRootMerge is a key subprotocol of Π-Median. It is used to merge the $n^{\frac{1}{3}}$ medians of one list with another (and vice versa). Π-CubeRootMerge runs in $O(n)$ time and communication and $O(1)$ rounds. We first give a high level description of Π-CubeRootMerge, and then explain it in detail.

The fundamental idea is to invoke Π-AllPairsMergeInv twice. The first time, we merge X with $n^{\frac{2}{3}}$ medians of Y. As $|X| = n^{\frac{1}{3}}$, this takes linear (in n) time and lets us identify every block of Y (of size $n^{\frac{1}{3}}$) that contains an element of X in $X \bigsqcup Y$. $|X| = n^{\frac{1}{3}}$ also implies there can be at most $n^{\frac{1}{3}}$ such blocks. We

Π-CubeRootMerge **Protocol**

Input: Secret-shared lists $[\![X]\!]$ and $[\![Y]\!]$ such that $|X| = m = n^{\frac{1}{3}}$ and $|Y| = n$.
Output: Secret-shared permutation $[\![\sigma]\!]$ such that $\sigma(X||Y)$ is merged.

```
Π-CubeRootMerge($[\![X]\!], [\![Y]\!]$) :

 1. $[\![\sigma]\!] :=$ Π-CubeRootMergeInv($[\![X]\!], [\![Y]\!]$)
 2. return Π-Inv($[\![\sigma]\!]$)

Π-CubeRootMergeInv($[\![X]\!], [\![Y]\!]$) :

 1. Find (up to $m$) blocks of $Y$ that have elements of $X$ in them:
    a. $k := n^{\frac{2}{3}}$
    b. $[\![Y']\!] :=$ ComputeMedians($[\![Y]\!], k$)
    c. $[\![\pi]\!] :=$ Π-AllPairsMergeInv($[\![X]\!], [\![Y']\!]$)
    d. $[\![Y'_i.\mathsf{IsExtracted}]\!] := [\![\pi_{m+i+1}]\!] > ([\![\pi_{m+i}]\!] + 1), \forall i \in [k-1]$
    e. $[\![Y'_{k-1}.\mathsf{IsExtracted}]\!] := ([\![\pi_{m+k-1}]\!] \neq m + k - 1)$
 2. Extract (up to $m$) blocks of $Y$ that any $X_i$ will merge with:
    a. for $i \in [k]$ : $[\![B'_i]\!] := [\![Y_{[im,(i+1)m]}]\!]$
    b. for $i \in [k]$ : $[\![B'_i.\mathsf{idx}]\!] := i$, $[\![B'_i.\mathsf{IsReal}]\!] := [\![Y'_i.\mathsf{IsExtracted}]\!]$
    c. $([\![B]\!], [\![B.\mathsf{isReal}]\!], [\![\theta]\!]) :=$ Π-ExtractOrderedPad($[\![B']\!], m$)
    d. $[\![Y'']\!] := ([\![B_0]\!] || \ldots || [\![B_{m-1}]\!])$
 3. Get the merging permutation for the extracted $Y$ and the $X$:
    a. $[\![\rho]\!] :=$ Π-AllPairsMergeInv($[\![X]\!], [\![Y'']\!], [\![1^m]\!], [\![Y''.\mathsf{IsReal}]\!]$)
 4. Update $\rho$ to count the missing blocks of $Y$. In parallel do:
    a. Update the positions of $\rho$ for $X$ to count the missing blocks of $Y$:
        i. $[\![Y''.\mathsf{Jump}]\!] := 0^k$
       ii. $[\![Y''_0.\mathsf{Jump}]\!] := [\![B_0.\mathsf{Idx}]\!] \cdot m$
      iii. for $i \in [1, m]$ : $[\![Y''_{im}.\mathsf{Jump}]\!] := ([\![B_i.\mathsf{Idx}]\!] - [\![B_{i-1}.\mathsf{Idx}]\!] - 1) \cdot m$
       iv. $[\![t]\!] :=$ Π-PermuteInv($0^m || [\![Y''.\mathsf{Jump}]\!], [\![\rho]\!]$)
        v. for $i \in [1, k+m]$ : $[\![t_i]\!] := [\![t_i]\!] + [\![t_{i-1}]\!]$
       vi. $[\![d]\!] :=$ Π-Permute($[\![t]\!], [\![\rho]\!]$)
      vii. for $i \in [m]$ : $[\![\sigma_i]\!] := [\![\rho_i]\!] + [\![d_i]\!]$
    b. Count how many $X_i$ came before each $Y''_j$ and map that back to $Y$:
        i. for $i \in [k]$ : $[\![c_i]\!] := [\![\rho_{m+i}]\!] - i$
       ii. for $i \in [1, k]$ : $[\![d_i]\!] := [\![c_i]\!] - [\![c_{i-1}]\!], [\![d_0]\!] := [\![c_0]\!]$
      iii. for $i \in [m]$ : $[\![\beta_i]\!] := [\![d_{[im,im+m]}]\!]$
       iv. $[\![\beta']\!] :=$ Π-UnextractOrdered($[\![\beta]\!], [\![\theta]\!]$)
        v. for $i \in [k]$ : $[\![d'_{[im,(i+1)m]}]\!] := [\![\beta'_i]\!]$
       vi. $[\![\sigma_m]\!] := [\![d_0]\!]$
      vii. for $i \in [1, n]$ : $[\![\sigma_{m+i}]\!] := [\![\sigma_{m+i-1}]\!] + [\![d'_i]\!] + 1$
 5. return $[\![\sigma]\!]$
```

Fig. 10: Π-CubeRootMerge implements secure merge when the input lists $|X| = m$, $|Y| = n$. It runs in $O(n)$ time and communication and $O(1)$ rounds.

next securely extract these blocks of Y (to get $n^{\frac{2}{3}}$ elements after potentially padding) and merge them again with X. This allows us to identify the positions of elements of X in the extracted blocks of Y. With some non-trivial index accounting, we can then compute the final (inverse) permutation.

Note that Figure 10 consists of 2 protocols. Π-CubeRootMerge is the main merge protocol; Π-CubeRootMergeInv is its subprotocol that computes the inverse permutation of the merge. Π-CubeRootMerge simply invokes Π-CubeRootMergeInv (step 1), inverts the resulting permutation and outputs it (step 2). The bulk of work is done in Π-CubeRootMergeInv. We now explain Π-CubeRootMergeInv step by step (the steps correspond to those in Figure 10):

1. **[Find blocks of Y that have elements of X in them]** We first set $k := n^{\frac{2}{3}}$ (step 1a) and compute the k medians of Y (step 1b). The medians split Y into $m = n^{\frac{1}{3}}$-size blocks. We now determine in which of these blocks elements of X would merge. Note that as $|X| = n^{\frac{1}{3}}$, there can be at most $n^{\frac{1}{3}}$ of them. We invoke Π-AllPairsMergeInv on X and the k medians of Y (step 1c). Note that this step is $O(n)$ as $m \cdot k = n^{\frac{1}{3}} \cdot n^{\frac{2}{3}} = n$. Recall Π-AllPairsMergeInv returns the inverse permutation π that would merge X with the medians of Y. We are now ready to compute which blocks of Y contain elements from X. In step 1d we simply look at all neighboring pairs of entries of π corresponding to Y (i.e., $\pi_{m+i+1} - \pi_{m+i}$) and check if their difference is greater than one. In step 1e, we then compute the edge case at $k - 1$.
2. **[Extract blocks found in 1]** In this step, we first retrieve all blocks of Y (step 2a) and save their initial position alongside each block (step 2b). The initial position is not used at this step, but will be necessary in later steps to account for blocks that are not extracted. We now invoke Π-ExtractOrderedPad to extract all blocks of Y with elements of X in between in the order they appear (step 2c). We use the bits computed in the previous step to decide which blocks should be extracted (step 2b). We use the padded version of Π-ExtractOrdered as there can be at most m such blocks, but their exact number is unknown. The output contains the extracted blocks, a bit indicating if the extracted block is a non-dummy, and also a permutation θ. θ enables to unextract the blocks and is later used to compute final permutation. We save the extracted blocks in Y'' (step 2d).
3. **[Get the merging permutation for the extracted Y'' and X]** We now merge X with the extracted blocks Y'' from the previous step. This step can be executed in linear work and constant rounds by Π-AllPairsMergeInv (step 3a). We take care to use the version of Π-AllPairsMergeInv that merges the input lists and places all dummies at the end. This property is necessary for step 4. The output is an inverse permutation ρ.
4. **[Update the permutation to count missing blocks of Y]** We will now update the inverse permutation ρ such that it accounts for all blocks of Y (not just those extracted), and set the result to σ. We will first do it in step 4a for X (i.e., $\rho_{[m]}$) and only then in step 4b for Y (i.e., $\rho_{[m,m+n]}$). In step 5 we output the final σ.

(a) **[Update the permutation for X]** In steps 4(4a)i-4(4a)iii, we compute a vector Jump. Jump is zero at all positions in $[k]$ (step 4(4a)i) but at positions im, for $i \in [m]$. At these steps, Jump represents the number of unextracted block elements of Y in between each pair of extracted blocks (steps 4(4a)ii-4(4a)iii). In step 4(4a)iv, we prepend Jump with 0^m (one zero for each X_i) and permute the result according to ρ. After computing the prefix sum of the permuted result (step 4(4a)v), the output represents the offsets in $X \bigsqcup Y''$ due to the missing blocks of Y. We now permute in the reverse direction (step 4(4a)vi). In the first m elements of the result, we hold the offsets due to the missing blocks of Y for X. We add them to the current terms of $\rho_{[m]}$, and set the result to σ (step 4(4a)vii).

(b) **[Update the permutation for Y]** We first compute c_i, which represents the number of X_i before the extracted Y''_j (step 4(4b)i). Next, we compute d, which corresponds to the number of X_i between two consecutive elements of Y'' (step 4(4b)ii). We now need to extrapolate and compute the number of X_i before each element of Y. We first split d into m-sized blocks (step 4(4b)iii) and unextract them into their original positions in Y, the rest being zeros (step 4(4b)iv). Recall we computed θ in step 2c that allows us to do this. We flatten the result consisting of k m-sized blocks into a n-size list d' (step 4(4b)v). Note that d' now represents a vector which gives the number of X_i between all consecutive elements of Y. Thus, we can compute $\sigma_{[m,m+n]}$ with a simple prefix sum. In step 4(4b)vi, we set the first σ_m; in step 4(4b)vii, we set the remaining $\sigma_{[m+1,m+n]}$.

6.3 Block Alignment

Recall from Section 4.2 that [Val75]'s plaintext merge works by partitioning one list of size n based on the k medians of another list. This splits the full merge into k subproblems. This approach does not easily translate into a secure protocol as we cannot leak the sizes of the subproblems. In other words, we cannot leak the number of elements of one list lying between two medians of another list. We instead need to somehow *align* the subproblems so that their sizes are equal.

We do that, as introduced in [BBD$^+$22], by producing from the input lists of length n two expanded lists of length $2n$. In particular, we take the k medians from one list and merge $\frac{n}{k}$ copies of them into the other list. Lemma 1 of [BBD$^+$22] shows that these expanded lists are aligned, i.e., we can partition them into blocks of length $\frac{n}{k}$, merge the blocks separately, and then concatenate the outputs. Note that Lemma 1 is a rephrased and proven lemma from [BBD$^+$22], and hence we omit its proof. We emphasize that while we crucially rely on [BBD$^+$22]'s lemma, our protocol is significantly different. In particular, ours uses a relatively simple recursive structure on blocks of size $m = n^{2/3}$ while [BBD$^+$22] uses an extremely small blocks of size $m = \log\log n$ and a complex set of subprotocols to merge $k = n/\log\log n$ medians with a size n list in $O(n)$ time.

Lemma 1 **([BBD⁺22]).** *Let X and Y be two sorted lists such that $|X| = |Y| = n$. Let $X' :=$ $\mathsf{ComputeMedians}(X, k)$ denote the k medians of X. Then let $X' :=$ $\mathsf{DuplicateMedians}(X', \frac{n}{k})$ denote the list X' of size n after duplicating each median $\frac{n}{k}$ times. Let $X' \bigsqcup Y$ denote a list of size $2n$ after merging X' and Y. Similarly, compute $X \bigsqcup Y'$. The $2k$ medians of $X' \bigsqcup Y$ and $X \bigsqcup Y'$ are the k medians of X and the k medians of Y:*

$$\begin{aligned}\mathsf{ComputeMedians}(X' \bigsqcup Y, 2k) &= \mathsf{ComputeMedians}(X \bigsqcup Y', 2k)\\ &= \mathsf{ComputeMedians}(X, k) \bigsqcup \mathsf{ComputeMedians}(Y, k)\end{aligned}$$

6.4 Π-CubeRootMerge and Π-Median Proofs

We now prove Π-CubeRootMerge and Π-Median correct and secure. Similarly to Π-Logstar, the proofs are trivial for both protocols. Correctness can be verified by inspection. Simulation security stems from a simple composition argument as described in Section 3.5.

Theorem 3 **(Π-CubeRootMerge correctness).** Π-CubeRootMerge *realizes the $\mathcal{F}_{\mathsf{merge}}$ functionality when $|X| = n^{\frac{1}{3}}$ and $|Y| = n$.*

Proof. Correctness can be verified by inspection in conjunction with the description in Section 6.2. □

Theorem 4 **(Π-CubeRootMerge security).** Π-CubeRootMerge *is secure against semi-honest adversaries in the **sf**-hybrid model, where **sf** is the list of functionalities invoked by* Π-CubeRootMerge *(*AllPairsMergeInv*,* ExtractOrderedPad*,* UnextractOrdered*,* Permute*,* PermuteInv*, and* MPC*).*

Proof. Simulation follows exactly that of Section 3.5, where the circuit consists of AllPairsMergeInv, ExtractOrderedPad, UnextractOrdered, Permute, PermuteInv, and MPC gates. Hence, Π-CubeRootMerge is simulatable. □

Theorem 5 **(Π-Median correctness).** Π-Median *realizes the $\mathcal{F}_{\mathsf{merge}}$ functionality when $|X| = |Y| = n$.*

Proof. Correctness can be verified by inspection via the description in Section 4.2. □

Theorem 6 **(Π-Median security).** Π-Median *is secure against semi-honest adversaries in the **sf**-hybrid model, where **sf** is the list of functionalities invoked by* Π-Median *(*MergeInv*,* PermuteInv*,* CubeRootMergeInv*,* ExtractOrdered*, and* MPC*).*

Proof. Simulation follows exactly that of Section 3.5, where the circuit consists of MergeInv, PermuteInv, CubeRootMergeInv, ExtractOrdered, and MPC gates. Hence, Π-Median is simulatable. □

Π-SquareRootMerge **Protocol**

INPUT: Secret-shared lists $\llbracket X \rrbracket$ and $\llbracket Y \rrbracket$ such that $|X| = k = n^{\frac{1}{2}}$ and $|Y| = n$.
OUTPUT: Secret-shared permutation $\llbracket \pi \rrbracket$ such that $\pi(X||Y)$ is merged.

```
Π-SquareRootMerge($\llbracket X \rrbracket, \llbracket Y \rrbracket$) :

1. Find the blocks of $Y$ into which elements of $X$ go:
   a. $\llbracket Y' \rrbracket := \mathsf{ComputeMedians}(\llbracket Y \rrbracket, k)$
   b. $\llbracket X_i.\mathsf{lessThanY}'_j \rrbracket := (\llbracket X_i \rrbracket < \llbracket Y'_j \rrbracket)$, $\forall i, j \in [k]$
   c. $\llbracket X_i.\mathsf{mapsToY}'_{j-1} \rrbracket := \llbracket X_i.\mathsf{lessThanY}'_{j-1} \rrbracket \oplus \llbracket X_i.\mathsf{lessThanY}'_j \rrbracket$, $\forall i \in [k], j \in [1, k]$
   d. $\llbracket X_0.\mathsf{first} \rrbracket := 1$, $\llbracket X_i.\mathsf{first} \rrbracket := \bigoplus_{j \in [k]} (\llbracket X_i.\mathsf{mapsToY}'_j \rrbracket \wedge \neg \llbracket X_{i-1}.\mathsf{mapsToY}'_j \rrbracket)$,$\forall i \in [1, k]$
2. Prepare to extract blocks of $Y$ for each $X_i$:
   a. parallel-for $i \in [k]$ :
        i. $\llbracket Y'_i \rrbracket := \llbracket Y_{[ik, ik+k]} \rrbracket$
       ii. $\llbracket Y'_i.\mathsf{BlockIdx} \rrbracket := i$
      iii. $\llbracket D_i \rrbracket := 0^k$
       iv. $\llbracket Y'_i.\mathsf{XIdx} \rrbracket := \bigoplus_{j \in [k]} (j + 1) \cdot (\llbracket X_j.\mathsf{first} \rrbracket \wedge \llbracket X_j.\mathsf{mapsToY}'_i \rrbracket)$
        v. $\llbracket D_i.\mathsf{XIdx} \rrbracket := (i + 1) \cdot (\neg \llbracket X_i.\mathsf{first} \rrbracket)$
       vi. $\llbracket Y'_i.\mathsf{Ctrl} \rrbracket := 0$, $\llbracket D_i.\mathsf{Ctrl} \rrbracket := 1$
   b. $(\llbracket B' \rrbracket, \llbracket \theta \rrbracket) := \Pi\text{-}\mathsf{Shuffle}(\llbracket Y' \rrbracket || \llbracket D \rrbracket)$
   c. $t := \mathsf{open}(\llbracket B'.\mathsf{XIdx} \rrbracket)$
   d. for $i \in [2k]$, if $t_i \neq 0 : \llbracket B_{t_i - 1} \rrbracket := \llbracket B'_i \rrbracket$
3. Extract the blocks of $Y$: $\llbracket S \rrbracket := \Pi\text{-}\mathsf{AggregationTree}(\llbracket B \rrbracket, \llbracket B.\mathsf{Ctrl} \rrbracket, \mathsf{prefix})$
4. Compute final index of $X$:
   a. $\llbracket \ell_{i,j} \rrbracket := (\llbracket X_i \rrbracket < \llbracket S_{i,j} \rrbracket)$, $\forall i \in [k], j \in [k]$
   b. $\llbracket w_{i,j} \rrbracket := \llbracket \ell_{i,j} \rrbracket \oplus \llbracket \ell_{i,j-1} \rrbracket$, $\forall i \in [k], j \in [1, k]$
   c. for $i \in [k] : \llbracket X_i.\mathsf{Idx} \rrbracket := i + \llbracket S_i.\mathsf{BlockIdx} \rrbracket k + \bigoplus_{j \in [k]} j \llbracket w_{i,j} \rrbracket$
5. Compute final index of $Y$:
   a. $\llbracket w \rrbracket := \Pi\text{-}\mathsf{AggregationTree}(\llbracket w \rrbracket, \llbracket B.\mathsf{Ctrl} \rrbracket, \mathsf{suffix})$
   b. for $i \in [2k] : \llbracket w'_i \rrbracket := 0^k$
   c. for $i \in [2k]$, if $t_i \neq 0 : \llbracket w'_i \rrbracket := \llbracket w_{t_i - 1} \rrbracket$
   d. $(\llbracket w' \rrbracket || \llbracket D \rrbracket) := \Pi\text{-}\mathsf{Unshuffle}(\llbracket w' \rrbracket, \llbracket \theta \rrbracket)$
   e. for $i \in [k] : \llbracket Y_{[ik, ik+k]}.\mathsf{Idx} \rrbracket := \llbracket w'_i \rrbracket$
   f. for $i \in [1, n] : \llbracket Y_i.\mathsf{Idx} \rrbracket := \llbracket Y_i.\mathsf{Idx} \rrbracket + \llbracket Y_{i-1}.\mathsf{Idx} \rrbracket + 1$
6. return $\Pi\text{-}\mathsf{Inv}(\llbracket X.\mathsf{Idx} \rrbracket || \llbracket Y.\mathsf{Idx} \rrbracket)$
```

Fig. 11: Π-SquareRootMerge implements secure merge when the input lists $|X| = n^{\frac{1}{2}}$, $|Y| = n$. It runs in $O(n)$ time and communication and $O(\log n)$ rounds.

7 SquareRootMerge: Our Asymmetric $(n^{\frac{1}{2}}, n)$ Merge

We now present Π-SquareRootMerge in formal detail. Again, we assume familiarity with the high-level idea of Π-SquareRootMerge in Section 4.3. The key algorithm is presented in Figure 11.

This protocol is designed for the case of merging $k = \sqrt{n}$ and n-length lists. It runs in $O(n)$ time and communication and $O(\log n)$ rounds. We explain our main protocol Π-SquareRootMerge in Section 7.1, and prove our protocol correct and secure in Section 7.2.

7.1 Π-SquareRootMerge

Now we are ready to present Π-SquareRootMerge step by step:

1. **[Find the blocks of Y into which elements of X go]** We first find k medians Y' of Y (step 1a). These medians split Y into k same-sized blocks. We next compare each X_i with each median Y'_j (step 1b). This requires n secure comparisons and allows us to compute in which block of Y each X_i goes (step 1c). I.e., for each X_i, we compute a one-hot vector $\mathsf{mapsToY'}_j$ of size k that is non-zero only at the block j where X_i belongs. We use this bitvector to additionally compute, for all X_i, a bit $X_i.\mathsf{first}$. This bit indicates if X_i is the smallest element of X that goes to a block j in Y (step 1d). Both mapsToY' and first will be necessary in the following step that prepares the input for the aggregation trees.
2. **[Prepare to extract blocks of Y for each X_i]** To extract the blocks, we need to consider that some blocks may need to be extracted *more than once*. However, this needs to remain oblivious. Our approach works by extracting each block (belonging to some X_i) at most once and extracting a unique dummy block whenever one block is needed repeatedly. Then we replace the dummy blocks with the copied blocks (corresponding to the block substituted by the dummy) via an aggregation tree, which is done in the next step 3. Now, we show how to construct the aggregation tree inputs. We first split Y into k-size blocks Y' (step 2(2a)i) and save the initial block index (step 2(2a)ii). We will need the block index in later steps to account for the unextracted blocks in the final index calculation. Then we create k all-zero secret-shared dummy blocks D (step 2(2a)iii). We now mark Y' and D with an index $\mathsf{XIdx} \in [1, k+1]$ such that it indicates which block to retrieve for each X_i (steps 2(2a)iv-2(2a)v). We cleverly use the bits first and mapsTo from step 1 to mark a block Y'_j for X_i assuming (1) X_i belongs to Y'_j's block, and (2) X_i is the smallest element from X that goes into Y'_j's block. If (2) does not hold, we mark the ith dummy block D_i. We additionally mark all Y' blocks with 0 and all dummy D blocks with 1 (step 2(2a)vi). These bits are called control bits Ctrl and help the aggregation tree decide which blocks should be copied into dummies. Now that we have marked the blocks, we obliviously retrieve them by shuffling $Y'||D$ (step 2b) , opening the marks XIdx (step 2c), and selecting in order the blocks marked with $[1, k+1]$ (step 2d). Note that opening the marks is secure as we shuffled the blocks, and hence we cannot correlate the mark with a particular block. Note that the shuffle also returns a secret-shared permutation θ that will be used in step 5 to unshuffle and place the extracted blocks in their original position.

3. [**Extract the blocks of** Y] From previous step, we hold a list of k blocks B. Recall these blocks potentially hold many dummy blocks. In this step, we replace each dummy block B_i with the largest real block $B_{j<i}$. We use an aggregation tree protocol (step 3). We input the blocks B alongside the associated control bits $B.\mathsf{Ctrl}$ and receive k blocks S as output. I.e., we hold block S_i for each element X_i such that $S_{i,0} < X_i \leq S_{i,k-1}$.
4. [**Compute final index of** X] We are now ready to compute the final indices of all X_i in the merged list $X \bigsqcup Y$. We start by comparing each X_i with all k elements in its associated block S_i (step 4a). This requires n secure comparisons. We then compute a one-hot vector w_i, which is non-zero at the index where X_i goes in S_i (step 4b). Now we can compute the final index $X_i.\mathsf{Idx}$ (step 4c). This index can be viewed as a sum of 3 summands: (1) i, the number of elements in X before X_i, (2) $[\![S_i.\mathsf{BlockIdx}]\!]k$, the number of elements in all the blocks of Y preceding S_i, and (3) $\bigoplus_{j\in[k]} j[\![w_{i,j}]\!]$, the number of elements preceding X_i in the block S_i.
5. [**Compute final index of** Y] Recall this step is more complex than computing the final indices of X as we need to consolidate possibly multiple copies of blocks (i.e., when 2 or more X_i belong to the same block of Y). We do that with aggregation trees (step 5a). In the previous step 4b, we computed a one-hot vector w that indicates where X_i fits in S_i. We use w as input to the aggregation tree along with the same control bits as in step 3. This time we run a *suffix* aggregation tree. The output is a list of k blocks, w, which computes the number of X_i that fit between any 2 consecutive elements of Y. If multiple elements of X go into a single block of Y, the aggregation tree aggregates them into a block where $B_i.\mathsf{Ctrl} = 0$. We need to pull out these blocks with $B_i.\mathsf{Ctrl} = 0$ and place them in their original positions in Y. Recall we hold a permutation θ from step 2 that will help us unshuffle these blocks. In steps 5b-5c, we place the blocks into the positions where they were retrieved after the shuffle in step 2d and place 0s in all other blocks. We then unshuffle in step 5d. The input to the shuffle was of length $2k$ as we included one dummy for each block. Hence, the output after the unshuffle is of length $2k$ where the k dummies were placed after the first k blocks. We only care about the first k blocks w'. w' now consists of all zeros except for the indices where X_i should be inserted. To compute the final indices, we start with w' (step 5e) and compute its prefix sum (step 5f). Note that at each step we also add 1 to count the number of Y_i before each Y_j.
6. [**Compute merge permutation** π] Step 5f computes the inverse permutation. Hence, we invoke Π-Inv to invert the permutation and output it.

7.2 Π-SquareRootMerge Proofs

Theorem 7 (Π-SquareRootMerge **correctness).** Π-SquareRootMerge *realizes the* $\mathcal{F}_{\mathsf{merge}}$ *functionality when* $|X| = n^{\frac{1}{2}}$ *and* $|Y| = n$.

Proof. Correctness can be verified by inspection via the logic in Section 4.3. □

Theorem 8 (Π-SquareRootMerge **security**)**.** Π-SquareRootMerge*, as our other protocols, is secure against semi-honest adversaries in the* ***sf****-hybrid model, where* ***sf*** *is the list of functionalities invoked by* Π-SquareRootMerge *(*Shuffle*,* Unshuffle*,* AggregationTree*, and* MPC*).*

Proof. Simulation follows exactly that of Section 3.5, where the circuit consists of Shuffle, Unshuffle, AggregationTree, and MPC gates. Hence, Π-SquareRootMerge is simulatable. □

8 Evaluation

In this section, we estimate the concrete costs of our protocols. We start with the symmetric Π-Logstar and Π-Median (Section 8.1) and then continue with the asymmetric Π-SquareRootMerge and Π-CubeRootMerge (Section 8.2). We consider input lists of size $n = 2^{20}$ with $\ell = 128$ bit elements, and express the cost in terms of: (1) the number of comparisons, and (2) the number of rounds due to comparisons. For the number of rounds, we assume GMW-style comparisons, which require $\log \ell$ rounds. For the asymmetric protocols, we adjust the size of one list to $n^{\frac{1}{2}}$ and $n^{\frac{1}{3}}$, respectively. Note that we estimated the total bandwidth cost, with secure comparisons being the primary bottleneck, responsible for more than 90% of the total bandwidth in all our protocols except the Π-SquareRootMerge protocol. Furthermore, since our protocols are described at a low level, essentially as pseudocode, we do not expect any additional significant costs that are unaccounted for in our analysis.

We benchmark our protocols against the state-of-the-art Batcher's network merge and shuffle-then-sort (see Section 2) . Recall that the shuffle-then-sort technique concatenates $X||Y$, shuffles them, and then uses some secure sort such as quick-sort [HKI+13,PRRS24]. To sort a list of size $n = n_0 + n_1$ with ℓ-bit elements via quick-sort, we require $O(n \log n)$ secure comparisons and $O(\log n \log \ell)$ rounds. We use 1.44 for constant (empirical constant resulting in $1.44 \cdot n \log n$ comparisons), originating from the choice of pivots to partition the sub-arrays. In Batcher's network, we use $\frac{n}{2} \log n$ comparisons and $(1 + \log n) \log \ell$ rounds. For completeness, we also include the costs of [BBD+22] when evaluating our symmetric protocols, despite their large constants. More specifically, we include [BBD+22]'s costs for their (1) full protocol, and also for their (2) $\Pi_{\mathsf{SSM}\text{-loglog}n}$ subprotocol (Step 1 in our high-level description of [BBD+22] in Section 3.3), which also merges two arbitrary lists of length n, but is used only as a subprotocol of [BBD+22]'s full protocol.

8.1 Π-Logstar and Π-Median Evaluation

We first evaluate our symmetric protocols. We present our findings in Figure 12, then interpret our results, and discuss some key aspects of our cost estimates.

Protocol	**# Comparisons**	**# Rounds**
[BBD$^+$22]'s Full Protocol	$1.27 \cdot 10^8$	483
[BBD$^+$22]'s Subrotocol $\mathit{\Pi}_{\mathsf{SSM}\text{-}\mathrm{loglog}n}$	$1.15 \cdot 10^8$	179
(Shuffled) Quick-Sort	$6.34 \cdot 10^7$	212
Batcher's Network	$2.20 \cdot 10^7$	154
Π-Logstar	$\mathbf{1.53 \cdot 10^7}$	155
Π-Median	$5.43 \cdot 10^7$	**118**

Fig. 12: Comparison of our protocols Π-Logstar and Π-Median with the state-of-the-art merge techniques. We let the input length $n = 2^{20}$ and the element bitlength $\ell = 128$. We express our cost in terms of number of comparisons and the number of rounds due to comparisons, which we discover is a bottleneck in our protocols.

Figure 12 Results Interpretation. For $n = 2^{20}$, Π-Logstar reduces the number of comparisons $\approx 1.4\times$ over Batcher's network and $\approx 4.1\times$ over shuffle-then-sort. It achieves this without significantly increasing the number of rounds. To compute the comparisons, Π-Logstar uses 155 rounds, while Batcher's network uses 154 and shuffle-then-sort 212. We note that for a larger input, Π-Logstar's improvement becomes more pronounced due to superior asymptotics. E.g., for $n = 2^{30}$, Π-Logstar, with $1.69 \cdot 10^{10}$ comparisons, reduces the number of comparisons $\approx 2.0\times$ over Batcher's network and $\approx 5.7\times$ over shuffle-then-sort.[5] At the same time, Π-Logstar uses 239 rounds, whereas Batcher's network uses 224 rounds and shuffle-then-sort uses 313 rounds. Π-Median provides a tradeoff between bandwidth and rounds. For $n = 2^{20}$, it increases the number of required comparisons $\approx 2.5\times$ over Batcher's network while still decreasing them $\approx 1.2\times$ over shuffle-then-sort, but decreases the number of rounds over both, i.e. $\approx 1.3\times$ over Batcher's network and $\approx 1.8\times$ over shuffle-then-sort. For $n = 2^{30}$, Π-Median, with $7.45 \cdot 10^{10}$ comparisons, increases bandwidth $\approx 2.2\times$ over Batcher's network while still decreasing it $\approx 1.3\times$ over shuffle-then-sort, but decreases the number of rounds over both, namely $\approx 1.5\times$ over Batcher's network and $\approx 2.1\times$ over shuffle-then-sort. Hence, Π-Median is suitable for high-latency networks. [BBD$^+$22] stresses superior asymptotics, but is concretely less efficient even than shuffle-then-sort. With respect to Π-Logstar and $n = 2^{20}$, it uses $\approx 8.3\times$ more comparisons and $\approx 3.1\times$ more rounds. [BBD$^+$22]'s $\mathit{\Pi}_{\mathsf{SSM}\text{-}\mathrm{loglog}n}$ subprotocol (Step 1 in Section 3.3), which also serves as a standalone symmetric merge, uses $\approx 7.5\times$ more comparisons than Π-Logstar and $\approx 1.2\times$ more rounds.

We now discuss key aspects of our cost estimates. We start with Π-Logstar, continue with Π-Median, and then finish with [BBD$^+$22].

[5] We note that databases with $n \geq 2^{30}$ are not uncommon. For example, such large datasets are frequently found in areas like network traffic monitoring, financial transactions, and online advertising data.

Π-Logstar. We follow Π-Logstar as defined in Figure 8 except we set the block size $m := 7$ instead of $\log n$. In the base case, we then use [MI04]'s merging network that requires 21 comparisons for inputs of size 7 (which is slightly less than Batcher's network).[6] Note that with this change, we execute both the recursive step and the base case once. To sort medians in step 2b, we use Batcher's network.

Π-Median. We follow our algorithm almost exactly as in Figure 9. Our only deviations are in the base case where we increase the size of the base case so that we finish recursion after exactly 2 steps for both $n = 2^{20}$ and $n = 2^{30}$. We use Batcher's network [Bat68] in the base case.

[BBD⁺22]. We note that their protocol is highly optimized with respect to asymptotics. We do not see any simple way to optimize their protocol for concrete efficiency.

8.2 Π-SquareRootMerge and Π-CubeRootMerge Evaluation

Next, we evaluate our asymmetric protocols. We first evaluate Π-SquareRootMerge, then we evaluate Π-CubeRootMerge. For each protocol, we present our findings and interpret our results.

Π-SquareRootMerge. See our results in Figure 13. We present the costs for the exact same algorithm as in Figure 11.

Protocol	**# Comparisons**	**# Rounds (Agg. Tree and Comparisons)**
(Shuffled) Quick-Sort	$3.02 \cdot 10^7$	202
Batcher's Network	$1.05 \cdot 10^7$	147
Π-SquareRootMerge	$\mathbf{2.10 \cdot 10^6}$	**56**

Fig. 13: Comparison of Π-SquareRootMerge with the state-of-the-art merge techniques. We let $n = 2^{20}$ and set the length of the input lists to $n^{\frac{1}{2}}$ and n, respectively. We express our cost in terms of number of comparisons and the number of rounds due to comparisons and aggregation trees (as comparisons are not a bottleneck for round complexity in Π-SquareRootMerge).

Figure 13 Results Interpretation. While for the symmetric protocols and our Π-CubeRootMerge comparisons constitute an overwhelming cost, this is not quite true for Π-SquareRootMerge. We estimate comparisons make up $\approx 50\%$ of the total bandwidth cost. Hence, while we decrease the number of comparisons $\approx 5\times$ over Batcher's merge, we estimate our bandwidth is $\approx 2.5\times$ smaller. For the

[6] For $n = 2^{30}$, we use $m = 8$ and a merging network with 25 comparisons [MI04].

rounds, aggregation trees in combination with the comparisons are the overwhelming bottleneck. We thus combine the round cost for the comparisons and the aggregation trees to estimate the total cost. We get a $\approx 2.6\times$ improvement over Batcher's merge.

Π-CubeRootMerge. See our results in Figure 14. We present the costs for the exact same algorithm as in Figure 10. Note that a similar performance was already achieved by [BBD+22]'s protocol, but did not have concrete estimates. Our Π-CubeRootMerge is highly similar to that protocol, but is modified and expressed in our notation.

Protocol	**# Comparisons**	**# Rounds**
(Shuffled) Quick-Sort	$3.02 \cdot 10^7$	202
Batcher's Network	$1.05 \cdot 10^7$	147
Π-CubeRootMerge	$\mathbf{2.11 \cdot 10^6}$	**23**

Fig. 14: Comparison of Π-CubeRootMerge with the state-of-the-art merge techniques. We let $n = 2^{20}$ and set the length of the input lists to $n^{\frac{1}{3}}$ and n. As in Figure 12, we express our cost in terms of number of comparisons and the number of rounds due to comparisons.

Figure 14 Results Interpretation. Π-CubeRootMerge reduces the number of comparisons $\approx 5.0\times$. This corresponds to approximately the same bandwidth reduction. For the number of rounds, we estimate that comparisons count for $\approx \frac{2}{3}$ of the total rounds. Hence, we estimate we reduce the total round complexity $\approx 4.3\times$.

References

AHI+22. Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 125–138, New York, NY, USA, 2022. Association for Computing Machinery.

AHK+23. Gilad Asharov, Koki Hamada, Ryo Kikuchi, Ariel Nof, Benny Pinkas, and Junichi Tomida. Secure statistical analysis on multiple datasets: Join and group-by. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 3298–3312, New York, NY, USA, 2023. Association for Computing Machinery.

AKS83. M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in c log n parallel steps. Combinatorica, Volume 3, Issue 1, pp. 1–19, 1983.

APR$^+$24. A. Agarwal, S. Peceny, M. Raykova, P. Schoppmann, and K. Seth. Communication-efficient secure logistic regression. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, pages 440–467, Los Alamitos, CA, USA, jul 2024. IEEE Computer Society.

Bat68. K.E. Batcher. Sorting networks and their applications. Proceedings of the April 30 - May 2, 1968, spring joint computer conference, pp. 307–314, 1968.

BBD$^+$22. Mark Blunk, Paul Bunn, Samuel Dittmer, Steve Lu, and Rafail Ostrovsky. Secure merge in linear time and O(log log N) rounds. Cryptology ePrint Archive, Report 2022/590, 2022. https://eprint.iacr.org/2022/590.

BCG$^+$18. Christina Boura, Ilaria Chillotti, Nicolas Gama, Dimitar Jetchev, Stanislav Peceny, and Alexander Petric. High-precision privacy-preserving real-valued function evaluation. In Sarah Meiklejohn and Kazue Sako, editors, *FC 2018*, volume 10957 of *LNCS*, pages 183–202. Springer, Heidelberg, February / March 2018.

BDG$^+$22. Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. Secret-shared joins with multiplicity from aggregation trees. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 209–222. ACM Press, November 2022.

CHI$^+$19. Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. An efficient secure three-party sorting protocol with an honest majority. Cryptology ePrint Archive, Report 2019/695, 2019. https://eprint.iacr.org/2019/695.

CKN$^+$18. T.-H. Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 158–188. Springer, Heidelberg, December 2018.

DSZ15. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

FNO22. B.H. Falk, , R. Nema, and R. Ostrovsky. A linear-time 2-party secure merge protocol. International Symposium on Cyber Security, Cryptology, and Machine Learning (CSCML), pp. 408–427, 2022.

FO21. B.H. Falk and R. Ostrovsky. Secure merge with o(n log log n) secure operations. Conference on Information-Theoretic Cryptography (ITC), Article No. 7, pp. 7:1–7:29, 2021.

Goo10. Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In Moses Charika, editor, *21st SODA*, pages 1262–1277. ACM-SIAM, January 2010.

Goo14. Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In David B. Shmoys, editor, *46th ACM STOC*, pages 684–693. ACM Press, May / June 2014.

HICT14. Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint Archive, Report 2014/121, 2014. https://eprint.iacr.org/2014/121.

HKI$^+$13. Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC 12*, volume 7839 of *LNCS*, pages 202–216. Springer, Heidelberg, November 2013.

HKP20. David Heath, Vladimir Kolesnikov, and Stanislav Peceny. MOTIF: (almost) free branching in GMW - via vector-scalar multiplication. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 3–30. Springer, Heidelberg, December 2020.

HKP21. David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Masked triples - amortizing multiplication triples across conditionals. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 319–348. Springer, Heidelberg, May 2021.

KS08. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

MI04. G. Morohashi and S. Iwata. Some minimum merging networks. Theoretical Computer Science, 329(1-3), 237–250, 2004.

MR18. Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

PRRS24. Stanislav Peceny, Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. Efficient permutation correlations and batched random access for two-party computation. Cryptology ePrint Archive, Paper 2024/547, 2024. https://eprint.iacr.org/2024/547.

Val75. L.G. Valiant. Parallelism in comparison problems. SIAM Journal on Computing, Vol. 4, Iss. 3, pp. 348-355, 1975.

YPHK23. Yibin Yang, Stanislav Peceny, David Heath, and Vladimir Kolesnikov. Towards generic mpc compilers via variable instruction set architectures (visas). In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2516–2530, New York, NY, USA, 2023. Association for Computing Machinery.

ZRE15. Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

Supplementary Material

Disclaimer

Case studies, comparisons, statistics, research and recommendations are provided "AS IS" and intended for informational purposes only and should not be relied upon for operational, marketing, legal, technical, tax, financial or other advice. Visa Inc. neither makes any warranty or representation as to the completeness or accuracy of the information within this document, nor assumes any liability or responsibility that may result from reliance on such information. The Information contained herein is not intended as investment or legal advice, and readers are encouraged to seek the advice of a competent professional where such advice is required.

These materials and best practice recommendations are provided for informational purposes only and should not be relied upon for marketing, legal, regulatory or other advice. Recommended marketing materials should be independently evaluated in light of your specific business needs and any applicable laws and regulations. Visa is not responsible for your use of the marketing materials, best practice recommendations, or other information, including errors of any kind, contained in this document.