

Linear-Time Secure Merge in $O(\log \log n)$ Rounds

Mark Blunk 

Stealth Software Technologies, Inc., Los Angeles, CA, USA

Paul Bunn 


Stealth Software Technologies, Inc., Los Angeles, CA, USA

Samuel Dittmer 

Stealth Software Technologies, Inc., Los Angeles, CA, USA

Steve Lu 

Stealth Software Technologies, Inc., Los Angeles, CA, USA

Rafail Ostrovsky 

Departments of Computer Science and Mathematics, UCLA, Los Angeles, CA, USA

Abstract

The problem of Secure Merge consists of combining two sorted lists (which are either held separately by two parties, or secret-shared among two or more parties), and outputting a single merged (sorted) list, secret-shared among all parties. Just as *insecure* algorithms for comparison-based **sorting** are slower than **merging** (i.e., for lists of size n , $\Theta(n \log n)$ versus $\Theta(n)$), we explore whether an analogous separation exists for *secure* protocols; namely, if there exist techniques for performing *secure* merge that are more performant than simply invoking secure sort.

We answer this question affirmatively by constructing a secure merge protocol with optimal $\Theta(n)$ communication and computation, and $\Theta(\log \log n)$ rounds of communication. Our results are based solely on black-box use of basic secure primitives, such as secure comparison and secure shuffle. Since two-party secure primitives require computational assumptions, while three-party do not, our protocols achieve these bounds against semi-honest adversaries via a computationally secure two-party (resp. an information-theoretically secure three-party) secure merge protocol.

Secure **sort** is a fundamental building block used in many MPC protocols, e.g., various private set intersection protocols and oblivious RAM protocols. More efficient secure sort can lead to concrete improvements in the overall run-time. Since secure sort can often be replaced by secure merge – as inputs (from different participating players) can be presorted – an efficient secure merge protocol has wide applicability. There are also a range of applications in the field of secure databases, including secure database *joins*, as well as updatable database storage and search, whereby secure merge can be used to insert new entries into an existing (sorted) database.

In building our secure merge protocol, we develop several subprotocols that may be of independent interest. For example, we develop a protocol for secure *asymmetric* merge (when one list is much larger than the other).

2012 ACM Subject Classification Security and privacy → Cryptography; Security and privacy → Database and storage security

Keywords and phrases Secure Merge, Secure Sort, Secure Databases, Private Set Intersection

Digital Object Identifier 10.4230/LIPIcs.ITC.2025.7

Related Version *Full Version*: <https://eprint.iacr.org/2022/590>

Funding This work was performed under the following financial assistance award number 70NANB21H064 from U.S. Department of Commerce, National Institute of Standards and Technology. The statements, findings, conclusions, and recommendations are those of the author(s) and do not necessarily reflect the views of the National Institute of Standards and Technology or the U.S. Department of Commerce.



© Mark Blunk, Paul Bunn, Samuel Dittmer, Steve Lu, and Rafail Ostrovsky;
licensed under Creative Commons License CC-BY 4.0
6th Conference on Information-Theoretic Cryptography (ITC 2025).
Editor: Niv Gilboa; Article No. 7; pp. 7:1–7:23



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Rafail Ostrovsky: This research was supported in part by DARPA under Cooperative Agreement HR0011-20-2-0025, the Algorand Centers of Excellence programme managed by Algorand Foundation, NSF grants CNS-2246355, CCF-2220450, CNS-2001096, US-Israel BSF grant 2022370, Amazon Faculty Award and Sunday Group. Any views, opinions, findings, conclusions, or recommendations contained herein are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, the Algorand Foundation, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes, notwithstanding any copyright annotation therein.

1 Introduction

As the practice of collecting and analyzing data has increased in recent decades, combined with the growing desire of examining data of different types held by different organizations, there has been a corresponding desire for protocols that are able to compute on aggregated data without actually requiring the raw data to be combined or exposed. Indeed, the field of secure multiparty computation (MPC) within cryptography seeks to address exactly such scenarios. Meanwhile, with the **sort** functionality being a prominent component of several desired analyses, there has been significant work in optimizing *secure sort* functionality (e.g. **sort** under MPC), as well other basic secure functionalities that require **sort** as a subroutine. For example, in research in the MPC subfields of Private-Set Intersection (PSI) [8, 17, 24] and Oblivious RAM (ORAM) [20, 23], it is often the case that the secure **sort** is the bottleneck in resulting protocols in these areas.

While MPC protocols for **sort** – and related functionalities that build upon it – have become extremely efficient, there is an unavoidable cost of e.g., a security parameter multiplier in measuring the complexity of any secure computation, on top of the $\Theta(n \log n)$ cost of performing (insecure) **sort** (here n is the size of the list in question). Given the common scenario in which a handful of organizations each have access to their own data, and require a **sort** on their aggregated lists, it is natural to ask if one can avoid the inherent overhead of securely implementing **sort** by first having parties locally sort their own data (which can be done *insecurely*, and hence without incurring this overhead), and then performing a secure **merge** on the individual sorted lists. Indeed, this approach has the promise of adding security at minimal cost, since the overhead of adding security is now only applied to the $\Theta(n)$ **merge** protocol, which effectively means there is an extra $\Theta(\log n)$ cushion to absorb the overhead cost of performing the computation securely, and still having an overall $\Theta(n \log n)$ *secure sort* protocol.

Unfortunately, attempting to minimize the overhead of secure computation by first performing a local (insecure) **sort** followed by secure **merge** has to-date been an ineffective strategy, due to the fact that much less is known about secure **merge** protocols than secure **sort** protocols. Indeed, prior to our work, there was no known secure **merge** protocol that simultaneously achieved near-optimal asymptotic performance in the three key metrics: computation, communication, and round complexity (see Section 2.2 and Table 1 for a summary of prior works exploring secure merge, and how they perform in these three metrics). In this work, we present a secure **merge** protocol that effectively eliminates all inefficiencies, thereby substantially reducing the cost of secure **sort** (where the above strategy of first locally sorting one’s own data can be employed) and any secure protocol that builds on top of it. Namely, we construct a protocol that instantiates the following:

► **Theorem 1 (Informal).** *There exists a secure merge protocol for two or three parties with $\Theta(n)$ run-time (computation and communication) and $\Theta(\log \log n)$ rounds that relies only on black-box access to the secure functionalities defined in §3.4: comparison, shuffle, and conditional-select (mux).*

Theorem 1 is asymptotically optimal in two of the three key metrics (computation and communication), and near-optimal in round-complexity. Our theorem gives a two-party secure merge under standard cryptographic assumptions, and a three-party secure merge with information-theoretic setting with semi-honest security and no collusion. Indeed, in the two-party case, all black-box functionalities referenced in the theorem above can be efficiently realized using two standard cryptographic assumptions: Oblivious transfer (OT) and the existence of an additively homomorphic public key cryptosystem. Meanwhile, in the three-party case, these black-box functionalities can be realized with information-theoretic security, see e.g., [11]. While there are ways to extend our results to more parties beyond three, see for example the discussion in the full version; we do not explore these extensions directly in this paper.

Besides being useful as a stand-in for secure `sort` in scenarios where two or more parties can each locally sort their own data (“in the clear”), the secure `merge` functionality has broader relevance in many other applications as well. In the area of database management, for example, in settings where records are encrypted, our secure merge protocol can be used for database operations such as joins, or for inserting new records into the database [2, 9, 18, 27].

1.1 Paper Structure

In §2 we summarize previous work on the secure sort/merge problem, and provide a comparison of our main result with relevant earlier works in Table 1. In §3 we set notation and discuss the high level techniques and primitives on which our merge protocols rely. We present our main secure merge protocols, together with statements and proof sketches of their properties, in §4; and then analyze their complexity in §5. In §6 we present the two underlying secure merge protocols required by our protocols in §4. Supplementary material including a survey of applications whose performance could be improved by the adoption of our secure merge protocol as well as an expanded discussion of our main results with additional protocols – including the description of a separate constant round secure merge protocol in the asymmetric case that is a generalization of the protocol $\Pi_{\text{SAM-}n^{1/3}}$ of §6 (where one list has size n and the other n^α for fixed $\alpha < 1$) – can be found in the full version.

2 Previous Work

As the merging/sorting of lists is a fundamental problem in computer science, there has been enormous research in this area. Consequently, we summarize here only the most relevant works; see cited works and references therein for a more complete overview and discussion. Not surprisingly, many of the protocols for *securely* merging/sorting lists draw inspiration from their *insecure* counterparts (not to mention the generic approach of converting insecure protocols to secure protocols, e.g., via garbled circuits or ORAM or Garbled RAM (GRAM) or fully-homomorphic encryption). We discuss below previous work for both insecure and secure variants, and compare previous results to our main result in Table 1.

2.1 Insecure Merge Algorithms

Sorting Networks. The relationship between secure merging and secure sorting can be traced back to [3], which built a sorting network of size $O(n \log^2 n)$ from $\log n$ merging networks, each of size $O(n \log n)$. There are $(2n)! = 2^{O(n \log n)}$ permutations on $2n$ elements, but $\binom{2n}{n} = 2^{O(n)}$ possible ways two sorted lists can be merged together. This gives combinatorial lower bounds of $\Omega(n \log n)$ and $\Omega(n)$ comparators for sorting and merging networks, respectively. Although an asymptotically optimal sorting network of size $O(n \log n)$ was later achieved by Ajtai,

Komlos, and Szemerédi [1], merging networks cannot achieve the combinatorial lower bound of n comparators. A merging network on lists of size $\Theta(n)$ require $\Omega(n \log n)$ comparators, as shown by Yao and Yao [26], and depth of $\Omega(\log n)$, as shown by Hong and Sedgewick [16].

RAM and PRAM. The classical merge algorithm requires $O(n)$ work on a RAM machine, see e.g., [21]. As discussed below, [10] uses an approach that is inspired by this classical merge algorithm, and achieves matching asymptotic work (albeit with $O(n)$ round-complexity).

A parallel RAM machine (PRAM) allows multiple processors to act on the same set of memory in parallel. We are in particular interested in Concurrent-Read-Exclusive-Write (CREW) PRAM, where all processors can read the same memory simultaneously, but processors cannot write to the same memory address at the same time, see e.g., [4] for more formal definitions and a discussion of various PRAM models. For PRAM machines, Valiant showed in [25] that $O(n)$ processors could merge two lists of size $O(n)$ in time $O(\log \log n)$. This was improved by Borodin and Hopcroft [4] to $O(n/\log \log n)$ processors, who also showed that the time bound of $O(\log \log n)$ is optimal when limited to $O(n)$ processors. As a rough heuristic, we expect the number of processors and total work done by a PRAM algorithm to serve as a lower bound for the round complexity and communication complexity of a corresponding secure protocol, motivating the following:

► **Conjecture.** *Any linear-time protocol for two parties to securely merge their lists (each of size $\Theta(n)$) requires $\Omega(\log \log n)$ rounds of communication.*

Notice that if the above conjecture is valid, then our secure merge protocol (§4) is asymptotically optimal in all three key metrics. To lend weight to this conjecture, we note that if a 2-party protocol Π securely realizes some functionality \mathcal{F} in R rounds and C communication, and each party can execute each round of the protocol in $O(1)$ time on a CREW PRAM with C processors, then there exists an algorithm A that realizes \mathcal{F} on a single CREW PRAM machine with C processors, $O(RC)$ total work, and $O(R)$ time. Indeed, A merely executes Π , playing the role of all parties. If a protocol for secure merge existed with $C = n$, $R = o(\log \log n)$, this would immediately imply an insecure merge algorithm with $O(n)$ processors and $o(\log \log n)$ time, contradicting the lower bound of [4] cited above. Thus the conjecture is true for the special case of secure merge protocols where each round of the protocol can be executed in $O(1)$ time on a CREW PRAM with $O(n)$ processors.

Both the Valiant and the Borodin-Hopcroft algorithms rely on the following basic construction: Split each list into blocks of size \sqrt{n} , with \sqrt{n} medians. Running all pairwise comparisons between medians identifies which block of the opposite list each median is mapped to; then another round of pairwise comparisons identifies the exact position the median is mapped to within that block. This creates \sqrt{n} subproblems of size \sqrt{n} , giving the recurrence $c(n) = \sqrt{n} \cdot c(\sqrt{n})$, if $c(n)$ is the cost of merging two lists of length n . With sufficiently many processors, this recurrence yields $O(\log \log n)$ run-time. Our protocols in §4 use similar techniques as a starting point, though many additional ideas are needed to handle the *secure* setting, e.g. securely handling the case where blocks from one list potentially span more than one block of the other list.

2.2 Secure Merge Algorithms

Notation. To aid the comparison to prior work, we introduce the following constants: κ is a computational security parameter (e.g., $\kappa = 128$ is standard), β_{HE} (resp. β_{FHE}) is the ciphertext expansion of an additively homomorphic (resp. fully homomorphic) cryptosystem, γ is the cost of decryption, and μ is the cost of multiplication in the FHE cryptosystem.

The parameters β_{HE} , β_{FHE} , γ , and μ depend on the cryptosystem and on κ . Asymptotically, $\kappa \gg \log n$ is necessary so that a random bit-string of length κ cannot be guessed in the time it takes to traverse the list, and (for suitable choice of cryptosystems) β_{HE} and β_{FHE} approach 1 as the plaintext size grows. In practice, fully homomorphic encryption schemes are more costly than additive-only homomorphic ones, so we expect $\beta_{\text{FHE}} > \beta_{\text{HE}}$ and $\mu > \gamma$.

We assume the objects to be sorted are contained in $O(1)$ memory words of size W bits. Unless explicitly stated otherwise, our communication and computational complexity numbers are given in terms of memory words and primitive operations on memory words.

Security via Generic Transformation. We explore here two naïve solutions for transforming an *insecure* merge algorithm to a secure one (see Table 1 for a succinct comparison of our secure merge protocol to these naïve solutions):

GARBLED CIRCUITS. By choosing any (insecure) sorting algorithm that can be represented as a circuit, the parties can use garbled circuits to (securely) sort their list in $O(1)$ rounds. As discussed above in §2, for comparison-based merging networks, $\Omega(n \log n)$ comparisons and $\Omega(\log n)$ depth are necessary, and achievable by the Batcher merging network [3].

We note that obtaining κ bits of computational security when merging lists whose elements have size W bits requires $\kappa \cdot W$ bits, or κ words of communication for each comparison. Thus, obtaining secure merge via a garbled circuit approach (for a circuit representing a merging network) would result in a constant-round protocol with $\Omega(\kappa \cdot n \log n)$ communication.

On the other hand, using GMW-style circuit evaluation (see [12]) instead of garbled circuits can reduce the overhead of each comparison (from κ down to constant), but it incurs a hit in round complexity (proportional to the depth of the circuit instead of constant-round).

FULLY HOMOMORPHIC ENCRYPTION (FHE). An $O(1)$ round protocol can also be constructed by having one party encrypt their inputs under FHE and send the result to the other party, who performs the desired calculations on ciphertexts. The other party then subtracts a vector of random values \mathbf{r} from the (encrypted) merged list and sends the result back to the first party, who decrypts to obtain the merged list shifted by \mathbf{r} . Now the parties hold an additive sharing of the sorted list.

As with garbled circuits, however, the calculations the second party performs on the ciphertexts must be input-independent (in order to avoid information leakage), and so the calculation must be represented by a circuit. This means that communication is (asymptotically) lower than the garbled circuit approach, since the first party need only provide ciphertexts of his list (which correspond to *inputs* to the circuit), as opposed to providing information for each *gate*. Therefore the communication required for FHE is $O(\beta_{\text{HE}} n)$.

However, while communication in the FHE approach may be (asymptotically) reduced (compared to the garbled circuit approach), notice that the computation is still $\Omega(\mu n \log n)$, where μ is the cost of a multiplication under FHE, as the circuit requires $\Omega(n \log n)$ comparison (multiplication) operations. Additionally, since the circuit has depth $\Omega(\log n)$, FHE will require bootstrapping to avoid ciphertext blowup, which will be expensive in practice.

ORAM AND GRAM. ORAM incurs at least an $\Omega(\log n)$ overhead [13, 19]. Currently known GRAM constructions are built using ORAM as a building block. Therefore, if one is to take an insecure merge implementation and try to compile it into a secure circuit using ORAM or GRAM, both the computational and communication complexity will be $O(n \log n)$, and thus these techniques are inapplicable if we aim to achieve linear complexity.

Shuffle-Sort Paradigm. One challenge facing any comparison-based *secure* merge (or sort) protocol is that the results of each comparison must be kept secret from each party, or else security is lost. One approach to allowing the results of the comparisons to be known *without* information leakage is to first *shuffle* (in an oblivious manner) the input lists. However, since shuffling will destroy the property that the input lists are pre-sorted, this approach reduces a secure *merge* problem to a secure *sort* problem, hence incurring the $\log n$ efficiency loss.

Comparison-based sorting has $O(n \log n)$ communication and computation and $O(\log n)$ rounds, while secure shuffling requires $O(n)$ communication and $O(1)$ rounds, see e.g., [11, 15]. Therefore, a secure sort using the shuffle-sort paradigm requires $O(n \log n)$ communication and computation and $O(\log n)$ rounds, which is worse than our results in all three metrics.

Oblivious Sort. Because the constant from [1] is too large for practical applications, a number of other approaches to secure sorting have been explored. The shuffle-sort paradigm mentioned above is one example of a large family of *oblivious* sort protocols, which allow for a variable memory access pattern as long as it is independent of the underlying list values, or *data oblivious*. We mention here the radix sort of Hamada et al. [14], which achieves $O((W \log W + W)n + n \log n)$ communication (in memory words) and $O(1)$ round complexity in the three-party honest majority setting with constant bit lengths of elements. The communication complexity was later improved by Chida et. al [8], to $O(n \log n)$ memory words. However, the round complexity depends linearly on W , so when $W \approx \log n$, this matches the round complexity of the other protocols.

Secure Merge Protocols. There are several works that investigate secure merge directly. The first, due to Falk and Ostrovsky [11], achieves $O(n \log \log n)$ communication complexity with $O(\log n)$ round complexity. The second, due to Falk, Nema, Ostrovsky [10], achieves the asymptotically optimal $O(n)$ communication complexity (with small constants), but requires $O(n)$ rounds of communication. For many cryptographic applications, a high round complexity causes more of a bottleneck than a high communication complexity due to network latency. Therefore, the secure merge protocol of [10], while both simple and asymptotically optimal in terms of communication, may still not be practical due to high round complexity.

In a subsequent work [6], Chakraborty et al. present a series of secure merge protocols, optimizing for concrete efficiency. Some of their protocols use similar machinery to our work presented here – and indeed, their paper cites our work here as inspiring some of their ideas. In [6], the authors explore the trade-offs between communication and rounds, giving protocols with worse asymptotic performance than our protocols. See the full version for a discussion of the concrete efficiency of our protocols and comparisons with standard (insecure) merge protocols.

Three Party Sort and Merge protocols. The three party honest majority setting is a natural fit for real-world protocols in the client-server model, including ORAM and PSI protocols. Prior work in this area has shown how the use of three parties facilitates more efficient protocols, including through the use of one party to generate randomness for the other parties and more efficient shuffling protocols (see e.g., [11]). The oblivious sort protocols mentioned above [8, 14] use three parties for shuffling and to enable the use of a Shamir secret sharing scheme. In [7] Chan et al. give a three-server merge protocol in the course of building a three-server ORAM scheme. This merge protocol, which requires three servers and an honest client, is most similar to the FNO protocol [10], and similarly requires $O(n)$ round complexity.

2.3 Comparison of Results

In Table 1 we give the communication, computation, and round complexity of our secure merge protocols, in comparison with the approaches described above. Our main result is a secure merge protocol that has (optimal) communication and computation $O(n)$, and round complexity $O(\log \log n)$ (see Theorem 1). Notice that $O(\log \log n)$ round complexity is superior to all other protocols in Table 1 *except* the constant-round garbled circuit and FHE approaches, each of which is inferior in the other two metrics (computation and communication). We also describe an asymmetric merge protocol on lists of size (n^α, n) for fixed $\alpha < 1$ that achieves the same asymptotic complexity as our general secure merge protocol, but in only $O(1)$ rounds.

The parameters β_{HE} and γ arise out of the use of homomorphic encryption for two-party shuffle, and so are only relevant for comparing secure merge protocols against two-party merge networks with standard MPC. Namely, in the three-party setting (which is assumed for [8, 15]), we can set $\beta_{\text{HE}} = \gamma = 1$ for the last four protocols of Table 1.

■ **Table 1** Comparison of secure merge protocols, with parameters as in §2.2; namely: n is the list size, $\alpha < 1$ is any fixed constant, κ is a computational security parameter, μ is the cost of FHE multiplication, β_{HE} and β_{FHE} are ciphertext expansions, and γ is the decryption cost ($\beta_{\text{HE}} = \gamma = 1$ in the three-party setting). We set the word size $W = \Theta(\log n)$ to simplify the formulas.

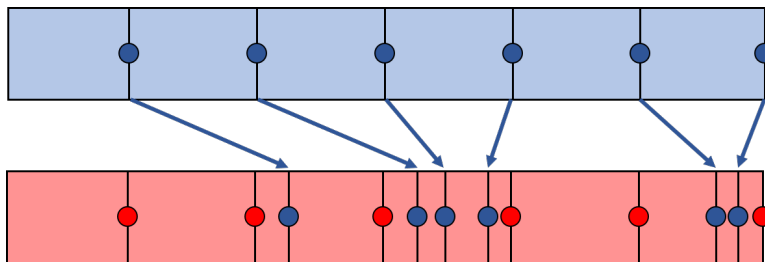
| Protocol | Computation | Communication | Rounds |
|---|----------------------------------|---|---------------------|
| (Garbled) Merge Network [Folklore] | $O(\kappa \cdot n \log n)$ | $O(\kappa \cdot n \log n)$ | $O(1)$ |
| (GMW) Merge Network [Folklore] | $O(n \log n)$ | $O(n \log n)$ | $O(\log n)$ |
| (FHE) Merge Network [Folklore] | $O(\mu \cdot n \log n)$ | $O(\beta_{\text{FHE}} n)$ | $O(1)$ |
| Shuffle-Sort [15] | $O(n \log n)$ | $O(n \log n)$ | $O(\log n)$ |
| Oblivious Radix Sort [8] | $O(n \log n)$ | $O(n \log n)$ | $O(\log n)$ |
| Secure Merge of [11] | $O(n \log \log n + \gamma n)$ | $O(n \log \log n + \beta_{\text{HE}} n)$ | $O(\log n)$ |
| Secure Merge of [10] | $O(\gamma n)$ | $O(\beta_{\text{HE}} n)$ | $O(n)$ |
| Our Secure (n^α, n) Merge [Fig. 5] | $O(2^{1/(1-\alpha)^3} \gamma n)$ | $O(2^{1/(1-\alpha)^3} \beta_{\text{HE}} n)$ | $O(1/(1-\alpha)^3)$ |
| Our Secure (n, n) Merge [Fig. 4] | $O(\gamma n)$ | $O(\beta_{\text{HE}} n)$ | $O(\log \log n)$ |

While our primary focus in this paper is on asymptotic performance (rounds, communication, computation), our protocols have many degrees of freedom that allow customization for concrete performance as well. By making different choices based on actual conditions (list size, network bandwidth, etc.) – in terms of which sub-protocols to use and when to stop recursing between our high-level merge protocols – we can give specializations of our protocol that provide high concrete performance. For example, our protocol will outperform a generic instantiation of the secure merge protocol that is based on the Batcher odd-even merge network, in both round complexity and the number of secure comparisons – see the full version for details.

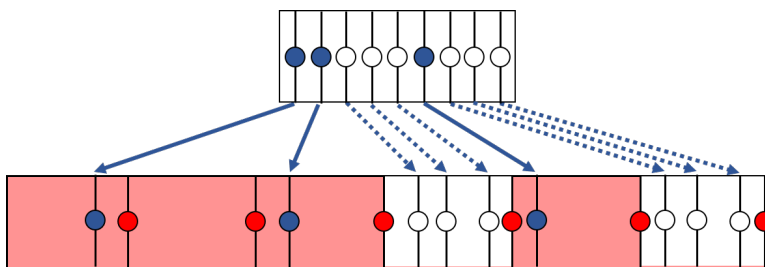
3 Overview of Techniques

At a high level, the strategy of our secure merge protocols is to *partition* the original lists into several blocks, then *align* these blocks (find the appropriate block(s) from the other list that span the “same” range of values); perform secure merge on these blocks; and finally combine (concatenate) the blocks together to obtain the final merged list. Ignoring for the moment issues in aligning the blocks (e.g. a block from one list spanning numerous blocks from the other list), this “partition-align” strategy – of using partitioning to achieve secure merge on

larger lists via several smaller secure merge subproblems – has the following appeal: If we partition the lists into k blocks, each with n/k elements, then for e.g. $k = n/\log \log n$, we could apply the linear protocol of [10] to each block. Since each block contains $n/k = \log \log n$ elements, each subproblem requires $O(\log \log n)$ communication, computation, and round-complexity. Furthermore, since each subproblem can be performed in parallel, the total cost across all $k = n/\log \log n$ blocks will be $O(n)$ computation and communication (and $O(\log \log n)$ rounds). This matches our target complexity in all three metrics.



■ **Figure 1** Merging the k medians (blue dots) from the top list into the correct locations with respect to the bottom list (the k medians of the bottom list are shown in red).



■ **Figure 2** Merging a smaller list (9 elements) into a larger list, and then classifying the elements in the first list and blocks from the second list that are “poorly-aligned.” Namely, poorly-aligned elements in the first list (depicted as white) means multiple (in this case three or more) elements from the first list map to the *same block* of the second list; and similarly the blocks in the second list that contain three or more elements from the first are poorly-aligned (and also depicted as white).

3.1 Identifying Poorly-Aligned Blocks

Of course, the above simplified strategy and analysis ignores several challenges that arise in practice:

Block Alignment. In order for the **partition-align** strategy to make sense when merging together two blocks (one from each list): Given a block (contiguous set of values) from one list, we must identify the appropriate block(s) from the other list that contains the “same” range of values.

Poorly-Aligned Blocks. If we partition say the first list into equal block sizes of n/k elements, the simplified analysis above assumed these blocks precisely align with exactly one block from the second list. In practice, a given block from the first list can align with arbitrarily many blocks from the second list, including e.g. the extreme case where a single block from the first list has a range of values that encompasses the entirety of the second list (see Figure 2).

Obliviousness of Alignment. Notice that being able to observe how the blocks from one list overlap with the blocks of the other list can leak information about the (relative) values on each list, thereby compromising overall security. In particular, any secure merge protocol employing the **partition-align** approach must hide all information regarding the nature of the alignments.

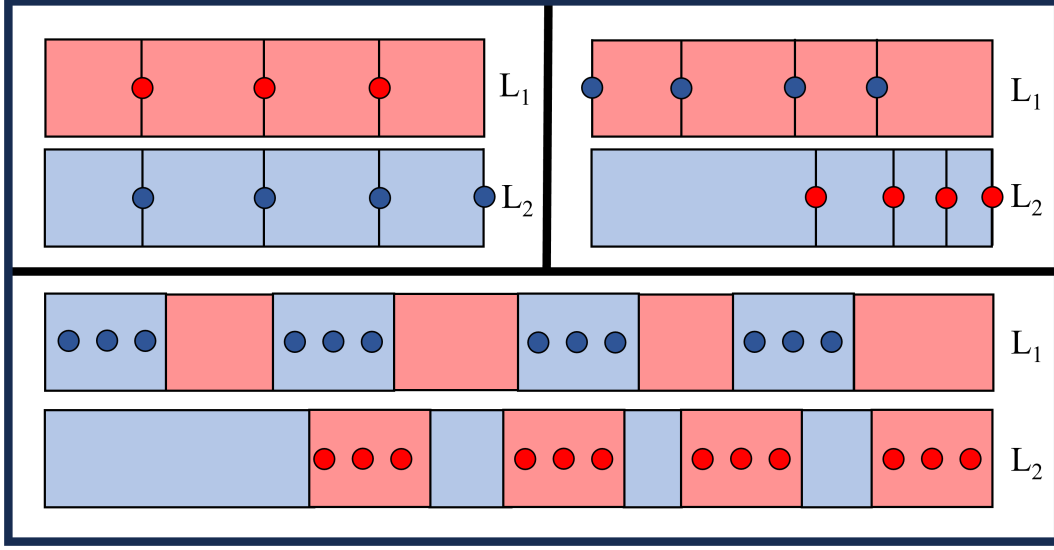
Our main contribution in this paper can be viewed as resolving the above three challenges:

1. We resolve the “Block Alignment” challenge by running a secure merge protocol to identify where the k partition points (“medians”) of one list belong within the second list. This requires a secure *asymmetric* (since one of the lists – of size k – is smaller than the other list – of size n) merge protocol. However, since this protocol is only invoked twice (once in each direction to merge the k medians of one list into the other), it can have complexities proportional to the *larger* list size n and still achieve our overall target metrics; we leverage this fact in our secure asymmetric merge protocol (Figure 5).
2. We resolve the “Poorly-Aligned Blocks” challenge as follows. First, we classify a block as “poorly-aligned” if (the range of values in) it spans “too many” blocks of the other list (or vice-versa). Then for merging poorly-aligned blocks, we again employ a secure asymmetric merge (merging one block from one list with multiple blocks from the other list). Depending on how “poor” the alignment (e.g. a single block from one list might span *all* blocks of the other list), this secure asymmetric merge protocol might have the larger list of size comparable to the original list.
3. We resolve the “Obliviousness of Alignment” challenge in three ways. First, we hide the classification of which blocks are well/poorly-aligned. Second, for the merging of “well-aligned” blocks, we apply a remarkable lemma (Lemma 2 below) about k -medians that allows the introduction of dummy elements to each block to ensure they are perfectly aligned, and then merge the resulting (slightly larger) blocks (see Figure 3). Third, for the poorly-aligned blocks, we observe that a “worst possible” alignment scenario can be defined, which provides a bound on the number and nature of the poorly-aligned blocks. In particular, we bound the number of highly *unbalanced* invocations of the secure asymmetric merge protocols – those in which a single block from one list spans *many* blocks from the other – which are costly to run. In particular, we always perform the same set of secure asymmetric merges (i.e. for a (fixed, known) set of list sizes) for the poorly-aligned blocks, regardless of how many blocks are actually classified as poorly-aligned (and the specific nature of their “poor” alignment).

► **Lemma 2.** *Let L_1 and L_2 denote two (sorted) lists of size n , and let $\mathcal{M}_{1,k}$ and $\mathcal{M}_{2,k}$ denote their k medians. Let $L'_1 = L_1 \sqcup \mathcal{M}_{2,k}$ denote the list (of size $2n$) resulting from merging $\mathcal{M}_{2,k}$ with L_1 , with each element in $\mathcal{M}_{2,k}$ duplicated n/k times in L'_1 . Let \mathcal{M}'_{2k} denote the $2k$ medians of L'_1 . Then the $2k$ medians of L'_1 are exactly the (merged) k medians of L_1 and L_2 : $\mathcal{M}'_{2k} = \mathcal{M}_{1,k} \sqcup \mathcal{M}_{2,k}$.*

While several details need to be worked out – such as tuning parameters and specifying how to handle dummy values and superfluous merging of (phantom/ non-existent) poorly-aligned blocks, the above strategy describes our high level approach. After introducing the notation that will be used in the remainder of this paper, we next go into more detail on the specific techniques and subroutines that will be employed in our secure merge protocols.

For a given block from one list, we need to identify which block(s) from the other list span a similar range of values. We first count the number of blocks from the second list that span the range of values in the first list, and then classify the blocks as “poorly-aligned” if the number of blocks from the second list is too large (greater than some constant). We



■ **Figure 3** The *Expanding Medians* strategy for converting “well-aligned” to *perfect* alignment: The k -medians of L_1 and L_2 are identified (top-left) and merged into the opposite list (top-right). Then each median is duplicated in place $\frac{n}{k}$ times (bottom), and the resulting lists will consist of $2k$ blocks that are perfectly aligned (can be merged in parallel).

then collect all poorly-aligned blocks/elements from L_1 and L_2 (marked white in Figure 2) and treat them as an additional subproblem, after padding with dummy elements to avoid leaking information, and explain how to bound the number of poorly-aligned blocks in §5.2.

3.2 The Tag-Shuffle-Reveal Paradigm

We make repeated use of the *tag-shuffle-reveal* paradigm, which should be considered analogous to the shuffle-sort paradigm of prior sorting protocols, see e.g. [11, 15], or an extension of the shuffle-reveal paradigm of [14]. Succinctly, the tag-shuffle-reveal paradigm starts with each element of a list (obviously) tagged with some label. This label can be (a secret sharing of) its current index, or it can be the result of some multiparty computation, for example a bit representing the output of a comparison against another value. Then, after shuffling the list, the tag is revealed, and the list entries are rearranged accordingly. Because the shuffle step ensures that the tags are randomly ordered, the only requirement to ensure security is to ensure that the set of values the revealed tags take on do not depend on the underlying data. We use \hat{x}_i to denote the element x in position i after shuffling.

3.3 Extraction Protocols

One central application of the tag-shuffle-reveal paradigm is our collection of *extraction protocols* for pulling marked elements from a larger list into a smaller list or set of lists. Marked elements can be extracted and kept in their original order ($\Pi_{\text{Ext-Ord}}$), or extracted and shuffled (Π_{Extract}), or extracted into bins based on a tag they are marked with ($\Pi_{\text{Ext-Bin}}$). Of course, each of these protocols should reveal nothing about the location or number of tagged elements, and so the outputs will be padded with dummy elements where necessary. We use extraction protocols in a black box way in §4-6; full descriptions and proofs of the Π_{Extract} , $\Pi_{\text{Ext-Ord}}$, and $\Pi_{\text{Ext-Bin}}$ protocols can be found in the full version.

3.4 Primitive Functionalities

We recall the secure merge protocol from [10] discussed above, which requires $O(n)$ communication and $O(n)$ rounds, which we call $\Pi_{\text{SM-FNO}}$ in this work. Additionally, we use the trivial $O(n^2)$ communication, $O(1)$ round merge protocol, which makes every possible pair of comparisons. We call this protocol $\Pi_{\text{SM-ALL}}$ (see the full version for a full description of this protocol and a proof of its properties).

The protocols we present below are realized with black box calls to five “primitive” functionalities: Π_{Reveal} , Π_{Comp} , Π_{Sel} , Π_{Shuffle} , Π_{Dup} that act on additively shared secret values. Oblivious transfer and the existence of an additively homomorphic rerandomizable public key cryptosystem are sufficient assumptions to realize these functionalities (and $\Pi_{\text{SM-FNO}}$), as described below.

In Π_{Reveal} , parties begin with secret-shares of a value $[x]$ and end with the value x .

In Π_{Comp} , parties have shares of two values x and y , and as output they receive shares of a bit denoting the result of a comparison operator $[x > y]$, $[x \geq y]$ or $[x == y]$. These comparisons can be computed by converting to shares of bits and applying garbled circuits, which requires at least $O(W \log W)$ boolean gates on words of W bits. A promising alternative is the GMW-based approach of Nishide and Ohta [22], which requires $O(W)$ bits of communication and $O(1)$ rounds of communication and is concretely efficient.

In Π_{Sel} , parties perform multiplication of two values $[b]$ and $[x]$, where b is either 0 or 1, and x can be any value. Equivalently, parties compute shares of the ternary operator $b ? x : 0$, where b is either XOR shared or additively shared over a larger field. Π_{Sel} can be realized using MPC multiplication or via string-OTs with rerandomizable encryption.

In Π_{Shuffle} , parties begin with shares of a list, and end with shares of the same list, in a new (unknown) order. To get the desired asymptotics, we require that the shuffle have $O(1)$ rounds and $O(n)$ total communication. Such protocols exist and can be realized via an additively homomorphic, semantically secure cryptosystem with constant ciphertext expansion. At a high level, the shuffle works by allowing each party to hold an encryption of the list under the other party’s secret key, and then shuffling and re-randomizing; see [11] for the full protocol and a more thorough treatment.

In Π_{Dup} , parties begin with shares of a list of size n in which exactly k elements are marked with a given “tag” value (the tag value and the number of elements k that have this value are both public knowledge). After this protocol, each element with the specified tag value has been duplicated m times (for arbitrary public integer m), so that the final list size is $n + km$. We demonstrate in the full version how Π_{Dup} can be instantiated via two invocations of Π_{Shuffle} and n invocations of Π_{Comp} .

3.5 Security Model

We provide security analyses of our protocols within the Universally Composable (UC) framework, against a semi-honest adversary corrupting one of the two parties. UC security is essential since our protocols are built on recursive calls to sub-protocols. We present the protocol in the input setting where each party holds shares of each list being sorted, although, as mentioned in the introduction, the protocol can be adapted to other, more specific settings. We remark that the adversary does not have direct access to the memory access pattern of the other party. However, outside of the shuffling protocol, both parties have identical memory access patterns, and so the adversary can deduce most of the memory accesses of the honest party, and our proofs of security show the adversary learns nothing from these memory accesses.

We prove UC security of the protocols in this paper against static semi-honest adversaries under the standard simulator definition of security, see e.g. [5]. It is straightforward to simulate the behavior of the adversary during the protocol in any environment, since the adversary is semi-honest and must follow the protocol. What remains to be checked is the behavior of the adversary on the input, i.e. that the adversary input is still extractable without the simulator being able to “rewind” the adversary. We address this in the standard way by requiring both parties to commit to their inputs under an extractable commitment, so that the adversary input can be recovered without rewinding. We omit the details.

An ideal functionality \mathcal{F} for each protocol interacts with the parties in the following way:

- **Setup.** Each party sends their inputs to \mathcal{F} , who stores them plus an id sid .
- **Execution.** Each party sends the command $(\text{Execute}, sid)$ to \mathcal{F} , who computes the desired output and stores it.
- **Reveal.** Each party sends the command (Reveal, sid) to \mathcal{F} , who sends the output of the execute step to each party.

Note that we could instead combine multiple ideal functionalities into a black-box functionality $\mathcal{F}^{\text{black-box}}$ with a family of execution commands (Execute_i) corresponding to each of the protocols defined in this paper. This provides an alternate way to address composability, and guarantees that inputs to one protocol match outputs from another protocol.

3.6 Notation

For any sorted lists L_1 and L_2 , let \sqcup denote the “merge” of two lists (i.e. \sqcup is functionally equivalent to (multi-)set union followed by sort): $L_1 \sqcup L_2 = \text{Sort}(L_1 \cup L_2)$. For any sorted list L_j of size n , and for any $k|n$, let $\mathcal{M}_{j,k}$ denote the k “medians” of L_j . Namely, if list $L_j = \{u_1, \dots, u_n\}$, then: $\mathcal{M}_{j,k} := \{u_{\frac{\ell \cdot n}{k}}\}_{\ell=1}^k$. Basic properties that follow from the definition of medians can be found in the full version.

Throughout the paper we distinguish between secure *symmetric* merge Π_{SSM} and secure *asymmetric* merge Π_{SAM} . For symmetric merge, the lists are of roughly the same size or differ by a constant factor (since one list can be padded with dummies to match the length of the other). For asymmetric merge, the ratio of list sizes is larger than constant.

4 Description of Secure Merge Protocols

Our merge protocols come in two variants: symmetric merge, where the two lists are of equal size, and asymmetric merge, where one list is significantly smaller than the other. One useful technique that we employ in both settings is using black box calls to one flavor of merge to solve the other. This iterative process then terminates with a “base” version of each variant (symmetric and asymmetric merge), and we introduce in §6 two efficient protocols – $\Pi_{\text{SSM-log log}}$ and $\Pi_{\text{SAM-}n^{1/3}}$ – that can be used as the base protocols, allowing us to achieve our overall target metrics for secure merge as stated in Theorem 1. We summarize the dependency of our main protocols on these subprotocols in Table 2. That table, together with the asserted metrics of the primitive functionalities (such as Π_{Extract}), is sufficient to establish the stated asymptotics of Theorem 1.

In this section, we present our secure symmetric and asymmetric merge protocols. Each follows the *partition-align* approach, which has four phases:

- **Partition.** We invoke a subprotocol to determine where the partition points (medians) of one list lie within the other list (see e.g. Figure 1).
- **Align Blocks.** A (lightweight) MPC protocol is run to determine, for each block in one list, which block(s) in the other list span the same range of values (see e.g. Figure 2).

■ **Table 2** Protocol and subprotocol relations. The Cost column uses:

- For $\Pi_{\text{SSM}}(n)$: Set $k = \frac{n}{\log \log n}$, $\Pi_{\text{SAM}}(k, n) = \text{Fig. 5}$, $\Pi_{\text{SSM}}(\frac{n}{k}) = \Pi_{\text{SM-FNO}}$.
- For $\Pi_{\text{SAM}}(k, n)$: $\Pi_{\text{SSM}}(k) = \Pi_{\text{SSM-log log}}(k)$ (Fig. 7), $\Pi_{\text{SSM}}(\frac{n}{k}) = \Pi_{\text{SM-FNO}}$.

| Name | Calls to Subprotocols | Cost |
|--|---|------------------------|
| $\Pi_{\text{SSM}}(n)$ [Fig. 4] | $2 \cdot \Pi_{\text{SAM}}(k, n), 2k \cdot \Pi_{\text{SSM}}(n/k)$ | $O(n)$ |
| $\Pi_{\text{SAM}}(k, n)$ [Fig. 5] | $2 \cdot \Pi_{\text{SSM}}(k), k \cdot \Pi_{\text{SSM}}(n/k)$ | $O(n + k \log \log k)$ |
| $\Pi_{\text{SSM-log log}}(n)$ [Fig. 7] | $O(\log \log n) \cdot \Pi_{\text{Shuf}}(n), O(n) \cdot \Pi_{\text{Rev,Comp,Sel}}$ | $O(n \log \log n)$ |
| $\Pi_{\text{SAM-n}^{1/3}}(n^{1/3}, n)$ [Fig. 9 (full version)] | $2n^{1/3} \cdot \Pi_{\text{SM-ALL}}(n^{1/3}, n^{1/3})$ | $O(n)$ |

- **Merge Blocks.** This phase involves the merging of both the “well-aligned” blocks and the “poorly-aligned” blocks (see e.g. Figure 3 for well-aligned blocks).
- **Combine Blocks.** The results from the previous step are combined, and any dummy elements that were added are removed, producing the final merged list.

4.1 Secure Symmetric Merge

Our *Secure Symmetric Merge* protocol $\Pi_{\text{SSM}}(n)$ is sketched in Figure 4. For the Partition phase, it uses a secure asymmetric merge protocol to merge the k -medians of one list into the other list (and vice-versa). For the Align Blocks phase, we expand each median (in its merged location within the other list) into k -copies; which by Lemma 2 guarantees that each block of n/k elements is perfectly aligned (see Figure 3). Thus, there are no “poorly-aligned” blocks for the Merge Blocks phase, and each block is merged via a secure symmetric merge protocol (for lists of size n/k). Specification of our “abstract” secure symmetric merge protocol is presented in Figure 4; this protocol is made concrete by specifying choices for partition/block size k and the specific merge sub-protocols used in the Partition and Merge Blocks phases. In particular, the metrics of Theorem 1 are obtained by setting $k = n / \log \log n$ and using $\Pi_{\text{SSM}'}(n/k) = \Pi_{\text{SSM-FNO}}$ and $\Pi_{\text{SAM}}(k, n) = \Pi_{\text{SAM}}(k, n, \Pi_{\text{SSM-FNO}}, \Pi_{\text{SSM-log log}})$, which refers to the secure asymmetric merge protocol of Figure 5, using for its subprotocols $\Pi_{\text{SSM}'}(n/k) = \Pi_{\text{SSM-log log}}$ and $\Pi_{\text{SSM}''}(k) = \Pi_{\text{SSM-FNO}}$.

Secure Symmetric Merge Protocol

Input. Two parties $\mathcal{P}_1, \mathcal{P}_2$ (additively) secret-share two *sorted* lists L_1 and L_2 , each of size n . Also as input, a parameter k with $k|n$, and specifications of subprotocols $\Pi_{\text{SSM}'}(n/k)$ and $\Pi_{\text{SAM}}(k, n)$.

Output. The two lists have been merged into an output list $L_1 \sqcup L_2$, which has size $2n$ and is (additively) secret-shared amongst the two parties.

Protocol (sketch).

1. **Partition.** Invoke secure asymmetric merge protocol $\Pi_{\text{SAM}}(k, n)$ (Fig. 5) to merge the k medians of L_2 with list L_1 (and vice-versa).
2. **Align Blocks.** Expand Medians (Fig. 3) by running the duplicate values Π_{Dup} protocol twice, which expands the sizes of the output lists from Step 1 to be $2n$ and ensures they are “aligned”.
3. **Merge Blocks.** Run the secure symmetric merge protocol $\Pi_{\text{SSM}'}(n/k)$, in parallel, on each of the $2k$ (aligned) blocks.
4. **Combine Blocks.** Concatenate the results of the $2k$ parallel invocations of secure symmetric merge from the previous step, and run the secure ordered extract $\Pi_{\text{Ext-Ord}}$ protocol to remove dummy elements.

■ **Figure 4** Overview of our Secure Symmetric Merge Protocol. For the top-level protocol, use $k = n / \log \log n$, and use $\Pi_{\text{SAM}}(k, n, \Pi_{\text{SSM-FNO}}, \Pi_{\text{SSM-log log}})$ for the asymmetric merge protocol in Step 1, and use $\Pi_{\text{SSM-FNO}}$ for each of the symmetric merge protocols run in parallel in Step 3.

4.2 Secure Asymmetric Merge

Our *Secure Asymmetric Merge* protocol $\Pi_{\text{SAM}}(k, n)$ is sketched in Figure 5. For the Partition phase, it uses a secure symmetric merge protocol to merge the k medians of the larger list with the (entirety of the) smaller list. For the Align Blocks phase, we use the $\Pi_{\text{Ext-Bin}}$ protocol (§3.3) to securely classify elements/blocks as poorly-aligned vs. well-aligned. For the Merge Blocks phase: for the well-aligned items, we apply a secure symmetric merge protocol to merge (at most) n/k elements from L_1 into the appropriate block of n/k elements of L_2 . For the poorly-aligned items, we first argue that the cumulative number of elements in L_2 that lie in a poorly-aligned block is at most k (and the same is trivially true for L_1 , which only has k elements), and therefore we use another secure symmetric merge protocol (for lists of size k) to merge poorly-aligned elements from L_1 into poorly-aligned blocks of L_2 .

The “abstract” secure asymmetric merge protocol in Figure 5 is made concrete by specifying choices for the specific merge sub-protocols used in the Partition and Merge Blocks phases. In particular, the metrics of Theorem 1 are obtained by using $\Pi_{\text{SSM}'}(n/k) = \Pi_{\text{SSM-FNO}}$ and $\Pi_{\text{SSM}''}(k) = \Pi_{\text{SSM-log log}}$.

5 Analyses of Protocols in Section 4

In analyzing the performance of a protocol, RCost will denote the round-complexity, and CCost the computation and overall communication complexity.

5.1 Analysis of Abstract Secure Symmetric Merge Protocol of §4.1

Security. The security of the $\Pi_{\text{SSM}}(n)$ protocol follows immediately from the security of the underlying Π_{Dup} , $\Pi_{\text{Ext-Ord}}$, $\Pi_{\text{SSM}'}$, and Π_{SAM} protocols.

Correctness. Assuming correctness of $\Pi_{\text{SSM}'}(n/k)$, $\Pi_{\text{SAM}}(k, n)$, Π_{Dup} , and $\Pi_{\text{Ext-Ord}}$ subprotocols, we need only demonstrate that the concatenation done in Step 4 above is correct, that is, that the blocks “align” as per the partitioning. Namely, this follows from Lemma 2, which demonstrates that lists L_1' and L_2' have the same list of $2k$ medians (both equal $\mathcal{M}_{1,k} \sqcup \mathcal{M}_{2,k}$).

Cost. Step (1) invokes the $\Pi_{\text{SAM}}(k, n)$ protocol (Figure 5) twice; Step (2) invokes the $\Pi_{\text{Dup}}(k, n/k)$ protocol twice; Step (3) invokes the $\Pi_{\text{SSM}'}(n/k)$ protocol $2k$ times; and Step (4) invokes the secure ordered extract $\Pi_{\text{Ext-Ord}}(4n, 2n)$ protocol. Using the $\Pi_{\text{Dup}}(k, n/k)$ and the $\Pi_{\text{Ext-Ord}}(4n, 2n)$ protocols, and assuming constant-round and linear secure protocols for Π_{Comp} , Π_{Reveal} , and Π_{Shuffle} , adding up these costs yields:

$$\begin{aligned} \text{RCost}(\Pi_{\text{SSM}}(n)) &= \text{RCost}(\Pi_{\text{SAM}}(k, n)) + \text{RCost}(\Pi_{\text{Dup}}(k, n/k)) + \\ &\quad \text{RCost}(\Pi_{\text{SSM}'}(n/k)) + \text{RCost}(\Pi_{\text{Ext-Ord}}(4n, 2n)) \\ &= O(1) + \text{RCost}(\Pi_{\text{SAM}}(k, n)) + \text{RCost}(\Pi_{\text{SSM}'}(n/k)) \\ \text{CCost}(\Pi_{\text{SSM}}(n)) &= 2 \cdot \text{CCost}(\Pi_{\text{SAM}}(k, n)) + 2 \cdot \text{CCost}(\Pi_{\text{Dup}}(k, n/k)) + \\ &\quad 2k \cdot \text{CCost}(\Pi_{\text{SSM}'}(n/k)) + \text{CCost}(\Pi_{\text{Ext-Ord}}(4n, 2n)) \\ &= 2 \cdot \text{CCost}(\Pi_{\text{SAM}}(k, n)) + 2k \cdot \text{CCost}(\Pi_{\text{SSM}'}(n/k)) + O(n) \end{aligned}$$

Using $\Pi_{\text{SSM-FNO}}$ for $\Pi_{\text{SSM}'}(n/k)$, and using for $\Pi_{\text{SAM}}(k, n)$ our protocol of Fig. 5 – with subprotocols $\Pi_{\text{SSM-FNO}}$ for $\Pi_{\text{SSM}'}(n/k)$ and $\Pi_{\text{SSM-log log}}$ for $\Pi_{\text{SSM}''}(k)$ – and using $k = n/\log \log n$, the cost becomes:

Secure Asymmetric Merge Protocol $\Pi_{\text{SAM}}(k, n)$

Input. Two parties $\mathcal{P}_1, \mathcal{P}_2$ (additively) secret-share *sorted* list L_1 of size k and L_2 of size n . Also, specification of subprotocols $\Pi_{\text{SSM}'}(n/k)$ and $\Pi_{\text{SSM}''}(k)$.

Output. The two lists have been merged into an output list $L_1 \sqcup L_2$, which has size $k + n$ and is (additively) secret-shared amongst the two parties.

Protocol.

1. **Partition**(Fig. 1). Run $\Pi_{\text{SSM}''}(L_1, \mathcal{M}_{2,k})$ to merge L_1 and the k medians of L_2 .
2. a. **Align Blocks: Label**(Fig. 2). Partition L_2 into k blocks, each of size n/k (the k medians of L_2 define the right-boundary of each block). Define a block of L_2 to be “well-aligned” if it has fewer than n/k elements of L_1 that map to it. Well-aligned blocks are identified by doing (in parallel, via Π_{Comp}) a linear scan of the merge positions of L_1 (from Step 1), and comparing whether elements i and $(i + n/k + 1)$ are in the same block. Meanwhile, define elements of L_1 to be “well-aligned” if they lie in a well-aligned block.
 - b. **Align Blocks: Extract Poorly-Aligned**(Fig. 2). Using the merge positions from the previous step, run the $\Pi_{\text{Ext-Ord}}$ protocol on L_1 (respectively on L_2) to extract all “poorly-aligned” (i.e. *not* well-aligned) elements of L_1 (resp. L_2), and let L_1^{PA} (resp. L_2^{PA}) denote the extracted elements. Note that there are at most k poorly-aligned elements from each list (Observation 4), so L_1^{PA} and L_2^{PA} will each have size k , with $\Pi_{\text{Ext-Ord}}$ extracting dummy-elements to pad each list to exactly this size.
 - c. **Align Blocks: Extract Well-Aligned**(Fig. 2). For the well-aligned elements of L_1 , run the $\Pi_{\text{Ext-Bin}}$ protocol to extract the well-aligned elements of L_1 into the separate lists (each of size n/k), based on which block of L_2 they lie in (this information is available from the merge done in Step 1). By definition of “well-aligned,” there are at most n/k such elements for each block, and $\Pi_{\text{Ext-Bin}}$ extracts exactly this many elements into each list, padding with dummy elements when necessary. Let $\{L_{1,m}^{WA}\}_m$ denote the k output lists of the $\Pi_{\text{Ext-Bin}}$ protocol. Meanwhile, for L_2 , we simply extract all elements of the i^{th} block into $L_{2,i}^{WA}$ if and only if block i is well-aligned (otherwise $L_{2,i}^{WA}$ is filled with dummy elements). Using the labels created in Step 2, this can be done by running the $\Pi_{\text{Ext-Ord}}$ protocol on each block of L_2 .
3. a. **Merge Blocks: Poorly-Aligned.** Run $\Pi_{\text{SSM}''}(k)$ on L_1^{PA} and L_2^{PA} .
 - b. **Merge Blocks: Well-Aligned.** Run $\Pi_{\text{SSM}'}(n/k)$, in parallel, on each of the k (aligned) blocks $\{L_{1,m}^{WA}\}_m$ and $\{L_{2,m}^{WA}\}_m$.
4. **Combine Blocks.** The final index of any element in the merged list is:

$$(\#Left_1^{WA}) + (\#Left_1^{PA}) + (\#Left_2^{WA}) + (\#Left_2^{PA}) + \text{Block_Index}$$

where e.g. $\#Left_1^{WA}$ denotes the number of well-aligned elements of L_1 that lie in a block to the left of this element’s block, and Block_Index is this element’s index within its block. Each element’s final index is computed from the outputs of the merges done in Steps 1, 3a, and 3b, as per the analysis in the Correctness argument (Section 5.2).

■ **Figure 5** Secure Asymmetric Merge Protocol. In our main protocol, we take $k = n/\log \log n$, $\Pi_{\text{SSM}'} := \Pi_{\text{SSM-FNO}}$, and $\Pi_{\text{SSM}''} := \Pi_{\text{SSM-log log}}$.

$$\begin{aligned} \text{RCost}(\Pi_{\text{SSM}}(n)) &= [\text{RCost}(\Pi_{\text{SSM-log log}}(k) + \text{RCost}(\Pi_{\text{SSM-FNO}}(n/k))] + \\ &\quad \text{RCost}(\Pi_{\text{SSM-FNO}}(n/k)) \\ &= O(\log \log k) + O(n/k) = O(\log \log n). \\ \text{CCost}(\Pi_{\text{SSM}}(n)) &= 2 \cdot [2 \cdot \text{CCost}(\Pi_{\text{SSM-log log}}(k) + k \cdot \text{CCost}(\Pi_{\text{SSM-FNO}}(n/k))] + \\ &\quad 2k \cdot \text{CCost}(\Pi_{\text{SSM-FNO}}(n/k)) + O(n) \\ &= [O(k \log \log k) + k \cdot O(n/k)] + 2k \cdot O(n/k) + O(n) = O(n). \end{aligned}$$

5.2 Analysis of Abstract Secure Asymmetric Merge Protocol of §4.2

For all steps in this subsection, we refer to Figure 5.

Security. For Steps 1, 2a, 2b, 2c, 3a, and 3b: security follows from the security of the underlying subprotocols. Namely, a simulator for the protocol for either party calls the simulator for each subprotocol: $\Pi_{\text{SSM}'}$, Π_{Comp} , $\Pi_{\text{Ext-Ord}}$, $\Pi_{\text{Ext-Bin}}$, $\Pi_{\text{SSM}''}$, and $\Pi_{\text{SSM}'}$, respectively. Meanwhile, the correctness property ensures that the indices revealed in Step 4 are unique and are a (random) permutation of the values in $[1, \dots, (k+n)]$. Therefore, the simulator for Step 4 generates a random permutation of $(k+n)$ elements.

Correctness. We first clarify that the merges done in Steps 1, 3a, and 3b are done “in-place”: rather than actually *constructing* an output merged list, these merges instead determine each element’s *index* in what would be the merged list, and then append (shares of) this index as a tag applied to the element (in its original list). In this way, we may manipulate (add, subtract, etc.) the indices produced by the merges in Steps 1, 3a, and 3b, to compute each element’s index in the final merged list based on its indices in the outputs of the earlier “in-place” merges.

To verify correctness, it will be useful to set notation corresponding to the formula in Step 4 of Figure 5 as follows:

$$\begin{aligned} (U, V) &= (\#Left_1^{WA}, \#Left_1^{PA}): (\#(L1.WA), \#(L1.PA)) \text{ in all blocks to the left} \\ (W, X) &= (\#Left_2^{WA}, \#Left_2^{PA}): (\#(L2.WA), \#(L2.PA)) \text{ in all blocks to the left} \\ (Y, Z) &= (\#Same_1, \#Same_2): (\#L_1, \#L_2) \text{ in same block but to the left} \end{aligned}$$

where $Block_Index = Y + Z$ denotes the index of an element in its own block. Thus, any element’s final index in the merged list is: $U + V + W + X + Y + Z$. This quantity can be computed for each element based on information available from Steps 1-3, as follows:

$L_1.WA$: $(U + V + Y)$ is available from each element’s original position in L_1 ; $(W + X)$ is available as $j \cdot n/k$, where j is the block of L_2 that this element maps to (from Step 1); $(Y + Z)$ is the output index from Step 3b; and $-Y$ is this element’s position after merging its well-aligned block (Step 2c).

$L_2.WA$: $(U + V)$ is available from Step 2a; $(W + X)$ is available as $j \cdot n/k$, where j is the block of L_2 that this element lies in; and $(Y + Z)$ is the output index from Step 3b.

$L_1.PA$: $(U + V + Y)$ is available from each element’s original position in L_1 ; $(W + X)$ is available as $j \cdot n/k$, where j is the block of L_2 that this element maps to (from Step 1); $(V + X + Y + Z)$ is the output index from Step 3a; $(-V - Y)$ is from Step 2b; and $-X$ is n/k times the number of poorly-aligned blocks to the left, which is computable from the info in Step 2a.

$L_2.PA$: $(U + V)$ is available from Step 2a, while $-V$ is available by using the information from Step 1 applied just to the poorly-aligned items extracted in Step 2b; $(W + X)$ is available as $j \cdot n/k$, where j is the block of L_2 that this element lies in; $(V + X + Y + Z)$ is the output index from Step 3a; and $-X$ is n/k times the number of poorly-aligned blocks to the left, which is computable from the info in Step 2a.

It remains to show that the pre-conditions of the $\Pi_{\text{Ext-Ord}}$ and $\Pi_{\text{Ext-Bin}}$ protocols are satisfied. Namely, how to (efficiently) construct the input parameters C and T' of the $\Pi_{\text{Ext-Bin}}$ protocol; and that at most n/k elements are extracted from each list into each well-aligned block and at most k elements are extracted from each list into the poorly-aligned block. These are stated and proved in the following observations:

► **Observation 3.** (*Shares of*) the parameters C and T' to the $\Pi_{\text{Ext-Bin}}$ protocol used in Step 2c can be securely computed locally in $O(n)$ computation.

Proof. For each $i \in [1..k]$, let ι_i denote the position of the k^{th} median of L_2 in the output list of Step 1; and let δ_i be an indicator on whether $t_i > 0$. Then formulas for $C = \{c_j\}$ can be expressed iteratively as: $c_1 = \delta_1$ and $c_{j+1} = \delta_{j+1} \cdot (1 + \sum_{i \leq j} t'_i)$; and for $T' = \{t'_j\}$ as: $t'_1 = \iota_1 - 1$ and $t'_{j+1} = \iota_{j+1} - j - \sum_{i \leq j} t'_i$. While these expressions can be computed securely without introducing additional overhead to the overall protocol, the multiplication by δ_i in the expression of c_{j+1} would require a secure (non-local) computation. However, we leverage the fact that the $\Pi_{\text{Ext-Bin}}$ protocol (see full version) works correctly even if c_j is an arbitrary value when $t'_j = 0$. Thus, the δ_i terms can be dropped from the above expression, and then, by linearity of the secret-sharing scheme, each party can locally compute their shares of C and T' by using their shares of the relevant variables on the RHS of each of the above expressions for t'_j and c_j . \blacktriangleleft

► **Observation 4.** For both L_1 and L_2 , the number of poorly-aligned elements in Step 2b of the $\Pi_{\text{SAM}}(k, n)$ protocol of Figure 5 is at most k .

Proof. Since L_1 has size k , this statement is trivially true for L_1 . Meanwhile, for each poorly-aligned segment of L_2 , by definition there are at least $n/k + 1$ elements from L_1 that are assigned this segment. Thus, the k elements of L_1 can cause at most $m = k/(n/k + 1) < k^2/n$ poorly-aligned segments. Since each segment has size n/k , this corresponds to at most $m \cdot n/k < k$ elements of L_2 . \blacktriangleleft

► **Observation 5.** For both L_1 and L_2 , the number of well-aligned elements in each block of Step 2c of the $\Pi_{\text{SAM}}(k, n)$ protocol of Figure 5 is at most n/k .

Proof. Blocks are defined as the n/k elements to the left of (and including) each (of the k) median of L_2 , so this is trivially true for L_2 . Meanwhile, for L_1 , any block that contains more than n/k elements of L_1 will be a poorly-aligned block, and consequently the corresponding elements from L_1 will all be labelled “poorly-aligned,” which means *no* elements from L_1 will be extracted by $\Pi_{\text{Ext-Bin}}$ in Step 2c for this block. \blacktriangleleft

Cost.

- Step (1) has $\text{RCost}(\Pi_{\text{SSM}''}(k))$ and $\text{CCost}(\Pi_{\text{SSM}''}(k))$.
- Step (2a) has $\text{RCost}(\Pi_{\text{Comp}})$ and $\Theta(k) \cdot \text{CCost}(\Pi_{\text{Comp}})$.
- Step (2b) has $\text{RCost}(\Pi_{\text{Ext-Ord}}(k, k)) + \text{RCost}(\Pi_{\text{Ext-Ord}}(n, k))$ and $\text{CCost}(\Pi_{\text{Ext-Ord}}(k, k)) + \text{CCost}(\Pi_{\text{Ext-Ord}}(n, k))$.
- Step (2c) has $\text{RCost}(\Pi_{\text{Ext-Bin}}(k, k, n/k)) + k \cdot \text{RCost}(\Pi_{\text{Ext-Ord}}(n/k, n/k))$ and $\text{CCost}(\Pi_{\text{Ext-Bin}}(k, k, n/k)) + k \cdot \text{CCost}(\Pi_{\text{Ext-Ord}}(n/k, n/k))$.
- Step (3a) has $\text{RCost}(\Pi_{\text{SSM}''}(k))$ and $\text{CCost}(\Pi_{\text{SSM}''}(k))$.
- Step (3b) has $\text{RCost}(\Pi_{\text{SSM}'}(n/k))$ and $k \cdot \text{CCost}(\Pi_{\text{SSM}'}(n/k))$.
- Step (4) has $\text{RCost}(\Pi_{\text{Extract}}(2(k+n), k+n)) + \text{RCost}(\Pi_{\text{Reveal}})$ and $\text{CCost}(\Pi_{\text{Extract}}(2(k+n), k+n)) + (k+n) \cdot \text{CCost}(\Pi_{\text{Reveal}})$.

Using the costs of subprotocols Π_{Extract} , $\Pi_{\text{Ext-Ord}}$, and $\Pi_{\text{Ext-Bin}}$, and assuming constant-round and linear secure protocols for Π_{Comp} , Π_{Reveal} , and Π_{Shuffle} , we add up the contributions from each step to obtain:

$$\begin{aligned}
\text{RCost}(\Pi_{\text{SAM}}(k, n)) &= \text{RCost}(\Pi_{\text{SSM}''}(k)) + \text{RCost}(\Pi_{\text{SSM}'}(n/k)) + \\
&\quad \text{RCost}(\Pi_{\text{Comp}}) + \text{RCost}(\Pi_{\text{Reveal}}) + \\
&\quad \text{RCost}(\Pi_{\text{Ext-Ord}}(k, k)) + \text{RCost}(\Pi_{\text{Ext-Ord}}(n, k)) \\
&\quad \text{RCost}(\Pi_{\text{Ext-Bin}}(k, k, n/k)) + \text{RCost}(\Pi_{\text{Ext-Ord}}(n/k, n/k)) \\
&\quad \text{RCost}(\Pi_{\text{Extract}}(2(k+n), k+n)) \\
&= O(1) + \text{RCost}(\Pi_{\text{SSM}''}(k)) + \text{RCost}(\Pi_{\text{SSM}'}(n/k)) \\
\text{CCost}(\Pi_{\text{SAM}}(k, n)) &= 2 \cdot \text{CCost}(\Pi_{\text{SSM}''}(k)) + k \cdot \text{CCost}(\Pi_{\text{SSM}'}(n/k)) + \\
&\quad \Theta(k) \cdot \text{CCost}(\Pi_{\text{Comp}}) + (k+n) \cdot \text{CCost}(\Pi_{\text{Reveal}}) + \\
&\quad \text{CCost}(\Pi_{\text{Ext-Ord}}(k, k)) + \text{CCost}(\Pi_{\text{Ext-Ord}}(n, k)) \\
&\quad \text{CCost}(\Pi_{\text{Ext-Bin}}(k, k, n/k)) + k \cdot \text{CCost}(\Pi_{\text{Ext-Ord}}(n/k, n/k)) \\
&\quad \text{CCost}(\Pi_{\text{Extract}}(2(k+n), k+n)) \\
&= O(k+n) + 2 \cdot \text{CCost}(\Pi_{\text{SSM}''}(k)) + k \cdot \text{CCost}(\Pi_{\text{SSM}'}(n/k))
\end{aligned}$$

Using $\Pi_{\text{SAM-FNO}}$ for $\Pi_{\text{SSM}'}(n/k)$ and $\Pi_{\text{SSM-log log}}$ for $\Pi_{\text{SSM}''}(k)$, the cost is:

$$\begin{aligned}
\text{RCost}(\Pi_{\text{SAM}}(k, n)) &= O(\log \log k) + O(n/k) = O(n/k + \log \log k) = O(\log \log n) \\
\text{CCost}(\Pi_{\text{SAM}}(k, n)) &= O(k+n) + O(k \log \log k) + O(n) = O(n + k \log \log k) = O(n)
\end{aligned}$$

where the final equality for costs comes as a result of setting $k = n / \log \log n$.

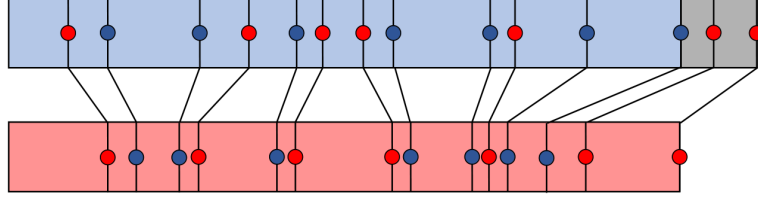
6 Description of Base Protocols: $\Pi_{\text{SSM-log log}}$ and $\Pi_{\text{SAM-n}^{1/3}}$

In this section, we present our $\Pi_{\text{SSM-log log}}$ and $\Pi_{\text{SAM-n}^\alpha}$ base protocols (for $\alpha = 1/3$; the general case for $\alpha < 1$ can be found in the full version).

6.1 Secure Symmetric Merge with $O(n \log \log n)$ Communication

As an ingredient for $\Pi_{\text{SAM}}(k, n)$, we need a secure symmetric merge with $O(n \log \log n)$ communication and $O(\log \log n)$ rounds, which we call $\Pi_{\text{SSM-log log}}$. We give this protocol in Figure 7. For the Partition phase, it uses a secure asymmetric merge protocol (namely, the $\Pi_{\text{SAM-n}^{1/3}}$ in the next section) to merge the $k = n^{1/3}$ medians of each list into the other list. The $2k$ medians of both lists thus partition each list into $2k$ blocks of size at most n/k . For the Align Blocks phase, we group each block from L_1 with the corresponding block from L_2 , as shown in Figure 6, and pad each block with dummies so that it has length exactly n/k . For the Merge Blocks phase, each pair of matching blocks are merged together; and then the Combine Blocks phase simply concatenates and removes dummy elements that were introduced to hide the alignment of blocks.

The protocol is recursive, and, if implemented naïvely, the problem sizes would double at each step of the recursion, for $O(n \log n)$ computation (instead of the desired $O(n \log \log n)$), because there are $\log \log n$ steps total. In Fig. 7, we show how to discard sub-problems along the way to avoid this blow-up.



■ **Figure 6** Each list is partitioned into $2k$ blocks by the k medians from both lists. The two grey blocks appended to the end of the top list are made up entirely of dummy elements, and show how to handle medians from one list that merge to a location outside the other list. Each block is padded with the appropriate number of dummy elements to ensure it has n/k elements. Each pair of aligned blocks (visualized via the connecting line segments) are then merged in $2k$ sub-problems (run in parallel) for $\Pi_{\text{SSM}}(n/k)$, with the final result obtained by concatenating the results of these $2k$ merges and removing dummy elements.

6.2 Analysis of the $\Pi_{\text{SSM}-\log \log(n)}$ Protocol of §6.1

Security. To simulate either player's view during an execution of the protocol, a simulator can call the simulator of the sub-protocols on every step except for Step 11, since that is the only step where values are revealed to the parties. For Step 11, the parties only see a random permutation of $\{1, 2, \dots, 2n\}$, so their views can be simulated by randomly sampling such a permutation.

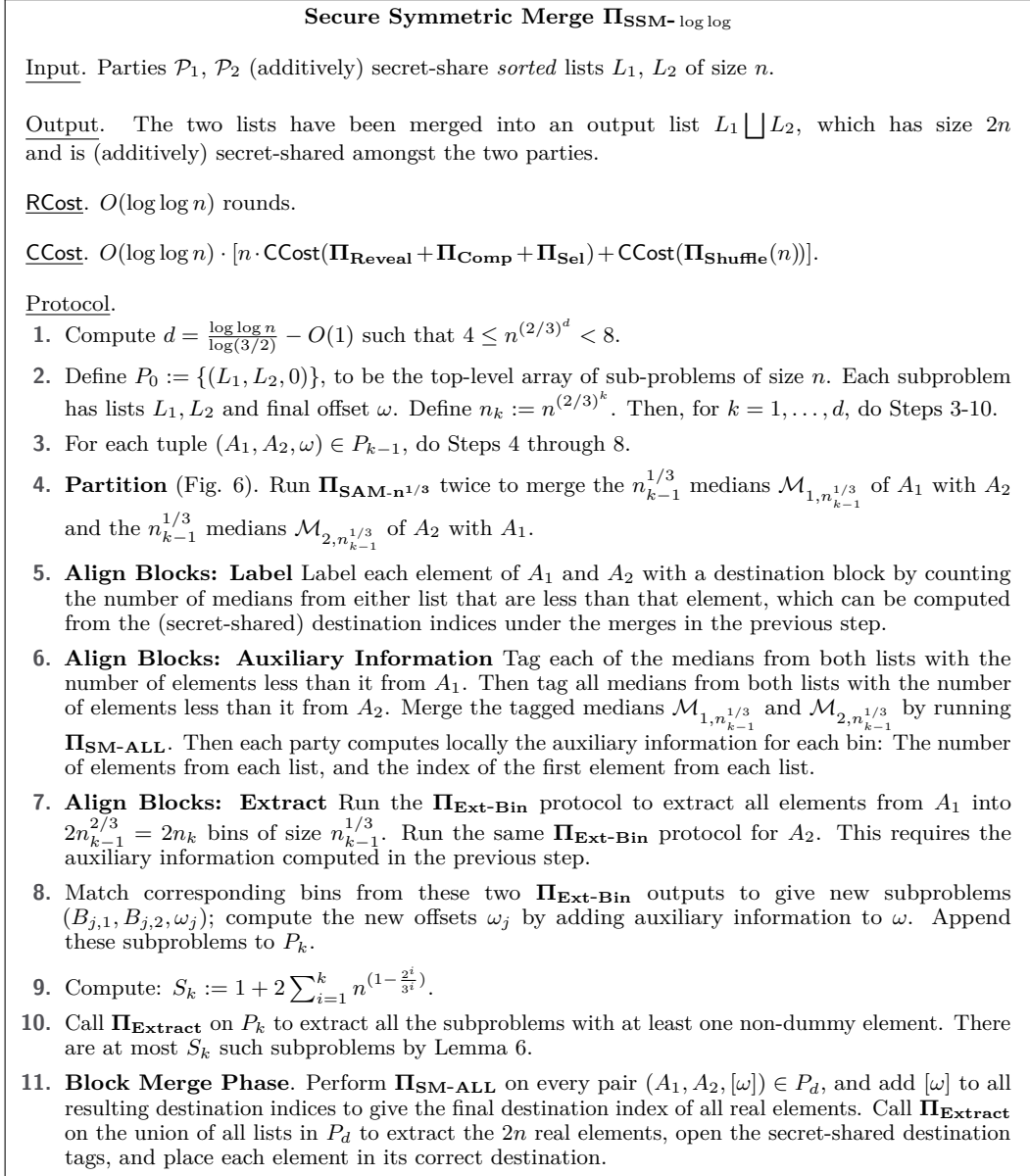
Correctness. The overall goal will be to show that the subproblems extracted in Step 10 contain a partition of the $2n$ elements into well-ordered blocks, so that the real elements of each block are either all greater than or all less than the real elements from another block. We must show this well-orderedness property and that all elements are included, so that it is truly a partition. We proceed step-by-step.

In Step 4, the $\Pi_{\text{SAM-}n^{1/3}}$ protocols are performed “in-place”, so that each element gets a (secret-shared) tag of their destination under this merge. These tags can be transformed into shares of the number of medians from the opposite list less than a given element, or shares of the number of elements from the opposite list less than a given median.

In Step 5, the shares of the destination block are computed by adding the shares of the number of medians of the opposite list less than a given element to the floor of the element's index divided by $n_{k-1}^{2/3}$, which counts the number of medians from the same list less than a given element, and is a public value.

In Step 6, again, the number of elements less than a given median from the opposite list is a secret shared value generated in Step 4, and the number of elements less than a given median in the same list is a public value. This time, when we run $\Pi_{\text{SM-ALL}}$, we do not run it “in-place”, but return the output list of tagged medians. Note that we tag all medians first with an A_1 -tag (counting elements from A_1 less than that median), then with an A_2 -tag, so that after merging, the A_1 medians and A_2 medians are indistinguishable, and we can compute the correct auxiliary information without leaking information. The index of the first element of A_1 (resp. A_2) in the j -th bin is the A_1 -tag (resp. A_2 -tag), since the first element in the j -th bin is the first element not less than the j -th median. The number of elements from A_1 (resp. A_2) in the j -th bin is equal to the $(j+1)$ -th A_1 -tag (resp. A_2 -tag) minus the j -th tag, since this is the difference between the start index of two adjacent bins.

For Step 7, we have just shown the correctness of the auxiliary information generated for $\Pi_{\text{Ext-Bin}}$. It remains to show that the assumed bound on the size of each bin holds. The union of the values from $\mathcal{M}_{1, n_{k-1}^{1/3}}$ and $\mathcal{M}_{2, n_{k-1}^{1/3}}$ will partition each list A_1 and A_2 into $2n_{k-1}^{1/3}$



■ **Figure 7** $O(n \log \log n)$ Secure Symmetric Merge Protocol

blocks (since the list of both medians includes the maximum of both lists, no non-median element can lie to the right of all medians, see Fig. 6). Each block will have at most n_k real elements, since each block is either composed of the interval between two adjacent medians of the same list, which has exactly n_k elements, or a sub-interval of such an interval.

In Step 8, we obtain the new offset ω_j by adding ω (the offset for the current subproblem) to the j -th A_1 -tag and the j -th A_2 tag. These two tags count the number of elements from A_1 and A_2 less than the j -th median, but we must verify that all of these elements are real. But all dummy elements are sorted to the right of all real-valued medians, so either the entire subproblem will be dummy, and the subproblem will be discarded in Step 10, or all elements counted by the A_1 -tag and the A_2 -tag are non-dummy, and our computation of ω_j is correct.

The value S_k and the bound on non-dummy subproblems in Step 10 follow from Lemma 6, below. The correctness of the final indices in Step 11 follows from the correctness of the offsets ω_j computed along the way and the correctness of $\Pi_{\text{SM-ALL}}$, and since we have verified that P_d contains a partition of the $2n$ real elements, the bound on real elements for the final Π_{Extract} call is also correct.

► **Lemma 6.** *At a depth of d , there are at most $S_d := 1 + 2 \sum_{i=1}^d n^{1-\frac{2^i}{3^i}}$ sub-problems without all elements dummy. For $d = \frac{\log \log n}{\log(3/2)} - O(1)$, $S_d \leq 4n^{1-(2/3)^d}$.*

Proof. Let N_k be the number of subproblems at depth k with at least one non-dummy element, that is, the cardinality of the set $|P_k|$ after the extraction in Step 10. Then we have $N_0 = 1$ and we will prove that $N_{k+1} \leq N_k + 2n^{1-(2/3)^k}$, which is sufficient to prove the lemma. To prove this recurrence relation, we will divide the subproblems generated before the extraction in Step 10 into three categories: A group of subproblems that have an average density of at least $1/2$ real elements, one special subproblem, and a group of subproblems made up entirely of dummy elements. Since subproblems at level k have $2n^{(2/3)^k}$ elements, and there are $2n$ real elements total, there can be at most $2n/n^{(2/3)^k} = 2n^{1-(2/3)^k}$ subproblems in the first category. There will also be, of course, N_k special subproblems, which will prove the recurrence. It remains to describe this categorization.

Let (A_1, A_2, ω) be a subproblem at level $k-1$, and let m_1 and m_2 be the number of median elements that are real in A_1 and A_2 respectively. Then the first $m_1 + m_2$ blocks (as determined in Step 5) will include at least the first $m_1 n^{(2/3)^k}$ real elements of A_1 and the first $m_2 n^{(2/3)^k}$ real elements of A_2 . Thus these $m_1 + m_2$ subproblems will have $2(m_1 + m_2)n^{(2/3)^k}$ elements and at least $(m_1 + m_2)n^{(2/3)^k}$ real elements, which proves that their average density is at least $1/2$. Because the next median of each list is a dummy element, all remaining real elements of both A_1 and A_2 will be mapped into the $m_1 + m_2 + 1$ -th block, which is the special subproblem.

For the final bound on S_d , note that there exists a unique value of d , with $d < \frac{\log \log n}{\log(3/2)}$, such that $4 \leq n^{(2/3)^d} < 8$. We thus have for $k < d$ the desired bound:

$$\frac{n^{1-(2/3)^k}}{n^{1-(2/3)^{k+1}}} = n^{-\frac{2^k}{3^{k+1}}} \leq \frac{1}{2} \quad \Rightarrow \quad S_d \leq 2n^{(2/3)^d} \sum_{i=0}^d \left(\frac{1}{2}\right)^{d-i} < 4n^{(2/3)^d}. \quad \blacktriangleleft$$

Cost. At a depth k , Steps 4 through 8 of Figure 7 require $O(n^{(2/3)^k})$ communication and computation and $O(1)$ rounds for each element of P_k , since the computations in each of Steps 4 through 8 are linear. By Lemma 6, at a depth of k , there are at most $S_k = 2n^{1-(2/3)^k} + O(n^{1-(2/3)^{k+1}}) = O(n^{1-(2/3)^k})$ subproblems, so the total work for Steps 4 through 8 is $O(n)$ communication and computation and $O(1)$ rounds. Steps 4 through 8 have the effect of doubling the total number of elements, so the total size of the lists in P_k before the extraction in Step 10 is $2S_k n^{1-(2/3)^k} = O(n)$, and so Step 10 requires $O(n)$ work each time it is called. Since Steps 3-10 are called $\log \log n - O(1)$ times, the total communication is $O(n \log \log n)$ and the round complexity is $O(\log \log n)$, as desired.

6.3 Secure Asymmetric Merge on $(n^{1/3}, n)$

For the special case where $|L_1| = O(n^{1/3})$ and $|L_2| = O(n)$, succinctly describe

an asymmetric merge protocol with communication complexity $O(n)$ and round complexity $O(1)$. By calling $\Pi_{\text{SM-ALL}}$ on L_1 and the $n^{2/3}$ medians of L_2 , we can identify every block of L_2 which contains an element of L_1 after merging the two lists. Because there are $O(n^{1/3})$

elements of L_1 , there are $O(n^{1/3})$ such blocks of L_2 . After extracting these blocks, we run $\Pi_{\text{SM-ALL}}$ again on L_1 and the $O(n^{2/3})$ elements of the extracted blocks of L_2 . After some careful accounting of the index shifts during these steps, we get the destination indices for all the elements, which gives the desired merge protocol. Due to space considerations, we refer the reader to Fig. 9 of the full version for details.

References

- 1 M. Ajtai, J. Komlós, and E. Szemerédi. An $o(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. Association for Computing Machinery. doi:10.1145/800061.808726.
- 2 Arvind Arasu and Raghav Kaushik. Oblivious query processing. In *Proc. 17th International Conference on Database Theory (ICDT)*, Athens, Greece, March 24–28, 2014, pages 26–37. OpenProceedings.org, 2014. doi:10.5441/002/icdt.2014.07.
- 3 K. E. Batchier. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. Association for Computing Machinery. doi:10.1145/1468075.1468121.
- 4 Allan Borodin and John E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *J. Comput. Syst. Sci.*, 30(1):130–145, 1985. doi:10.1016/0022-0000(85)90008-X.
- 5 Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1):143–202, 2000. doi:10.1007/s001459910006.
- 6 Suvradip Chakraborty, Stanislav Peceny, Srinivasan Raghuraman, and Peter Rindal. Logstar: Efficient linear* time secure merge. *IACR Cryptol. ePrint Arch.*, page 159, 2024. URL: <https://eprint.iacr.org/2024/159>.
- 7 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 *Advances in Cryptology – ASIACRYPT 2018*, volume 11274 of *Lecture Notes in Computer Science*, pages 158–188. Springer, 2018. doi:10.1007/978-3-030-03332-3_7.
- 8 Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. An efficient secure three-party sorting protocol with an honest majority. *IACR Cryptol. ePrint Arch.*, page 695, 2019. URL: <https://eprint.iacr.org/2019/695>.
- 9 Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, October 2019. doi:10.14778/3364324.3364331.
- 10 Brett Hemenway Falk, Rohit Nema, and Rafail Ostrovsky. A linear-time 2-party secure merge protocol. In *Cyber Security, Cryptology, and Machine Learning - 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30 - July 1, 2022, Proceedings*, volume 13301 of *Lecture Notes in Computer Science*, pages 408–427. Springer, 2022. doi:10.1007/978-3-031-07689-3_30.
- 11 Brett Hemenway Falk and Rafail Ostrovsky. Secure merge with $o(n \log \log n)$ secure operations. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*, volume 199 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:29, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITC.2021.7.
- 12 O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/28395.28420.
- 13 Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996. doi:10.1145/233551.233553.
- 14 Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. *IACR Cryptol. ePrint Arch.*, page 121, 2014. URL: <http://eprint.iacr.org/2014/121>.
- 15 Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *International*

- Conference on Information Security and Cryptology*, pages 202–216. Springer, 2013. doi:10.1007/978-3-642-37682-5_15.
- 16 Zhu Hong and Robert Sedgewick. Notes on merging networks (preliminary version). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 296–302, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/800070.802204.
 - 17 Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. URL: <https://www.ndss-symposium.org/ndss2012/private-set-intersection-are-garbled-circuits-better-custom-protocols>.
 - 18 Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *Topics in Cryptology - CT-RSA 2016*, pages 90–107, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-29485-8_6.
 - 19 Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *Advances in Cryptology – CRYPTO 2018*, pages 523–542, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96881-0_18.
 - 20 Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography*, pages 377–396, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-36594-2_22.
 - 21 Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer Publishing Company, Incorporated, 1 edition, 2008. doi:10.1007/978-3-540-77978-0.
 - 22 Takashi Nishide and Kazuo Ohta. Constant-round multiparty computation for interval test, equality test, and comparison. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 90-A(5):960–968, 2007. doi:10.1093/ietfec/e90-a.5.960.
 - 23 Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Advances in Cryptology – CRYPTO 2010*, pages 502–519, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-14623-7_27.
 - 24 Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2), January 2018. doi:10.1145/3154794.
 - 25 Leslie G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348–355, 1975. doi:10.1137/0204030.
 - 26 Andrew Chi-Chih Yao and Foong Frances Yao. Lower bounds on merging networks. *J. ACM*, 23(3):566–571, 1976. doi:10.1145/321958.321976.
 - 27 Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 283–298. USENIX Association, 2017. URL: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng>.