Linear-Time Secure Merge in $O(\text{loglog } n)$ Rounds

Mark Blunk[1], Paul Bunn[1], Samuel Dittmer[1], Steve Lu[1], and Rafail Ostrovsky[2]

[1]Stealth Software Technologies, Inc. {mark,paul,sam,steve}@stealthsoftwareinc.com
[2]UCLA Departments of Computer Science, Mathematics. rafail@cs.ucla.edu

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 the analogous separation exists for *secure* protocols; namely, if there exist techniques for performing *secure* merge that are more performant than simply invoking a secure multiparty sort.

We answer this question affirmatively by constructing a semi-honest 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 secure primitive do not, our result imply a computationally secure two-party secure merge protocol and an information-theoretically secure three-party secure merge protocol with these bounds.

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 – since 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 (where one list is much larger than the other).

Keywords. Secure Merge, Secure Sort, Secure Databases, Private Set Intersection.

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) [9, 24, 40] and Oblivious RAM (ORAM) [33, 38], 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 of the complexity of any secure computation, on top of the $\Theta(n \log n)$ cost of performing (insecure) sort. 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. 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 §4.4: comparison, shuffle, reveal, and conditional-addition.*

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, a and 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., [14]. While there are ways to extend our results to more parties beyond three, see for example the discussion in Remark 1, 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, 12, 26, 48].

1.1 Paper Structure

In §2 we summarize previous work on the secure sort/merge problem, and provide a comparison table of our main result with relevant earlier works in Table 1. In §3 we provide a survey of applications whose performance could be improved by the adoption of our secure merge protocol. In §4 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 proofs of their properties, in §5. In §6 we present the two underlying secure merge protocols required by our protocols in §5. Supplementary material with expanded discussion and additional protocols – including the description of a separate constant round secure merge protocol in the asymmetric case that is a generalization of the protocol $\mathbf{\Pi_{SAM\text{-}n^{1/3}}}$ of §6 (where one list has size n and the other n^α for fixed $\alpha < 1$) – can be found in Sections 7 - 10.

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; please 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 [4], 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 Szemeredi [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 [47], and depth of $\Omega(\log n)$, as shown by Hong and Sedgewick [23].

RAM and PRAM. The classical merge algorithm requires $O(n)$ work on a RAM machine, see e.g., [35]. As discussed below, the approach of [13] 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., [5] for more formal definitions and a discussion of various PRAM models.

For PRAM machines, Valiant showed in [45] 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 [5] 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 (§5) 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 [5] 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 §5 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.

Remark 1. A follow-up work by Hayashi, Nakano, and Olariu [22] demonstrates a protocol with $O(\log\log n + \log k)$ run-time for merging k lists of size $O(n)$. Following the intuition that run-time in the PRAM model roughly corresponds to (a lower-bound on) round-complexity in the secure setting, this suggests that a linear-time secure merge protocol for k parties could have round complexity $O(\log\log n + \log k)$. For example, this would imply that an optimal secure merge protocol for $k = O(\log n)$ parties (each with a list of size $O(n)$) could have linear run-time and $O(\log\log n)$ round complexity. However, because our protocols in §5 rely on black-box use of a constant-round secure shuffle subprotocol with $O(n)$ communication, and since currently known instantiations of secure shuffle with linear communication have *linear* or worse dependence on the number of parties (in terms of round-complexity) [14, 32], direct extension of our secure merge protocol to $k > 3$ parties would have $O(\log\log n + k)$ round complexity.

2.2 Secure Merge Algorithms

Notation. To compare prior work with our protocols as concretely as possible, 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 in our chosen 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 to realize than additive-only homomorphic schemes, 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 [4].

Additionally, 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 [17]) 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. We can also construct a $O(1)$ round protocol by having one party encrypt their inputs under fully homomorphic encryption (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 the FHE approach is only $O(\beta_{\text{HE}} n)$.

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 $\Omega(\log n)$, the FHE scheme 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 [18,31]. 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 this shuffling inherently destroys the property that the two input lists are already sorted, this approach reduces a secure *merge* problem to a secure *sort* problem, hence incurring the $\log n$ efficiency loss.

Comparison-based sorting requires $O(n \log n)$ communication and computation and $O(\log n)$ rounds, while secure shuffling requires $O(n)$ communication and $O(1)$ rounds, see e.g., [14, 21]. 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 secure merge approach on 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, Ikarashi, Chida, and Takahashi [20], 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 [9], 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 [14], achieves $O(n \log \log n)$ communication complexity with $O(\log n)$ round complexity. The second, due to Falk, Nema, Ostrovsky [13], 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 [13], while both simple and asymptotically optimal in terms of communication, may still not be practical due to high round complexity.

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., [14]). The oblivious sort protocols mentioned above [9, 20] 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 [13], 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).

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 [9, 21]), we can set $\beta_{\text{HE}} = \gamma = 1$ for the last four protocols of Table 1.

We also give 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, while requiring only $O(1)$ rounds.

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 [21]	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Oblivious Radix Sort [9]	$O(n \log n)$	$O(n \log n)$	$O(W)$
Secure Merge of [14]	$O(n \log\log n + \gamma n)$	$O(n \log\log n + \beta_{\text{HE}} n)$	$O(\log n)$
Secure Merge of [13]	$O(\gamma n)$	$O(\beta_{\text{HE}} n)$	$O(n)$
Our Secure $(\mathbf{n}^\alpha, \mathbf{n})$ 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 $(\mathbf{n}, \mathbf{n})$ Merge [Fig. 4]	$O(\gamma n)$	$O(\beta_{\text{HE}} n)$	$O(\log\log n)$

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).

3 Applications

We give here a brief survey of areas of cryptography where our secure merge protocol could be used to improve performance. Determining how competitive these secure merge-based improvements would be in practice depends on concrete choices about implementation, especially the choice of shuffling protocol and the point in the recursion where we switch to a simpler merging protocol, and is outside the scope of this work.

3.1 Private Set Intersection

Private Set Intersection (PSI), where two parties wish to learn the elements in the intersection of their two lists, or some function on the intersection, is one of the most widely used applications of multiparty computation to a specialized setting. We survey the results in this area, and then mention particular protocols where secure merge could improve performance.

Many of the approaches are designed by identifying powerful cryptographic machinery. Among public key encryption protocols, we mention the early protocol built off of Diffie-Hellman key agreement protocols first presented in Meadows [34]; see also [25, 41], recent advances including the OPRF approach of [15], the blind-RSA approach of Cristofaro and Tsudik [10], and the Bloom filter based approach first introduced in Debnath and Dutta [11]. Among the OT-based family of

approaches, we cite [39] and the follow-up work [40], as well as [29], which applied OPRF tools to the OT-based approach. As an example of the homomorphic encryption based approach, we mention [8], as well as [44], which specialized to the case of PSI-cardinality, and [27], which gives publicly verifiable PSI.

In contrast to these approaches, the *circuit-based* family of approaches aims to use generic tools of multiparty computation. The garbled circuit based approach of Huang, Evans, and Katz, [24] outperforms prominent public key based approaches and is more flexible than the OT-based approaches (because any generic computation on the outputs of the PSI protocol can be performed by simply appending additional gates to the merging network circuit and garbling). These comparisons were explored further in [40], which also benchmarked an implementation of a merging network under the GMW paradigm, and approaches using a naïve quadratic comparison-based sort after hashing elements into sufficiently small buckets. In the circuit-based family of approaches we also mention the earlier work of Jonsson, Kreitz, and Uddin [28] and an approach based on radix sort in the three-party with honest majority setting [9].

A PSI protocol could be built off of our merge protocol. As our merge protocol requires an additive homomorphic encryption system for shuffling but otherwise can be built from generic GMW machinery, this would situate a derived PSI protocol in between the bespoke protocols and the generic approaches. It can merge two sorted lists in time $O(n)$, unlike the Yao and GMW-based approaches which each require $O(n \log n)$ time, and shares with them the flexibility of being able to perform any generic MPC protocol on the outputs of the PSI protocol.

3.2 Oblivious RAM

Oblivious RAM, introduced by Goldreich and Ostrovsky [18], is a cryptographic data structure that behaves like a standard Random Access Machine, but disguises the locations of the encrypted memory being accessed from an adversary that can observe the sequence of data reads. This allows for the direct emulation of a RAM program in zero knowledge and multiparty computation settings, without requiring unrolling the program and converting to a circuit (see e.g., [33]).

The seminal work of Goldreich and Ostrovsky [18] established that any Oblivious RAM machine must have $\Omega(\log n)$ overhead for accessing a RAM of size n, and gave a construction with overhead $O(\log^3 n)$ that was unlikely to be practical on machines at the time. There has been substantial progress since then, both in closing the theoretical gap between upper and lower bound, and in bring Oblivious RAM towards the realm of practical applications.

Among the practical Oblivious RAM protocols, there are two primary families of approaches. The first was the hierarchical-based approach of [18], where the data is stored in a series of hash tables arranged "hierarchically", top-to-bottom, improved in a series of works, including in [19, 30, 38]. The second approach is the tree-based approach of [42], where data is stored in a binary tree that is updated as it is traversed. This approach inspired other lines of work, such as the related Path ORAM [43], Circuit ORAM [46], and the optimization [16]. In terms of asymptotic overhead, improvements over the years culminated in the two hierarchical-based protocols PanORAMa [37], with $O(\log n \log \log n)$ overhead, and finally OptORAMa [3], with the optimal $O(\log n)$ overhead.

The tree-based protocols often do not require oblivious sorting, but most of the hierarchical protocols do, and so could potentially be improved by the introduction of secure merge. Since OptORAMa achieves the lower bound for ORAM overhead, secure merge could not improve the asymptotics of ORAM constructions, but could improve the concrete constants of existing constructions, or improve the asymptotics of non-optimal constructions, perhaps leading to a simpler realization of optimal ORAM. The major obstacle to making progress in this way is that the elements being sorted in hierarchical ORAM are generally random elements that have been removed

into a stash or series of stashes, and so are guaranteed to be randomly shuffled, the opposite of what we need for secure merge. A new idea is necessary to embed structure into an Oblivious RAM construction so that secure merge can exploit that structure without leaking information through the pattern of memory accesses to that structure.

3.3 Secure Database Management

The power of cloud computing has grown alongside concerns about privacy and security on those platforms. There has therefore been a great deal of research into secure database management applications, including secure searchable encryption [19, 26], where a database is encrypted and can be securely searched, oblivious analytics [48] where specific metrics can be computed on an encrypted database, and oblivious query processing [2, 12], where an encrypted database can obliviously process a streaming set of queries analogous to SQL queries.

The secure searchable encryption tools are built on top of ORAM, and so any improvements to ORAM due to secure merge will be inherited by them. For oblivious analytics and oblivious query processing, ORAM is used as one cryptographic tool, but there are also a number of calls to oblivious sorts to, for example, join lists from two separate parties. In some cases, these lists are guaranteed to be already sorted, and secure merge can be inserted directly into the protocol. In other cases, these lists are already secret-shared and shuffled, and further alterations to the protocol would be necessary to preserve the ordering on the lists to be sorted and thus access the efficiency gains of secure merge while preserving obliviousness.

Additionally, we note that in the special case when an encrypted database only rarely needs to be accessed obliviously, but needs to be frequently updated, the database can efficiently add new data using secure merge much more efficiently than using a secure sorting protocol.

4 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: Imagine we can partition to form k blocks, each with n/k elements. Then for e.g. $k = n/\log\log n$, we could apply the linear protocol of [13] 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, but still only $O(\log\log n)$ rounds. This matches our target complexity in all three metrics.

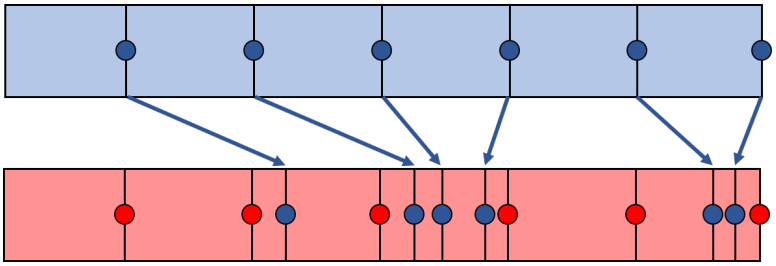

Figure 1: Merging the k medians from one list into the other list.

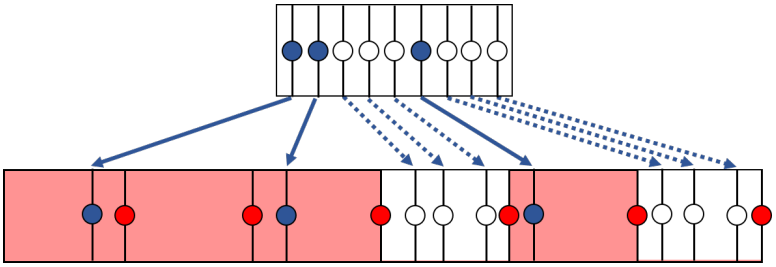

Figure 2: Merging a smaller list into a larger list, and then classifying the elements in the first list and blocks from the second list that are "poorly-aligned:" namely, when multiple (in this case three or more) elements from the first list map to the same block of the second list (pictured in white and with dashed arrows).

4.1 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 the first list, we must identify the appropriate block(s) from the second 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. Namely:

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 about k-medians (given in Section 10) 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).

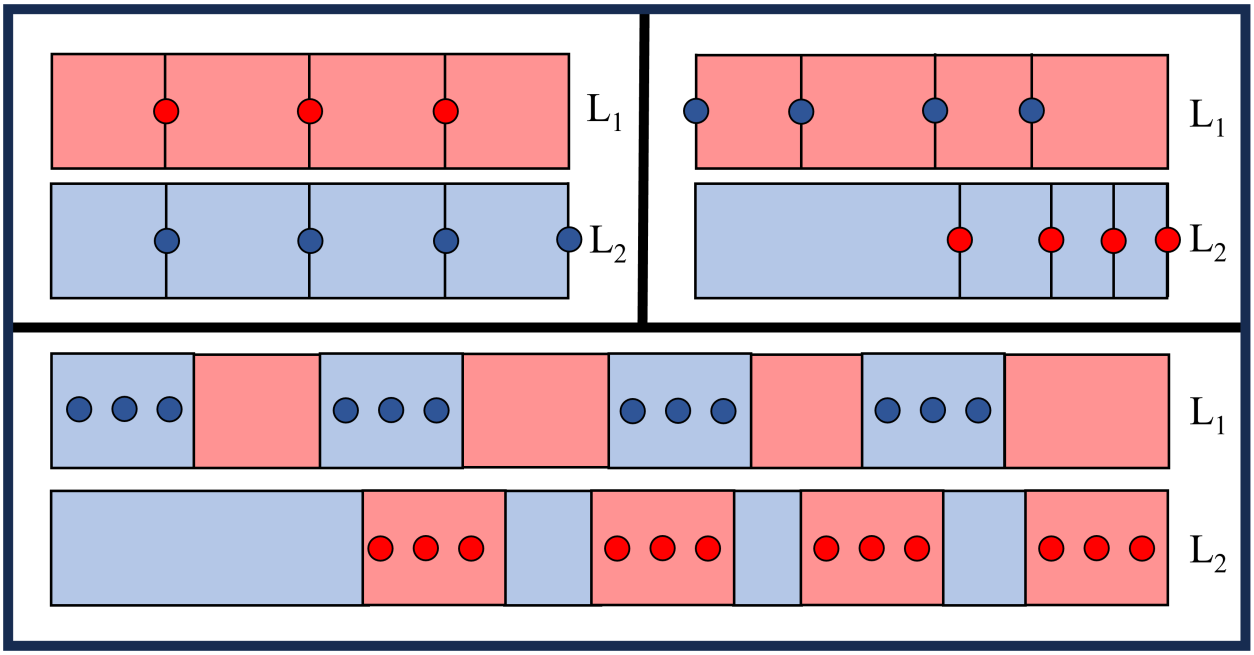

Figure 3: The *Expanding Medians* strategy for converting "well-aligned" to *perfect* alignment: The k-medians of L_1 and L_2 are identified (top-left) and merged into the opposite list (top-right). Then each median is duplicated in place $\frac{n}{k}$ times (bottom), and the resulting lists are broken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

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 (in a secure manner) the number of blocks from the second list that are required to 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 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 §5.4.

4.2 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. [21], or an extension of the shuffle-reveal paradigm of [20]. Our tag-shuffle-reveal paradigm is used implicitly in [14], but we expand its use here, and so describe it in more detail.

In the tag-shuffle-reveal paradigm, each element of a list is (obliviously) tagged with some label. This label can be (a secret sharing of) its current index, or it can be the result of some multiparty computation, for example a bit representing the output of a comparison against another value. Then, after shuffling the list, the tag or some part of the tag is 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.

4.3 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, or extracted and shuffled, or extracted into bins based on a tag they are marked with. 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. Full protocols and proofs can be found in Section 8. The protocol $\mathbf{\Pi_{Extract}}$ for extracting and shuffling marked elements is described in §8.1; the protocol $\mathbf{\Pi_{Ext\text{-}Ord}}$ for extracting marked elements in order in §8.3; and the protocol $\mathbf{\Pi_{Ext\text{-}Bin}}$ for extracting marked elements into bins in §8.5. We use extraction protocols in a black box way in §5-6.

4.4 Primitive Functionalities

We recall the secure merge protocol from [13] discussed above, which requires $O(n)$ communication and $O(n)$ rounds, which we call $\mathbf{\Pi_{SM-FNO}}$ 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 $\mathbf{\Pi_{SM\text{-}ALL}}$ and give it formally in §9.1 for completeness.

The protocols we present below are realized with black box calls to the following four "primitive" functionalities: $\mathbf{\Pi_{Reveal}}$, $\mathbf{\Pi_{Comp}}$, $\mathbf{\Pi_{Sel}}$, $\mathbf{\Pi_{Shuffle}}$ 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 $\mathbf{\Pi_{SM\text{-}FNO}}$), as described below.

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

In $\mathbf{\Pi_{Comp}}$, the 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 [36], which requires $O(W)$ bits of communication and $O(1)$ rounds of communication and is concretely efficient.

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

In $\mathbf{\Pi_{Shuffle}}$, the 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 [14] for the full protocol and a more thorough treatment.

4.5 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. [6]. 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 by way 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.

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

- **Setup**. Each party sends their inputs to $\mathcal{F}$, who stores them plus an id *sid*.
- **Execution**. The parties send the command $(\mathsf{Execute}, sid)$ to $\mathcal{F}$, who computes the desired output and stores it.
- **Reveal**. The parties send 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 $(\mathsf{Execute}_i)$ corresponding to each of the protocols

defined in this paper. This provides an alternate way to address the matter of composability, and guarantees that inputs to one protocol match outputs from another protocol.

4.6 Notation

For any two sorted lists L_1 and L_2, let $\bigsqcup$ denote the "merge" of two lists (i.e. $\bigsqcup$ is functionally equivalent to (multi-)set union followed by sort): $L_1 \bigsqcup L_2 = Sort(L_1 \cup L_2)$. 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 $L_j = \{u_1, \ldots, u_n\}$, then:

$$\mathcal{M}_{j,k} := \{u_{\frac{\ell \cdot n}{k}}\}_{\ell=1}^{k} \tag{2}$$

Basic properties that follow from the definition of medians are in Section 10.

Throughout the paper we distinguish between secure symmetric merge $\mathbf{\Pi_{SSM}}$ and secure asymmetric merge $\mathbf{\Pi_{SAM}}$. In a symmetric merge, the lists are of roughly the same size, or differ by at most a constant factor (since then one list can be padded with dummies to match the length of the other). In an asymmetric merge, the ratio of list sizes is larger than constant. Finally, in analyzing performance of a protocol, RCost will denote the round-complexity, and CCost the computation and overall communication complexity.

5 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 of our symmetric and asymmetric merge protocols (Figures 4 and 5) 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 – $\mathbf{\Pi_{SSM\text{-}\log\log n}}$ and $\mathbf{\Pi_{SAM-n^{1/3}}}$ – 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 $\mathbf{\Pi_{Extract}}$), is sufficient to establish the stated asymptotics of Theorem 1.

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

Table 2: Protocol and subprotocol relations. The Cost column uses:
- For $\mathbf{\Pi_{SSM}}(n)$: Set $k = \frac{n}{\log\log n}$, $\mathbf{\Pi_{SAM}}(k, n)$ = Fig. 5, $\mathbf{\Pi_{SSM}}(\frac{n}{k}) = \mathbf{\Pi_{SM\text{-}FNO}}$
- For $\mathbf{\Pi_{SAM}}(k,n)$: $\mathbf{\Pi_{SSM}}(k) = \mathbf{\Pi_{SSM\text{-}\log\log}}(k)$ (Fig. 7), $\mathbf{\Pi_{SSM}}(\frac{n}{k}) = \mathbf{\Pi_{SM\text{-}FNO}}$

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).
- **Merge Blocks**. This phase can be further specified as the merging of "well-aligned" blocks and the merging of "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 added there are removed, producing the final merged list.

5.1 Secure Symmetric Merge

Our *Secure Symmetric Merge* protocol $\mathbf{\Pi_{SSM}}(n)$ 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 10.1 (Section 10) 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 $\mathbf{\Pi_{SSM'}}(n/k) = \mathbf{\Pi_{SSM\text{-}FNO}}$ and $\mathbf{\Pi_{SAM}}(k,n) = \mathbf{\Pi_{SAM}}(k, n, \mathbf{\Pi_{SSM\text{-}FNO}}, \mathbf{\Pi_{SSM\text{-}\log\log n}})$, which refers to the secure asymmetric merge protocol of Figure 5, using for its subprotocols $\mathbf{\Pi_{SSM'}}(n/k) = \mathbf{\Pi_{SSM\text{-}\log\log n}}$ and $\mathbf{\Pi_{SSM''}}(k) = \mathbf{\Pi_{SSM\text{-}FNO}}$.

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 $\mathbf{\Pi_{SSM'}}(n/k)$ and $\mathbf{\Pi_{SAM}}(k,n)$.

Output. The two lists have been merged (i.e. combined so that the final list is sorted) into an output list $L_1 \bigsqcup L_2$, which has size $2n$ and is (additively) secret-shared amongst the two parties.

Protocol (sketch).

1. **Partition**. Invoke secure asymmetric merge protocol $\mathbf{\Pi_{SAM}}(k,n)$ (Figure 5) to merge the k medians of L_2 with list L_1 (and vice-versa).
2. **Align Blocks.** Invoke the Expanding Medians approach (Figure 3) by running the duplicate values $\mathbf{\Pi_{Dup}}$ protocol (Fig. 16) 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 $\mathbf{\Pi_{SSM'}}(n/k)$, 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 $\mathbf{\Pi_{Ext\text{-}Ord}}$ protocol (Fig. 13) to remove dummy elements.

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

5.2 Analysis of Abstract Secure Symmetric Merge Protocol of §5.1

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

Correctness.
Assuming correctness of $\mathbf{\Pi_{SSM'}}(n/k)$, $\mathbf{\Pi_{SAM}}(k,n)$, $\mathbf{\Pi_{Dup}}$, and $\mathbf{\Pi_{Ext\text{-}Ord}}$ 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 10.1, which demonstrates that lists L_1'' and L_2'' have the same list of $2k$ medians (both equal $\mathcal{M}_{1,k} \bigsqcup \mathcal{M}_{2,k}$).

Cost.

- Step (1) invokes the $\mathbf{\Pi_{SAM}}(k, n)$ protocol (Figure 5) twice.
- Step (2) invokes the $\mathbf{\Pi_{Dup}}(k, n/k)$ protocol (Figure 16) twice.
- Step (3) invokes the $\mathbf{\Pi_{SSM'}}(n/k)$ protocol $2k$ times.
- Step (4) invokes the secure ordered extract $\mathbf{\Pi_{Ext\text{-}Ord}}(4n, 2n)$ protocol.

Using the $\mathbf{\Pi_{Dup}}(k, n/k)$ protocol of Figure 16 and the $\mathbf{\Pi_{Ext\text{-}Ord}}(4n, 2n)$ of Figure 13, and assuming constant-round and linear secure protocols for $\mathbf{\Pi_{Comp}}$, $\mathbf{\Pi_{Reveal}}$, and $\mathbf{\Pi_{Shuffle}}$, adding up these costs yields:

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

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

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

5.3 Secure Asymmetric Merge

Our *Secure Asymmetric Merge* protocol $\mathbf{\Pi_{SAM}}(k, n)$ 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 $\mathbf{\Pi_{Ext\text{-}Bin}}$ protocol (§4.3) to securely perform the classifications of 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 $\mathbf{\Pi_{SSM'}}(n/k) = \mathbf{\Pi_{SSM\text{-}FNO}}$ and $\mathbf{\Pi_{SSM''}}(k) = \mathbf{\Pi_{SSM\text{-}\log\log}}$.

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

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 $\mathbf{\Pi_{SSM'}}(n/k)$ and $\mathbf{\Pi_{SSM''}}(k)$.

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

Protocol.

1. **Partition** (Fig. 1). Run $\mathbf{\Pi_{SSM''}}(L_1, \mathcal{M}_{2,k})$ to merge L_1 and the k medians of L_2.

2a. **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 $\mathbf{\Pi_{Comp}}$) 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.

2b. **Align Blocks: Extract Poorly-Aligned** (Fig. 2). Using the merge positions from the previous step, run the $\mathbf{\Pi_{Ext\text{-}Ord}}$ 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 5.2), so L_1^{PA} and L_2^{PA} will each have size k, with $\mathbf{\Pi_{Ext\text{-}Ord}}$ extracting dummy-elements to pad each list to exactly this size.

2c. **Align Blocks: Extract Well-Aligned** (Fig. 2). For the well-aligned elements of L_1, run the $\mathbf{\Pi_{Ext\text{-}Bin}}$ 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 $\mathbf{\Pi_{Ext\text{-}Bin}}$ 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 $\mathbf{\Pi_{Ext\text{-}Bin}}$ 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 $\mathbf{\Pi_{Ext\text{-}Ord}}$ protocol on each block of L_2.

3a. **Merge Blocks: Poorly-Aligned**. Run $\mathbf{\Pi_{SSM''}}(k)$ on L_1^{PA} and L_2^{PA}.

3b. **Merge Blocks: Well-Aligned**. Run $\mathbf{\Pi_{SSM'}}(n/k)$, 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:

$$\left(\#Left_1^{WA}\right) + \left(\#Left_1^{PA}\right) + \left(\#Left_2^{WA}\right) + \left(\#Left_2^{PA}\right) + Block\_Index$$

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 5.4).

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

5.4 Analysis of Abstract Secure Asymmetric Merge Protocol of §5.3

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: $\mathbf{\Pi_{SSM''}}$, $\mathbf{\Pi_{Comp}}$, $\mathbf{\Pi_{Ext\text{-}Ord}}$, $\mathbf{\Pi_{Ext\text{-}Bin}}$, $\mathbf{\Pi_{SSM''}}$, and $\mathbf{\Pi_{SSM'}}$, respectively. Meanwhile, the correctness property ensures that the indices revealed in Step 4 are unique and are a (random) permutation of the values in $[1, \ldots, (k+n)]$. 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:

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

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:

$\underline{L_1.WA}$: $(U + V + Y)$ 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).

$\underline{L_2.WA}$: $(U + V)$ 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.

$\underline{L_1.PA}$: $(U + V + Y)$ 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.

$\underline{L_2.PA}$: $(U + V)$ 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 $\mathbf{\Pi_{Ext\text{-}Ord}}$ and $\mathbf{\Pi_{Ext\text{-}Bin}}$ protocols are satisfied. Namely, how to (efficiently) construct the input parameters C and T' of the $\mathbf{\Pi_{Ext\text{-}Bin}}$ 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 5.1. *(Shares of) the parameters C and T' to the $\mathbf{\Pi_{Ext\text{-}Bin}}$ 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 $C = \{c_j\}$ and $T' = \{t'_j\}$ can be expressed iteratively as:

$$\begin{aligned} t'_1 &= \iota_1 - 1 \quad \text{and} \quad t'_{j+1} = \iota_{j+1} - j - \textstyle\sum_{i \le j} t'_i \\ c_1 &= \delta_1 \qquad \text{and} \quad c_{j+1} = \delta_{j+1} \cdot (1 + \textstyle\sum_{i \le j} t'_i) \end{aligned}$$

While the expressions above 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 our $\mathbf{\Pi_{Ext\text{-}Bin}}$ protocol of Fig. 14 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. □

Observation 5.2. *For both L_1 and L_2, the number of poorly-aligned elements in Step 2b of the $\mathbf{\Pi_{SAM}}(k, n)$ 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. □

Observation 5.3. *For both L_1 and L_2, the number of well-aligned elements in each block of Step 2c of the $\mathbf{\Pi_{SAM}}(k, n)$ 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 $\mathbf{\Pi_{Ext\text{-}Bin}}$ in Step 2c for this block. □

Cost.

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

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

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

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

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

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

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

In this section, we present our $\mathbf{\Pi_{SSM\text{-}\log\log n}}$ and $\mathbf{\Pi_{SAM-n^{\alpha}}}$ base protocols (for $\alpha = 1/3$; the general case for $\alpha < 1$ can be found in §7.3).

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

As an ingredient for $\mathbf{\Pi_{SAM}}(k,n)$, we need a secure symmetric merge with $O(n\log\log n)$ communication and $O(\log\log n)$ rounds, which we call $\mathbf{\Pi_{SSM\text{-}\log\log}}$. We give this protocol in Figure 7. For the Partition phase, it uses a secure asymmetric merge protocol (namely, the $\mathbf{\Pi_{SAM\text{-}n^{1/3}}}$ 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.

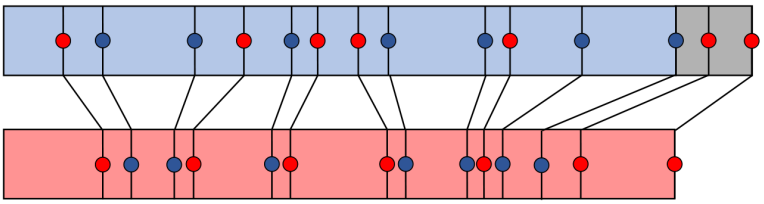

Figure 6: Each list is partitioned into $2k$ blocks by the k medians from both lists. The two grey blocks appended to the end of the top list are made up entirely of dummy elements, and show how to handle medians from one list that merge to a location outside the other list. Each block is padded with 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 $\mathbf{\Pi_{SSM}}(n/k)$, with the final result obtained by concatenating the results of these $2k$ merges and removing dummy elements.

6.2 Analysis of the $\mathbf{\Pi_{SSM\text{-}\log\log}}(n)$ 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 $\{1, 2, \ldots, 2n\}$, 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, we note that, as in §5.4, the $\mathbf{\Pi_{SAM\text{-}n^{1/3}}}$ 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 $\mathbf{\Pi_{SM\text{-}ALL}}$, 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 $\mathbf{\Pi_{Ext\text{-}Bin}}$. It remains to show that the assumed bound on the size of each bin holds. The union of

Secure Symmetric Merge $\mathbf{\Pi_{SSM\text{-}\log\log n}}$

Input. Parties $\mathcal{P}_1$, $\mathcal{P}_2$ (additively) secret-share *sorted* lists L_1, L_2 of size n.

Output. The two lists have been merged (i.e. combined so that the final list is sorted) into an output list $L_1 \bigsqcup L_2$, which has size $2n$ and is (additively) secret-shared amongst the two parties.

RCost. $O(\log\log n)$ rounds.

CCost. $O(\log\log n) \cdot [n \cdot \mathsf{CCost}(\mathbf{\Pi_{Reveal}} + \mathbf{\Pi_{Comp}} + \mathbf{\Pi_{Sel}}) + \mathsf{CCost}(\mathbf{\Pi_{Shuffle}}(n))]$.

Protocol.

1. Compute $d = \frac{\log\log n}{\log(3/2)} - O(1)$ such that $4 \leq n^{(2/3)^d} < 8$.
2. Define $P_0 := \{(L_1, L_2, 0)\}$, to be the top-level array of sub-problems of size n. Each subproblem consists of two lists L_1, L_2 and a final offset ω. Define $n_k := n^{(2/3)^k}$. Then, for $k = 1, \ldots, d$, do Steps 3 through 10.
3. For each tuple $(A_1, A_2, \omega) \in P_{k-1}$, do Steps 4 through 8.
4. **Partition** (Fig. 6). Run $\mathbf{\Pi_{SAM\text{-}n^{1/3}}}$ twice to merge the $n_{k-1}^{1/3}$ medians $\mathcal{M}_{1,n_{k-1}^{1/3}}$ of A_1 with A_2 and the $n_{k-1}^{1/3}$ medians $\mathcal{M}_{2,n_{k-1}^{1/3}}$ of A_2 with A_1.
5. **Align Blocks: Label** Label each element of A_1 and A_2 with a destination block by counting the number of medians from either list that are less than that element, which can be computed from the (secret-shared) destination indices under the merges in the previous step.
6. **Align Blocks: Auxiliary Information** Tag each of the medians from both lists with the number of elements less than it from A_1. Then tag all medians from both lists with the number of elements less than it from A_2. Run $\mathbf{\Pi_{SM\text{-}ALL}}$ to merge the tagged medians $\mathcal{M}_{1,n_{k-1}^{1/3}}$ with the tagged medians $\mathcal{M}_{2,n_{k-1}^{1/3}}$. Then each party computes locally the auxiliary information: The number of elements from each list in each bin, and the index of the first element from each list in each bin.
7. **Align Blocks: Extract** Run the $\mathbf{\Pi_{Ext\text{-}Bin}}$ protocol to extract all elements from A_1 into $2n_{k-1}^{2/3} = 2n_k$ bins of size $n_{k-1}^{1/3}$. Run the same $\mathbf{\Pi_{Ext\text{-}Bin}}$ protocol for A_2. This requires the auxiliary information computed in the previous step.
8. Match corresponding bins from these two $\mathbf{\Pi_{Ext\text{-}Bin}}$ outputs to give new subproblems $(B_{j,1}, B_{j,2}, \omega_j)$; compute the new offsets ω_j by adding auxiliary information to ω. Append these subproblems to P_k.
9. Compute: $S_k := 1 + 2\sum_{i=1}^{k} n^{(1-\frac{2^i}{3^i})}$.
10. Call $\mathbf{\Pi_{Extract}}$ on P_k to extract all the subproblems with at least one non-dummy element. There are at most S_k such subproblems by Lemma 6.1.
11. **Block Merge Phase**. Perform $\mathbf{\Pi_{SM\text{-}ALL}}$ on every pair $(A_1, A_2, [\omega]) \in P_d$, and add $[\omega]$ to all resulting destination indices to give the final destination index of all real elements. Call $\mathbf{\Pi_{Extract}}$ on the union of all lists in P_d to extract the $2n$ real elements, open the secret-shared destination tags, and place each element in its correct destination.

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

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 $2n_{k-1}^{1/3}$ 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 6.1, 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 $\mathbf{\Pi_{SM\text{-}ALL}}$, and since we have verified that P_d contains a partition of the $2n$ real elements, the bound on real elements for the final $\mathbf{\Pi_{Extract}}$ call is also correct. We now return to the lemma establishing the bound on the size of subproblems:

Lemma 6.1. *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 the recurrence relation:

$$N_{k+1} \leq N_k + 2n^{1-(2/3)^k},$$

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:

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

□

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 6.1, 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 $|L_2| = O(n)$, we present in Figure 9 an asymmetric merge protocol with communication complexity $O(n)$ and round complexity $O(1)$. By calling $\mathbf{\Pi_{SM\text{-}ALL}}$ (see Figure 15) 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 $\mathbf{\Pi_{SM\text{-}ALL}}$ 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. We give a sketch of the $\mathbf{\Pi_{SAM\text{-}n^{1/3}}}$ protocol in Figure 8 and the full protocol in Figure 9.

Cubic Secure Asymmetric Merge $\mathbf{\Pi_{SAM\text{-}n^{1/3}}}(n^{1/3}, n)$

Input. Two parties (additively) secret-share *sorted* lists (L_1, L_2) of sizes $(n^{1/3}, n)$.

Output. The two lists have been merged into an output list $L_1 \bigsqcup L_2$, which has size $n + n^{1/3}$ and is (additively) secret-shared amongst the two parties.

Protocol (Sketch).

1. **Partition**. Invoke the $\mathbf{\Pi_{SM\text{-}ALL}}$ on L_1 and the $n^{2/3}$-medians of L_2, thus identifying which block of L_2 that each element of L_1 lies within.
2. **Align Blocks**. Run $\mathbf{\Pi_{Extract}}$ to securely perform the classification of all blocks of L_2 as "poorly-aligned" or "well-aligned", where here a block of L_2 is well-aligned if at least one element of L_1 lies within that block.
3. **Merge Blocks**. Invoke the secure merge via compare all protocol on L_1 and the entire list of well-aligned blocks from L_2 extracted in Step 2.
4. **Combine Blocks**. Remove dummy elements (introduced in Steps 2 and 3).

Figure 8: Cubic Secure Asymmetric Merge $(n^{1/3}, n)$ Protocol

7 Full Description and Analyses of Protocols in §6

In this section, we give the full protocol descriptions for the merge protocols described in Section 6: $O(n \log \log n)$ Secure Symmetric Merge, and Cubic Secure Symmetric Merge. We also give the (n^α, n) asymmetric merge in §7.3, and associated subprotocols in the following subsections. We specify the inputs and outputs of each party under an ideal world execution, give proofs of correctness and security, and derive the round cost RCost and the communication cost CCost of each protocol in terms of the cost of underlying sub-protocols or underlying primitives.

7.1 Description of the $\mathbf{\Pi_{SAM\text{-}n^{1/3}}}(n^{1/3}, n)$ Protocol of §6.3

Cubic Secure Asymmetric Merge $\mathbf{\Pi_{SAM\text{-}n^{1/3}}}(n^{1/3}, n)$

Input. Two parties $\mathcal{P}_1$, $\mathcal{P}_2$ (additively) secret-share *sorted* list L_1 of size $n^{1/3}$ and L_2 of size n.

Output. The two lists have been merged (i.e. combined so that the final list is sorted) into an output list $L_1 \bigsqcup L_2$, which has size $n + n^{1/3}$ and is (additively) secret-shared amongst the two parties.

RCost. $O(1)$.

CCost. $(3n + 2n^{2/3} + 2n^{1/3})\mathsf{CCost}(\mathbf{\Pi_{Comp}}) + (4n + 2n^{2/3})\mathsf{CCost}(\mathbf{\Pi_{Reveal}}) + 3\mathsf{CCost}(\mathbf{\Pi_{Shuffle}})(n + n^{2/3})$.

Protocol.

1. **Initial Phase**. Invoke $\mathbf{\Pi_{SM\text{-}ALL}}(L_1, \mathcal{M}_{2,n^{2/3}})$ in the "in-place" setting, to obtain secret shares of the destinations of L_1 and $\mathcal{M}_{2,n^{2/3}}$ after merging.
2. **Label Phase**. Construct a shared list $T := ([\iota_{i,2}] - i) \bowtie ([\iota_{i,2} - \iota_{i-1,2} > 1])$ of length $n^{2/3}$, and a second list T_n formed by copying each element of T a total of $n^{1/3}$ times, so that T_n has total length n. Define τ_i to be i plus the first coordinate of the i^{th} element of T_n, that is, write $T_n := ([\tau_i] - i) \bowtie (\upsilon_i)$. Note that $[\tau_i]$ denotes the correct index of $a_{i,2}$ after merging if no elements of L_1 lie in the block to which the i^{th} element of L_2 belongs, and υ_i is an indicator for whether $a_{i,2}$ lies in a block that contains an element of L_1 after merging.
3. Construct $\overline{L}_2 := L_2 \bowtie ([\tau_i]) \bowtie ([i]) \bowtie ([\upsilon_i])$, a sequence of ordered 4-tuples.
4. Construct $\overline{L}_2'$ similarly, but reverse the condition tested by T, that is, replace each element $([\tau_i], [\upsilon_i]) \in T_n$ with $([\tau_i], 1 - [\upsilon_i])$.
5. **Align Blocks**. Define $L_2^A := \mathbf{\Pi_{Extract}}(\overline{L}_2, n^{2/3})$ and $L_2^B := \mathbf{\Pi_{Extract}}(\overline{L}_2', n)$, so that the union of the nun-dummy elements of L_2^A and L_2^B is $\overline{L}_2$.
6. **Merge Blocks**. Compute: $(\widehat{\overline{L}}_1, \widehat{L}_2^A) := \mathbf{\Pi_{SM\text{-}ALL}}(L_1, L_2^A)$. Write elements of $\widehat{\overline{L}}_1$ as $([a_{i,1}], [\kappa_{i,1}])$ and elements of $\widehat{L}_2^A$ as $([a_i^A], [\tau_{i,2}], [\widehat{i}], [\kappa_{i,2}])$.
7. Define $\overline{L}_2^A := ([a_i^A], [\widehat{i}] + [\kappa_{i,2}] - i, [\widehat{i}])$.
8. Construct the list $L_2^C := \mathbf{\Pi_{Shuffle}}(\overline{L}_2^A \cup L_2^B) := (\ell_i) \bowtie (\phi_{i,2}) \bowtie (\widehat{i})$.
9. Reveal the values $\widehat{i}$. Discard the dummy elements with $\widehat{i} = 0$ and arrange the remaining values $(\ell_i, \phi_{i,2})$ in order based on these values. Call the resulting list L_2^D.
10. Return $(L_1, i + n^{1/3} \cdot ([\iota_{i,1}] - i) + [\kappa_{i,1} \mod n^{1/3}])$ and L_2^D.

Figure 9: Cubic Secure Asymmetric Merge $(n^{1/3}, n)$ Protocol

7.2 Analysis of Secure Asymmetric Merge Protocol of §6.3

Security.
Security follows from the security of the underlying protocols, after verifying that the conditions for the calls to $\mathbf{\Pi_{Extract}}$ are satisfied and that nothing is revealed in the $\mathbf{\Pi_{Reveal}}$ call in Step 9. We verify both of these conditions in our proof of correctness, below.

Correctness.
The indicator υ_i introduced in Step 2 tests whether any elements of L_1 map into the i^{th} block of L_2. Because there are $n^{1/3}$ total elements of L_1, there are at most $n^{1/3}$ such blocks, and so at most $n^{2/3}$ elements in the blocks, since the blocks have size $n^{1/3}$. Thus the condition for calling $\mathbf{\Pi_{Extract}}$ to generate L_2^A in Step 5 is satisfied. Since there are n elements total in L_2, the condition for L_2^B is also satisfied.

For elements of L_1, $\iota_{i,1} - i$ counts the number of blocks of L_2 situated entirely to the left, and $\kappa_{i,1}$ counts the number of elements of L_2 situated to the left, plus some additional blocks that are double counted. Thus, after modding out by $n^{1/3}$, The indices for L_1 computed in Step 10 are correct.

For elements $\ell \in L_2^A$, the value $\kappa_{i,2} - i$ counts the total number of elements from all of L_1 that end left of ℓ, so adding back in $\hat{i}$ gives the desired destination index. For element $\ell \in L_2^B$, the value τ_i computed in Step 2 is already the correct destination index, since no elements from L_1 maybe into the block of ℓ. Thus in Step 9, all elements of L_2 are given the correct destination index.

Cost.
Step 1 requires $n^{1/3} \cdot n^{2/3} = n$ calls to $\mathbf{\Pi_{Comp}}$. Step 2 requires $n^{2/3}$ calls to $\mathbf{\Pi_{Comp}}$. Step 3 and 4 can both be computed locally. Step 5 requires $n + n^{2/3}$ comparisons, $3n + n^{2/3}$ calls to $\mathbf{\Pi_{Reveal}}$, and two calls to $\mathbf{\Pi_{Shuffle}}(n)$. Step 6 again requires n comparisons. Step 7 is free. Step 8 requires another call to $\mathbf{\Pi_{Shuffle}}$. Step 9 requires $n + n^{2/3}$ calls to $\mathbf{\Pi_{Reveal}}$. Finally, Step 10 requires $n^{1/3}$ modular reductions, which in turn requires $2n^{1/3}$ calls to $\mathbf{\Pi_{Comp}}$.

This gives a total cost of $O(1)$ rounds and communication:

$$(3n + 2n^{2/3} + 2n^{1/3}) \cdot \mathsf{CCost}(\mathbf{\Pi_{Comp}}) + (4n + 2n^{2/3}) \cdot \mathsf{CCost}(\mathbf{\Pi_{Reveal}}) + 3 \cdot \mathsf{CCost}(\mathbf{\Pi_{Shuffle}})(n + n^{2/3}). \tag{3}$$

7.3 Secure n^α Asymmetric Merge Protocols

In this section, we describe how to generalize the Secure Asymmetric Merge protocol from §6.3 to work on a pair of lists L_1 of size n^α (for *any* constant $\alpha < 1$), and L_2 of size n, that runs in time $O(n)$ with $O(1)$, for any fixed $\alpha < 1$, where the implied constants depend on α. As we show below, the implied constants are small for $\alpha = 1/3$, and bounded above by $2^{2/(1-\alpha)^3}$ in general for communication and $2/(1-\alpha)^3$ for round complexity.

Our protocol works by bootstrapping up from our cubic merge protocol $\mathbf{\Pi_{SM-n^{1/3}}}$ via a compiler from asymmetric merge on (n^β, n) (starting with $\beta = 1/3$) to asymmetric merge on (n^γ, n), with:

$$\gamma = \frac{1}{\beta^2 - 3\beta + 3}.$$

We write $\mathbf{\Pi_{SM-n^\beta}}$ for a particular secure merge protocol for the constant $\beta < 1$, and $\mathbf{\Pi_{SAM-\gamma-\beta}}$ for the protocol that bootstraps from $\mathbf{\Pi_{SM-n^\beta}}$ to $\mathbf{\Pi_{SM-n^\gamma}}$.

Note that the construction of $\mathbf{\Pi_{SM-n^{1/3}}}$, which is the only subprotocol needed for our main $\mathbf{\Pi_{SSM}}$ protocol, was presented above in §6.3, and its analysis is given in §7.2. We give the bootstrapping protocol in Figure 11 and the full protocol and analysis in §7.4.

We begin with the top-level protocol for performing secure asymmetric merge in time $O(n)$ and $O(1)$ rounds for fixed $\alpha < 1$. In terms of α, we give the upper bound of $O(n2^{2/(1-\alpha)^3})$ communication and $O(1/(1-\alpha)^3)$ rounds as $\alpha \to 1$.

The correctness and security of the $\mathbf{\Pi_{SAM-n^\alpha}}$ protocol follows immediately from correctness and security of $\mathbf{\Pi_{SAM-n^{1/3}}}$ and $\mathbf{\Pi_{SAM-\gamma-\beta}}$, as long as the sequence (γ_i) constructed in Step 1 of this protocol actually terminates. We show that in fact it terminates in at most $2/(1-\alpha)^3$ steps, from which the above bounds on communication and round complexity follow from the communication and round complexity of the previous two protocols.

But indeed, for any $\beta < \alpha$, we have

$$\frac{1}{\beta^2 - 3\beta + 3} - \beta = \frac{(1-\beta)^3}{(1-\beta)(2-\beta) + 1} \geq (1-\beta)^3/3 \geq (1-\alpha)^3/3.$$

Since we begin with $\beta = 1/3$, it requires at most

$$\frac{\alpha - 1/3}{(1-\alpha)^3/3} \leq \frac{2}{(1-\alpha)^3}$$

steps of the recurrence before the sequence of (γ_i)'s becomes greater than α.

Secure Asymmetric Merge $(n^\alpha, n)\mathbf{\Pi_{SAM\text{-}n^\alpha}}(n^\alpha, n)$

Input. Two parties $\mathcal{P}_1$, $\mathcal{P}_2$ (additively) secret-share *sorted* list L_1 of size n^α and L_2 of size n (for $\alpha \leq 1/2$).

Output. The two lists have been merged (i.e. combined so that the final list is sorted) into an output list $\overline{L_1 \bigsqcup L_2}$, which has size $n + n^\alpha$ and is (additively) secret-shared amongst the two parties.

RCost. $O(1/(1-\alpha)^3)$.

CCost. $O(n \cdot 2^{2/(1-\alpha)^3})$.

Protocol.

1. Construct a finite sequence (γ_i) as follows:
   - Set $\gamma_1 := 1/3$, and $i = 1$.
   - While $\gamma_i < \alpha$, set $\gamma_{i+1} \leftarrow \frac{1}{\gamma_i^2 - 3\gamma_i + 3}$ and $i \leftarrow i + 1$.
2. Let ℓ be the length of (γ_i). Then for $i \in [1..(\ell-1)]$, construct the protocol $\mathbf{\Pi_{SAM-\gamma_{i+1}}}$ by calling $\mathbf{\Pi_{SAM-\gamma_{i+1}-\gamma_i}}(n^{\gamma_{i+1}}, n, \mathbf{\Pi_{SAM-\gamma_i}})$. This gives a protocol $\mathbf{\Pi_{SAM-\gamma_\ell}}$, for $\gamma_\ell > \alpha$.
3. Pad the list L_1 with $n^{\gamma_\ell} - n^\alpha$ dummy elements, and call the result $\overline{L}_1$.
4. Call $\mathbf{\Pi_{Extract}}(\mathbf{\Pi_{SAM-\gamma_\ell}}(\overline{L}_1, L_2))$ to obtain the desired result, extracting all non-dummy elements.

Figure 10: Secure Asymmetric Merge (n^α, n) Protocol

7.4 Bootstrapping Protocol $\mathbf{\Pi_{SAM-\beta-\gamma}}$ Protocol

We give in Figure 11 a protocol that bootstraps from $\mathbf{\Pi_{SAM-n^\beta}}$ to $\mathbf{\Pi_{SAM-n^\gamma}}$, for γ close to β, in $O(1)$ rounds and roughly twice the communication of the previous level of bootstrapping. In particular, by setting e.g. $\beta = 1/3$, this yields a secure asymetric merge protocol on lists of size (n^γ, n) for any $\gamma \in [1/3, (1-\epsilon)]$, for any fixed $\epsilon > 0$.

Secure Asymmetric Merge Recursion Step $\mathbf{\Pi_{SAM\text{-}\beta\text{-}\gamma}}(n^\gamma, n, \mathbf{\Pi_{SAM-\beta}})$

Input. Two parties $\mathcal{P}_1$, $\mathcal{P}_2$ (additively) secret-share *sorted* list L_1 of size n^γ and L_2 of size n. Additionally, an algorithm $\mathbf{\Pi_{SAM-\beta}}$) is given, with:

$$\gamma = \frac{1}{\beta^2 - 3\beta + 3}.$$

Output. The two lists have been merged (i.e. combined so that the final list is sorted) into an output list $\overline{L_1 \bigsqcup L_2}$, which has size $n + n^\gamma$ and is (additively) secret-shared amongst the two parties.

Protocol.

1. **Partition**. Merge in the $n^{1-\gamma}$ medians of L_2 with L_1 using $\mathbf{\Pi_{SM\text{-}ALL}}$, thus identifying which block of L_2 the elements of L_1 lie within.
2. **Align Blocks**. Use calls to $\mathbf{\Pi_{Ext\text{-}Bin}}$ and $\mathbf{\Pi_{Ext\text{-}Ord}}$ to extract "well-aligned" blocks of L_2 to which at most $n^{\beta\gamma}$ elements from L_1 are assigned, and the remaining "poorly-aligned" blocks and elements.

3a. **Merge Blocks: Well-Aligned**. Iterate through each of the $n^{1-\gamma}$ well-aligned blocks of L_2, merging in the appropriate $\leq n^{\beta\gamma}$ elements in L_1 (as identified in Step 2) using $\mathbf{\Pi_{SAM\text{-}n^\beta}}$.

3b. **Merge Blocks: Poorly-Aligned**. Merge the poorly-aligned elements of L_1 with the remaining poorly-aligned elements of L_2, using $\mathbf{\Pi_{SAM\text{-}n^\beta}}$ again.

4. **Combine Blocks**. Remove dummy elements (introduced in Steps 3a-b).

Figure 11: Secure Asymmetric Merge (n^γ, n) Bootstrapping Protocol

7.5 Analysis of the Bootstrapping Protocol of §7.4

Security.
Security follows from the security of the underlying protocols, after verifying that the conditions for the calls to $\mathbf{\Pi_{Extract}}$ are satisfied and that nothing is revealed in the $\mathbf{\Pi_{Reveal}}$ call in Steps 12, 13, and 15. We verify both of these conditions in our proof of correctness, below.

Correctness.
The lists $(L^A_{1,k})$ contain the first $n^{\beta\gamma}$ elements mapped to each block of size n^γ of L_2, and so we can call $\mathbf{\Pi_{SM-n^\beta}}$ on these pairs of blocks. We extract into L^B_1 all elements after the first $n^{\beta\gamma}$. There are at most n^γ such elements. We extract into L^B_2 all blocks which have more than $n^{\beta\gamma}$ elements mapped to them. There are at most $n^{\gamma(1-\beta)}$ such blocks, and so at most $n^{\gamma(2-\beta)}$ such elements.

After dividing L^B_1 into blocks in Step 4, there are $n^{1-\gamma(2-\beta)}$ subproblems, which each have size $(n^{(\gamma(3-\beta))-1}, n^{\gamma(2-\beta)})$. By the assumption that $\gamma = 1/(\beta^2 - 3\beta + 3)$, we have $\beta \cdot \gamma(2-\beta) = ((\gamma(3-\beta)) - 1)$, so we can apply the $\mathbf{\Pi_{SM-n^\beta}}$ protocol here as well, in Step 4.

Correctness follows from similar index chasing as in the previous protocols.

Cost.
Step 1 requires n comparisons. Step 2 requires $o(n)$ comparisons. Step 3 is free. Step 4 requires n calls to min which gives n calls to $\mathbf{\Pi_{Comp}}$. Step 5 requires $o(n)$ calls to $\mathbf{\Pi_{Comp}}$ and $\mathbf{\Pi_{Sel}}$. Because $n^{\gamma(2-\beta)} = o(n)$, Step 6 requires $n + o(n)$ calls to $\mathbf{\Pi_{Reveal}}$ and $o(n)$ calls to $\mathbf{\Pi_{Comp}}$. Similarly, Step 7 requires $2n \cdot \mathbf{\Pi_{Comp}} + 3n \cdot \mathbf{\Pi_{Reveal}} + n \cdot \mathbf{\Pi_{Sel}}$ plus $o(n)$ communication. Step 8 requires $n^{1-\gamma}$ calls to $\mathbf{\Pi_{SM-n^\beta}}(n^{\beta\gamma}, n^\gamma)$, and similarly Step 9 requires $n^{1-\gamma}$ calls to $\mathbf{\Pi_{SM-n^\beta}}(n^{\beta\gamma}, n^\gamma)$. Finally, Steps 10 through 15 require $3n + o(n)$ calls to $\mathbf{\Pi_{Reveal}}$ and 3 shuffles.

This gives a total cost of $O(1) + \mathsf{RCost}(\mathbf{\Pi_{SM-n^\beta}})$ rounds and

$$(4n+o(n))\cdot\mathsf{CCost}(\mathbf{\Pi_{Comp}})+(7n+o(n))\cdot\mathsf{CCost}(\mathbf{\Pi_{Reveal}})+6\cdot\mathsf{CCost}(\mathbf{\Pi_{Shuffle}})(n)+$$
$$n\cdot\mathsf{CCost}(\mathbf{\Pi_{Sel}})+n^{1-\gamma}\cdot\mathsf{CCost}(\mathbf{\Pi_{SM\text{-}n^\beta}}(n^\gamma))+n^{1-\gamma(2-\beta)}\cdot\mathsf{CCost}(\mathbf{\Pi_{SM\text{-}n^\beta}}(n^{\gamma(2-\beta)}))$$

communication, and the last two terms can be bounded by $2\mathsf{CCost}\mathbf{\Pi_{SM\text{-}n^\beta}}(n)$.

8 Extraction Protocols

8.1 Extracting Unordered Marked Elements

Figure 12 has pseudo-code for our Unordered Extract protocol $\mathbf{\Pi_{Extract}}(A, t)$.

Extraction Protocol $\mathbf{\Pi_{Extract}}(A, t)$

Input. Two parties $\mathcal{P}_1$, $\mathcal{P}_2$ (additively) secret-share a list A of size n of elements of the form (a_i, ι_i), of which $t' \leq t$ elements are marked by $\iota_i = 1$, and the other elements have $\iota_i = 0$.

Output. Each party holds additive shares of a list B of length t containing all elements a_i with $\iota_i = 1$, together with $(t - t')$ *dummy* elements, shuffled randomly.

RCost. $O(1) + \mathsf{RCost}(\mathbf{\Pi_{Shuffle}}(n))$.

CCost. $\mathsf{CCost}(\mathbf{\Pi_{Shuffle}}(n)) + t\mathsf{RCost}(\mathbf{\Pi_{Comp}}) + (n + t)\mathsf{RCost}(\mathbf{\Pi_{Reveal}})$.

Protocol.

1. Append a "is-non-dummy" tag $d_i = 1$ to each element in A, so that each element i in A has form: (a_i, ι_i, d_i)
2. Generate shares of the total number of marked elements $[t'] := [\sum_{i=1}^{n} \iota_i]$. This can be done locally.
3. For $i = n + 1, \ldots, n + t$, append to A the element:
$$(a_i, \iota_i, d_i) := (0, \delta_i, 0), \qquad \text{where } \delta_i \text{ is an indicator on } (i - n) > t' \tag{4}$$
so that exactly t elements will have ι tag equal to '1'. Each party sets their share of a_i and d_i to 0, and computes their share of ι_i as per (4) using parallel invocations of a secure comparison protocol.
4. Invoke the shuffling protocol, denoting the output $\widehat{A} := \mathbf{\Pi_{Shuffle}}(A)$.
5. Reveal all values $\widehat{\iota_i}$ (exactly t of which will have value '1').
6. Initialize B to an empty list.
7. For $i = 1, \ldots (n + t)$, if $\widehat{\iota_i} = 1$, each party copies their share of $(\widehat{a_i}, \widehat{d_i})$ to B ($\widehat{\iota_i}$ is *not* copied over, since this tag has been revealed). Notice this will result in B having size t, with t' elements having "is-non-dummy" tag $\widehat{d} = 1$, and the remaining $(t - t')$ elements having "is-non-dummy" tag $\widehat{d} = 0$.

Figure 12: Extract and Shuffle Marked Elements Protocol

8.2 Analysis of the Unordered Extract protocol $\mathbf{\Pi_{Extract}}(A, t)$

Security.
Security of Steps 1, 2, and 3 follow from security of the underlying comparison and shuffling protocols. By correctness (shown below), the values $\widehat{\iota_i}$ will contain t 1's and n 0's, and after calling the $\mathbf{\Pi_{Shuffle}}$ protocol, these values will be randomly arranged in $\widehat{A}$, and so both party's view in Step 4 can be generated uniformly at random by a simulator.

Correctness.
There are exactly $t - t'$ values in $\{n + 1, \ldots, n + t\}$ with $i - n > t'$, and so Step 2 adds $t - t'$ values with $\iota_i = 1$ to the existing t' values with $\iota_i = 1$ in A. These are the values placed into B in Step 6, and they are randomly distributed by the correctness of $\mathbf{\Pi_{Shuffle}}$.

Cost.
Step 2 requires t comparison operations $\mathbf{\Pi_{Comp}}$ and Step 4 requires $n+t$ revealing operations. Combining with Step 3 gives $O(1)+\mathsf{RCost}(\mathbf{\Pi_{Shuffle}}(n))$ rounds and $\mathsf{CCost}\mathbf{\Pi_{Shuffle}}(n)+t\mathsf{RCost}(\mathbf{\Pi_{Comp}})+(n + t)\mathsf{RCost}(\mathbf{\Pi_{Reveal}})$.

8.3 Extracting Ordered Marked Elements

Figure 13 has pseudo-code for our Ordered Extract protocol $\mathbf{\Pi_{Ext\text{-}Ord}}(A, t)$.

Extraction Protocol $\mathbf{\Pi_{Ext\text{-}Ord}}(A, t)$

Input. Two parties $\mathcal{P}_1$, $\mathcal{P}_2$ (additively) secret-share a *sorted* list A of size n of elements of the form (a_i, ι_i), of which $t' \leq t$ elements are marked by $\iota_i = 1$, and the other elements have $\iota_i = 0$.

Output. Each party holds additive shares of a list B of length t containing all elements a_i with $\iota_i = 1$, in their original order, followed by $(t - t')$ dummy elements.

RCost. $O(1) + \mathsf{RCost}(\mathbf{\Pi_{Shuffle}}(n))$.

CCost. $\mathsf{CCost}(\mathbf{\Pi_{Shuffle}}(n)) + t \cdot \mathsf{RCost}(\mathbf{\Pi_{Comp}}) + (n + 2t) \cdot \mathsf{RCost}(\mathbf{\Pi_{Reveal}})$.

Protocol.

1. Compute the extracted position e_i of each element of A that has $\iota_i = 1$. Namely, the lowest-indexed (leftmost) element of A with $\iota_i = 1$ will have $e_i = 1$, the second lowest-indexed element will have $e_i = 2$, and so on. For all non-extracted elements of A (those with $\iota_i = 0$), define $e_i := 0$.
2. Append two additional tags (e_i, d_i) to each element in A, where e_i denotes each element's extracted index (or '0', for non-extracted elements), as per Step 1, and $d_i = 1$ represents (shares of) an "is-non-dummy" tag (so all original elements in A have $d_i = 1$).
3. Generate shares of the total number of marked elements $[t'] := [\sum_{i=1}^{n} \iota_i]$.
4. For $i = n + 1, \ldots, n + t$, append to A the element:

$$(a_i, \iota_i, e_i, d_i) := (0, \delta_i, (i - n + t'), 0),$$

   where δ_i is an indicator on $(i - n) > t'$ (so that exactly $(t - t')$ new elements will have ι tag equal to '1'), and e_i denotes the extracted index of this element (in case it is to be extracted). Each party sets their share of a_i and d_i to 0 and their share of $e_i = (i - n + t')$ using their shares of t' from Step 3, and computes their share of $\iota_i = \delta_i$ by invoking a secure comparison protocol.
5. Invoke the shuffling protocol, denoting the output $\widehat{A} := \mathbf{\Pi_{Shuffle}}(A)$.
6. Reveal all values $\widehat{\iota_i}$ (exactly t of which will have value '1').
7. Reveal all values $\widehat{e_i}$ for the elements that had $\widehat{\iota_i} = 1$.
8. Initialize B to an empty list.
9. For $i = 1, \ldots t$, set the i^{th} element of B to the (unique) element $(\widehat{a_j}, \widehat{d_j})$ with $\widehat{e_j} = i$ ($\widehat{\iota_j}$ and $\widehat{e_j}$ are *not* copied over, since these tags have been revealed). This will result in B having size t, with t' elements having "is-non-dummy" tag $\widehat{d} = 1$, and the remaining $(t - t')$ elements with "is-non-dummy" tag $\widehat{d} = 0$.

Figure 13: Stably Extract Marked Elements Protocol

8.4 Analysis of Ordered Extract protocol $\mathbf{\Pi_{Ext\text{-}Ord}}(A, t)$

Security.
Security of Steps 1, 3, 4, and 5 follow from security of the underlying selection, comparison, and shuffling protocols. By correctness (shown below), the values $\widehat{e}_i$ correspond to the integers from 1 to t, and after calling the $\mathbf{\Pi_{Shuffle}}$ protocol, these values will be randomly arranged in $\widehat{A}$, and so both party's view in Step 6 can be generated uniformly at random by a simulator. All remaining steps are performed locally, and can be imitated by a simulator.

Correctness.
As in $\mathbf{\Pi_{Extract}}$, there are exactly t values in $\{1, \ldots, n+t\}$ with $e_i \neq 0$, the values which will be extracted into B. For the t' non-dummy values from A with $\iota_i = 1$, the index $[e_i]$ counts the number of non-dummy values to the left of a_i in A, and so $[e_i]$ holds the destination index of a_i, and collectively, the indices $[e_i]$ take on the values $\{1, \ldots, t\}$.

The remaining $t - t'$ values in $\{n+1, \ldots, n+t\}$ with $i - n > t'$ are in fact $n + t' + 1, \ldots, n + t$, and so the corresponding values $[e_i]$ are $t'+1, \ldots, t$. Thus the nonzero values of e_i, for $1 \leq i \leq n+t$, collectively cover the values $\{1, \ldots, t\}$, with the first t' values being the non-dummy elements of A, in ordered, as desired.

Cost.
Note that Step 1 requires linear *computation* when implemented via the recursion $[e_i] = [\iota_i] + [e_{i-1}]$, and zero communication, since it can be performed locally. Step 2 requires t comparison operations $\mathbf{\Pi_{Comp}}$ and Step 4 requires $n + t$ revealing operations. Combining with Step 3 gives $O(1) + \mathsf{RCost}(\mathbf{\Pi_{Shuffle}}(n))$ rounds and $\mathsf{CCost}\mathbf{\Pi_{Shuffle}}(n) + t \cdot \mathsf{RCost}(\mathbf{\Pi_{Comp}}) + (n + 2t) \cdot \mathsf{RCost}(\mathbf{\Pi_{Reveal}})$.

8.5 Stable Extraction into Bins

Figure 14 has pseudo-code for our Stable Bin Extract protocol $\mathbf{\Pi_{Ext\text{-}Bin}}(A,t)$.

Stable Bin Extraction Algorithm $\mathbf{\Pi_{Ext\text{-}Bin}}(A, k, t, C, T')$

Input. Two parties $\mathcal{P}_1, \mathcal{P}_2$ (additively) secret-share a *sorted* list A of size n of elements of the form (a_i, ι_i), with $\iota_i \in \{0, \ldots, k\}$, and $t'_j \leq t$ elements have $\iota_i = j$, for each $j = 1, \ldots, k$. Additionally, we are guaranteed that all elements with $\iota_i = j$, for $j \neq 0$, are in a contiguous block (of t'_j consecutive elements). Finally, all parties hold (shares of) another pair of lists $(C = \{c_j\}, T' = \{t_j\})$, each of length k, with the property that c_j is (a share of) the index i of the first element of A with $\iota_i = j$, and t'_j is (a share of) the number of elements of A with $\iota_i = j$ (if no such element with $\iota_i = j$ exists, then c_j and t'_j are (shares of) 0).

Output. Each party holds additive shares of a sequence of lists $\{B_j\}$, for $j = 1, \ldots, k$, where each list B_j has length t and contains all elements a_i from A with $\iota_i = j$, in their original order in A, followed by $(t - t'_j)$ dummy elements.

RCost. $O(1) + \mathsf{RCost}(\mathbf{\Pi_{Shuffle}}(n + tk))$.

CCost. $\Theta(n{+}tk) \cdot (\mathsf{CCost}(\mathbf{\Pi_{Comp}}){+}\mathsf{CCost}(\mathbf{\Pi_{Rev}}){+}\mathsf{CCost}(\mathbf{\Pi_{Sel}})){+}\mathsf{CCost}(\mathbf{\Pi_{Shuf}}(n{+}tk))$.

Protocol.

1. Compute the extracted "within-bucket" position e_i of each element of A. Namely, for any $j \in [1..k]$, the element of A at position $i = c_j$, (which is the first/leftmost element of A with $\iota_i = j$) will have $e_i = 1$, and – by the Input guarantee of ι tags being in contiguous blocks – the next $(t'_j - 1)$ elements will get position tags: $e_{i'} = (1 + i' - i)$. For non-extracted elements of A (whose ι_i tag is *not* in $[1..k]$), define $e_i := 0$.
2. Append two additional tags (e_i, d_i) to each element in A, where e_i is as per Step 1, and $d_i = 1$ represents (shares of) an "is-non-dummy" tag (so all original elements in A have $d_i = 1$).
3. For each $j \in [1..k]$ and for each $i = 1, \ldots, t$, append to A the element:
$$(a_{j,i},\ \iota_{j,i},\ e_{j,i},\ d_{j,i}) := (0,\ j \cdot \delta_{j,i},\ i + t'_j,\ 0),$$
where $\delta_{j,i}$ is an indicator on $i \leq (t - t'_j)$ (so that exactly $(t - t'_j)$ *new* elements will have ι tag equal to j), and e_i denotes the extracted "within-block" index of this element (in case it is to be extracted). Each party sets their share of $a_{j,i}$ and $d_{j,i}$ to 0 and their share of $e_{j,i} = i + t'_j$ using their shares of t'_j, and computes their share of $\iota_{j,i} = j \cdot \delta_{j,i}$ by invoking a secure comparison protocol.
4. Invoke the shuffling protocol, denoting the output $\widehat{A} := \mathbf{\Pi_{Shuffle}}(A)$.
5. Reveal all values $\widehat{\iota_i}$ (exactly t_j of which will have value j, for each $j \in [1..k]$).
6. Reveal all values $\widehat{e_i}$ for the elements that had $\widehat{\iota_i} \in [1..k]$.
7. Initialize $\{B_j\}$ as a sequence of k lists of length t.
8. Set the ℓ^{th} element of B_j equal to the (unique) element of $\widehat{A}$ with $\widehat{\iota_i} = j$ and $\widehat{e_i} = \ell$. This will result in B_j having size t, with t'_j elements having "is-non-dummy" tag $\widehat{d} = 1$, and the remaining $(t - t'_j)$ elements with $\widehat{d} = 0$.

Figure 14: Stably Extract Marked Elements By Bin Protocol

8.6 Analysis of Ordered Bucketed Extract protocol $\mathbf{\Pi_{Ext\text{-}Bin}}(A, t)$.

Security.
A simulator can simulate the view of either party during Steps 1, 3, and 5 by invoking a simulator of the underlying protocols, and can imitate Steps 2, 7, and 8 exactly, since these steps are performed locally. Security for these steps follows from the security of the underlying protocol.

We show in our discussion of correctness that, in Step 6, the parties learn a random permutation of the tk ordered pairs (i, j) with $1 \leq i \leq t$ and $1 \leq j \leq k$, while in Step 4, the parties learn the first coordinates of all terms in that permutation. During Step 4 of a simulated execution of the

protocol, the simulator generates and stores the permutation for Step 6 uniformly at random, and then computes the resulting values for Step 4.

Correctness.
As in $\mathbf{\Pi_{Extract}}$ and $\mathbf{\Pi_{Ext\text{-}Ord}}$, the comparison operation in Step 1 adds $t - t'_j$ elements with $\iota_i = j$. The sequence of values κ_i for these dummy values is $\{t'_j + 1, \ldots, t-1, 0\}$, since the term with $\kappa_i = t$ is reduced modulo t.

Before Step 5, the remaining non-dummy values with $\iota_i = j$ have values κ_i equal to the sequence $\{c_j, c_j + 1, \ldots, c_j + t'_j - 1\}$, taken modulo t, where entry $c_j + (h-1)$ corresponds to the h^{th} element of A with $\iota_i = j$. Therefore, after Step 5, the values κ_i take on the values $\{1, 2, \ldots, t'_j\}$, where entry k corresponds to the h^{th} element of $\iota_i = j$.

Therefore, for $1 \le h \le t$ and $1 \le j \le k$, every pair (h, j) occurs exactly once in Step 6, and, for a fixed j, the first t'_j entries of B_j are the non-dummy elements of A with $\iota_i = j$, in order, as desired.

Cost.
Step 1 requires tk comparisons. Steps 4 and 6 require revealing $(n + tk)$ values. Step 5 requires tk subtractions (which can be performed locally), tk comparisons (to zero), and tk selection operations. Steps 2, 7, and 8 are performed locally. Each of Steps 1, 4, 5, and 6 therefore require $O(1)$ rounds. This gives a total round cost of $O(1) + \mathsf{RCost}(\mathbf{\Pi_{Shuffle}}(n + tk))$ and a total communication cost of $\Theta(n + tk) \cdot (\mathsf{CCost}(\mathbf{\Pi_{Comp}}) + \mathsf{CCost}(\mathbf{\Pi_{Reveal}}) + \mathsf{CCost}(\mathbf{\Pi_{Sel}})) + \mathsf{CCost}(\mathbf{\Pi_{Shuffle}}(n + tk))$, with linear total computation by each party.

9 Other Sub-Protocols

In this section, we present other sub-protocols invoked by any of the Secure Merge protocols above.

9.1 Secure Merge via Compare All

The following protocol (Figure 15) is a naïve protocol that simply performs all n^2 possible comparisons (securely) and is useful for terminating iterative/recursive processes when the reduced list size n is sufficiently small.

Secure Merge - Compare All $\mathbf{\Pi_{SM\text{-}ALL}}(n_1, n_2)$

Input. Two parties $\mathcal{P}_1$, $\mathcal{P}_2$ (additively) secret-share two *sorted* lists L_1 and L_2, of size n_1, n_2, with elements $\ell_{i,j}$, for $j \in \{1, 2\}$. When $n_1 = n_2 = n$, we write $\mathbf{\Pi_{SM\text{-}ALL}}(n)$.

Output. Lists $\widehat{L}_1$, $\widehat{L}_2$, with $\widehat{L}_j = L_j \bowtie (\iota_{i,j})_i$, where $\iota_{i,j}$ denotes the index of $\ell_{i,j}$ in $L_1 \bigsqcup L_2$.

RCost. $O(1)$ rounds.

CCost. $n_1 n_2 \cdot \mathsf{CCost}(\Pi_{Op})$.

Protocol.

1. Compute shares $[e_{i,j}] = [(\ell_{i,1} > \ell_{j,2})]$ for every pair $(i, j) \in \{1, \ldots, n_1\} \times \{1, \ldots, n_2\}$.
2. Locally compute $[\iota_{i,1}] = \sum_{j=1}^{n_2} [e_{i,j}]$ and $[\iota_{i,2}] = n_1 - \sum_{j=1}^{n_1} [e_{j,i}]$.
3. Return $(\widehat{L}_1, \widehat{L}_2) := (L_1 \bowtie (i + \iota_{i,1})_i, L_2 \bowtie (i + \iota_{i,2})_i)$

Figure 15: Secure Merge - Compare All Protocol

9.1.1 Analysis of the $\Pi_{\text{SM-ALL}}(n_1, n_2)$ Protocol

Security.
Security follows from the security of the underlying comparison protocol used in Step 1, since Steps 2 and 3 are performed locally.

Correctness.
By convention, we require elements from L_1 to be to the left of elements from L_2 if their values are equal.

The value $e_{i,j}$ is nonzero precisely when $\ell_{i,1} > \ell_{j,2}$, so $\iota_{i,1}$ counts the number of such values $\ell_{j,2}$. In the merged list $L_1 \bigsqcup L_2$, the entry $\ell_{i,1}$ will have $(i-1)$ elements from L_1 to its left, and $\iota_{i,1}$ elements from L_2 to its left, giving its final position as $i + \iota_{i,1}$.

Similarly $\sum_{j=1}^{n_1} e_{j,i}$ counts values $\ell_{j,1} > \ell_{i,2}$, and so $n_2 - \sum e_{j,i}$ counts the number of values $\ell_{j,1} \leq \ell_{i,2}$, and $i + \iota_{i,2}$ gives the destination index of $\ell_{i,2}$.

Cost.

- Step (1) incurs $2n_1n_2\mathsf{CCost}(\mathbf{\Pi_{Comp}})$ and $O(1)$ round cost (the cost of the underlying comparison protocol), since the shares $[e_{i,j}]$ and $[f_{i,j}]$ can all be computed in parallel, and there are $2n_1n_2$ such comparisons to compute.
- Steps (2) and (3) can be performed locally.

9.2 Duplicate Values Π_{Dup}

Duplicate Values $\mathbf{\Pi_{Dup}}(k, m, t, L)$

Input. Two parties $\mathcal{P}_1$, $\mathcal{P}_2$ (additively) secret-share list $L = \{(a_i, \iota_i)\}$ of size n. Exactly k elements have tag $\iota = t$. Also as input is desired duplication factor m.

Output. Output list L' of size $n + k \cdot (m-1)$ is shared between the two parties, where each element of L that had $\iota = t$ appears replicated m times ($m-1$ *new* elements appearing consecutively after the original element), and otherwise all elements of L' appear in the same order as they did in L. Also, in terms of security, it is okay for the parties to learn the (unordered) set of original tag values $\sigma(\{\iota_i\})$ for some unknown permutation σ, although they should *not* learn the order of these values, i.e. *which* tag values belong to *which* elements.

RCost. $\mathsf{RCost}(\mathbf{\Pi_{Comp}}) \ + \ \mathsf{RCost}(\mathbf{\Pi_{Shuffle}}(n + k \cdot m))$.

CCost. $O(n) \cdot \mathsf{CCost}(\mathbf{\Pi_{Comp}}) \ + \ O(1) \cdot \mathsf{CCost}(\mathbf{\Pi_{Shuffle}}(n + k \cdot m)) \ + \ O((n + k \cdot m))$.

Protocol.

1. By running a secure comparison protocol (in parallel) to compare actual tag values to t, we can reduce to the case that $t = 1$, and all elements that do *not* have this tag value have tag value zero.
2. (Locally) determine (shares of) the final index/position that each element in L will have in L', by keeping a running counter:
$$e_{i+1} = e_i + 1 + (m-1) \cdot \iota_i, \tag{5}$$
where $\iota_i = 1$ if and only if the i^{th} element is to be duplicated. Notice that, by linearity of the secret-sharing scheme, (5) can be computed locally. Append shares of position tag e_i to each element i of L.
3. Invoke a secure shuffling protocol, denoting the output $\widehat{L} := \mathbf{\Pi_{Shuffle}}(L)$.
4. Reveal original tag values: exactly k elements will have revealed tag value $\hat{\iota} = 1$.
5. For each element $(\widehat{a}_i, \widehat{e}_i)$ that has revealed tag $\hat{\iota} = 1$: introduce $(m-1)$ new elements $\{(\widehat{a}_i, \widehat{e}_i + j)\}_{j=1}^{m-1}$ that have the same value $\widehat{a}_i$ and increasing position tags $\widehat{e}_i + j$. Let $\widetilde{L}$ denote the resulting list, which has size $n + k \cdot (m-1)$.
6. Invoke a secure shuffling protocol $\mathbf{\Pi_{Shuffle}}(L)$ on $\widetilde{L}$.
7. Reveal all values $\widetilde{e}_i$, and arrange elements in output list L' as per this revealed (position) tag.

Figure 16: Duplicate Values Protocol

9.2.1 Analysis

Security.

- Step 1: Security follows from the security of the $\mathbf{\Pi_{Comp}}$ protocol.
- Step 2: This is done locally, so is automatically secure.
- Step 3: Security follows from the security of the $\mathbf{\Pi_{Shuffle}}$ protocol.
- Step 4: Leakage of the *set* of tag values is permitted (see 'Output' condition), and so security follows from the security of the $\mathbf{\Pi_{Shuffle}}$ protocol, which guarantees that the shuffle permutation σ is unknown to both parties.
- Step 5: This is done locally, so is automatically secure.
- Step 6: Security follows from the security of the $\mathbf{\Pi_{Shuffle}}$ protocol.
- Step 7: By Correctness of this protocol, the revealed tag values will necessarily be a random permutation on the indices $[1..(n + k \cdot (m-1))]$.

Correctness.
This is immediate based on the formula for computing final positions in (5), which leave a gap of size $(m-1)$ each time an element to be duplicated is encountered, and then in Step 5 these gaps

are exactly filled by the (copies of the) duplicated elements.

Cost.
Costs are computing by adding the contributions of each step, noting that steps involving local computation do not contribute to round complexity:

- Step 1: n invocations (in parallel) of the $\mathbf{\Pi_{Comp}}$ protocol.
- Step 2: $O(n)$ local computation.
- Step 3: $\mathbf{\Pi_{Shuffle}}(n)$ protocol.
- Step 4: $O(n)$ local computation.
- Step 5: $O(n + k \cdot (m-1))$ local computation.
- Step 6: $\mathbf{\Pi_{Shuffle}}(n + k \cdot (m-1))$ protocol.
- Step 7: $O(n + k \cdot (m-1))$ local computation.

10 Results on Medians

Notice that if we write $\mathcal{M}_{j,k} = \{v_1, v_2, \ldots, v_k\}$, then:

$$
\begin{array}{lll}
\text{(a)} & |\mathcal{M}_{j,k}| = k & \\
\text{(b)} & \text{If } k = 1, \text{ then:} & \mathcal{M}_{j,1} = \{u_n\} \text{ (the last element of } L_j) \\
\text{(c)} & \text{If } k = n = |L_j|, \text{ then:} & \mathcal{M}_{j,n} = L_j \\
\text{(d)} & \forall v_i \in \mathcal{M}_{j,k}: & \text{At least } i \cdot \left(\frac{n}{k}\right) \text{ items in } L_j \text{ that are} \leq v_i \\
\text{(e)} & \forall v_i \in \mathcal{M}_{j,k} \text{ and } v_i > v \in L_j: & \text{Less than } i \cdot \left(\frac{n}{k}\right) \text{ items in } L_j \text{ that are} \leq v \\
\text{(f)} & \forall v_i \in \mathcal{M}_{j,k}: & \text{At least } 1 + (k-i) \cdot \left(\frac{n}{k}\right) \text{ items in } L_j \text{ that are} \geq v_i \\
\text{(g)} & \forall v_i \in \mathcal{M}_{j,k} \text{ and } v_i < v \in L_j: & \text{Less than } 1 + (k-i) \cdot \left(\frac{n}{k}\right) \text{ items in } L_j \text{ that are} \geq v \\
\text{(h)} & v_k = u_n & \text{(the last element of } \mathcal{M}_{j,k} \text{ is the last element of } L_j)
\end{array}
\tag{6}
$$

For any value $v \in S$, where S is a totally ordered set, let:

$$
\begin{aligned}
LT_{L_j}(v) &= \text{Number of elements in } L_j \text{ that are less than } v \\
LTE_{L_j}(v) &= \text{Number of elements in } L_j \text{ that are less than or equal to } v \\
GT_{L_j}(v) &= \text{Number of elements in } L_j \text{ that are greater than } v \\
GTE_{L_j}(v) &= \text{Number of elements in } L_j \text{ that are greater than or equal to } v
\end{aligned}
$$

Note that each of the above four functions are monotone step functions with respect to v (increasing for LT and LTE and decreasing for GT and GTE), with jumps at each $v \in L_j$. Then for any $v \in L_j$ and any $v_i \in \mathcal{M}_{j,k}$:

$$n \;=\; LTE_{L_j}(v) + GT_{L_j}(v) \;=\; LT_{L_j}(v) + GTE_{L_j}(v) \tag{7a}$$

$$LTE_{L_j}(v_i) \geq i \cdot \left(\frac{n}{k}\right) \tag{7b}$$

$$LTE_{L_j}(v) < i \cdot \left(\frac{n}{k}\right) \quad \Leftrightarrow \quad v_i > v \tag{7c}$$

$$LTE_{L_j}(v) \geq i \cdot \left(\frac{n}{k}\right) \quad \Leftrightarrow \quad v_i \leq v \tag{7d}$$

$$LTE_{L_j}(v) = i \cdot \left(\frac{n}{k}\right) \quad \Rightarrow \quad v_i = v \tag{7e}$$

$$GTE_{L_j}(v_i) \geq 1 + (k-i) \cdot \left(\frac{n}{k}\right) \tag{7f}$$

$$GTE_{L_j}(v) < 1 + (k-i) \cdot \left(\frac{n}{k}\right) \quad \Leftrightarrow \quad v_i < v \tag{7g}$$

$$GTE_{L_j}(v) \geq 1 + (k-i) \cdot \left(\frac{n}{k}\right) \quad \Leftrightarrow \quad v_i \geq v \tag{7h}$$

$$GTE_{L_j}(v) = 1 + (k-i) \cdot \left(\frac{n}{k}\right) \quad \Rightarrow \quad v_i \leq v \tag{7i}$$

where (7a) is by definition; (7b) is (6.d) and (7f) is (6.f); the left implication of (7c) and (7g) are (6.e) and (6.g), and the right implication is from (7b) and (7f) together with the fact that LTE and GTE are monotone (increasing/decreasing, respectively) functions in v; (7d) and (7h) are the contrapositive of (7c) and (7g), respectively; and (7e) is because $v_i \leq v$ (by (7c)) and also $v_i \geq v$ by (7h) together with (7a) (and similarly (7i) follows from (7a), (7g), and (7d)).

The correctness of the above protocols will rely on the "alignment" property claimed in the Secure Symmetric Merge rubric above. Formally:

Lemma 10.1. *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 (as per (2)). Let $L'_1 = L_1 \bigsqcup \mathcal{M}_{2,k}$ 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:*

$$\mathcal{M}'_{2k} \;= \mathcal{M}_{1,k} \bigsqcup \mathcal{M}_{2,k}$$

Proof. Denote the lists of k medians as:

$$\mathcal{M}_{1,k} = \{u_1, u_2, \ldots, u_k\}$$
$$\mathcal{M}_{2,k} = \{v_1, v_2, \ldots, v_k\}$$

Let $\mathcal{M}_{2k} = \mathcal{M}_{1,k} \bigsqcup \mathcal{M}_{2,k}$, and denote $\mathcal{M}_{2k}$ and $\mathcal{M}'_{2k}$ as:

$$\mathcal{M}_{2k} = \{w_1, w_2, \ldots, w_{2k}\}$$
$$\mathcal{M}'_{2k} = \{z_1, z_2, \ldots, z_{2k}\}$$

Then the claim is that $\mathcal{M}'_{2k} = \mathcal{M}_{2k}$, which we argue by demonstrating that $z_i = w_i$ for all $1 \leq i \leq 2k$. Notice that $|L'_1| = 2n$ and $|\mathcal{M}'_{2k}| = 2k$, and by definition $\mathcal{M}'_{2k}$ evenly partitions L'_1 into $2k$ blocks, with each block of size n/k. Furthermore, notice that L'_1 is the merge of two lists:

- n elements are L_1
- n elements are the k elements of $\mathcal{M}_{2,k}$, each duplicated n/k times.

In particular, if $L'_2 := \bigcup_{\frac{n}{k}} \mathcal{M}_{2,k}$ (i.e. L'_2 is $\mathcal{M}_{2,k}$ duplicated n/k times), then observe that the k medians of L'_2 are exactly the same as the k medians of L_2, namely $\mathcal{M}_{2,k}$. Therefore, if (as *multi-sets*): $L'_1 = L_1 \cup L'_2$, then (as per (6)):

$$\begin{array}{rll} \text{For any } u_i \in \mathcal{M}_{1,k}\text{:} & \text{At least } i \cdot \left(\frac{n}{k}\right) & \text{items in } L'_1, \textbf{\textit{from}}\ \boldsymbol{L_1}, \text{ are} \leq u_i. \\ & \text{At least } 1 + (k\text{-}i) \cdot \left(\frac{n}{k}\right) & \text{items in } L'_1, \textbf{\textit{from}}\ \boldsymbol{L_1}, \text{ are} \geq u_i. \\ \text{For any } v_i \in \mathcal{M}_{2,k}\text{:} & \text{At least } i \cdot \left(\frac{n}{k}\right) & \text{items in } L'_1, \textbf{\textit{from}}\ \boldsymbol{L'_2}, \text{ are} \leq v_i. \\ & \text{At least } (1 + k\text{-}i) \cdot \left(\frac{n}{k}\right) & \text{items in } L'_1, \textbf{\textit{from}}\ \boldsymbol{L'_2}, \text{ are} \geq v_i. \end{array} \tag{8}$$

Notice that the last statement in (8) (unlike the second statement) does not come directly from (6.e). In particular, it relies on the fact that the medians of L'_2 are exactly described by $\mathcal{M}_{2,k}$ (the medians of L_2), and each such median appears a total of n/k times in L'_2. This is why the '1' appears *inside* the parentheses (as a multiplicative factor of the n/k) as opposed to outside (as an additive constant). We conclude the proof by showing that $z_i = w_i$ for all $1 \leq i \leq 2k$ by showing both inequalities:

$\underline{z_i \geq w_i}$. By (7d), this will follow if we can show that there are at least $i \cdot \frac{2n}{2k}$ values in L'_1 that are less than or equal to w_i. Let $x = x_i$ denote the maximal index such that $u_x \in \mathcal{M}_{1,k}$ appears *among the first i coordinates* of $\mathcal{M}_{2k}$. Similarly, let $y = y_i$ denote the the maximal index such that $v_y \in \mathcal{M}_{2,k}$ appears *among the first i coordinates* of $\mathcal{M}_{2k}$. By definition of $\mathcal{M}_{2k}$ (as the merge of $\mathcal{M}_{1,k}$ and $\mathcal{M}_{2,k}$), we have that $x_i + y_i = i$. And then by (8) above, we have that there are at least $x_i \cdot \frac{n}{k}$ elements in L_1 that are less than or equal to u_x, and since u_x is in the first i coordinates of $\mathcal{M}_{2k}$, we have that $u_x \leq w_i$, and hence (by monotonicity of LTE) there are at least $x_i \cdot \frac{n}{k}$ elements in L_1 that are less than or equal to w_i. Similarly, there are at least $y_i \cdot \frac{n}{k}$ elements in L'_2 that are less than or equal to w_i. Consequently, there are at least $(x_i + y_i) \cdot \frac{n}{k} = i \cdot \frac{n}{k}$ values in L'_1 that are less than or equal to w_i, as required.

$\underline{z_i \leq w_i}$. By (7h), this will follow if we can show that there are at least $1 + (2k - i) \cdot \frac{2n}{2k}$ values in L'_1 that are greater than or equal to w_i. Let $x = x_i$ denote the *minimal* index such that $u_x \in \mathcal{M}_{1,k}$ appears *at or after coordinate i* of $\mathcal{M}_{2k}$ (or define $x_i = k + 1$ if none of the values in $\mathcal{M}_{1,k}$ appear at or after coordinate i of $\mathcal{M}_{2k}$). Similarly, let $y = y_i$ denote the the minimal index such that $v_y \in \mathcal{M}_{2,k}$ appears *at or after coordinate i* of $\mathcal{M}_{2k}$ (or define $y_i = k + 1$ if none of the values in $\mathcal{M}_{2,k}$ appear at or after coordinate i of $\mathcal{M}_{2k}$). By definition of $\mathcal{M}_{2k}$ (as the merge of $\mathcal{M}_{1,k}$ and $\mathcal{M}_{2,k}$), we have that $x_i + y_i = i + 1$. By (8) above, we have that there are at least $\max(0, 1 + (k - x_i) \cdot \frac{n}{k})$ elements in L_1 that are greater than or equal to u_x, and since u_x has index at least i in $\mathcal{M}_{2k}$, we have that $u_x \geq w_i$, and hence (by monotonicity of GTE) there are at least $\max(0, 1 + (k - x_i) \cdot \frac{n}{k})$ elements in L_1 that are greater than or equal to w_i. Similarly, there are at least $(1 + k - y_i) \cdot \frac{n}{k}$ elements in L'_2 that are greater than or equal to w_i. Consequently, there are at least $1 + (1 + 2k - (x_i + y_i)) \cdot \frac{n}{k} = 1 + (2k - i) \cdot \frac{n}{k}$ values in L'_1 that are greater than or equal to w_i, as required. □

Acknowledgements

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.

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.

References

[1] Ajtai, M., Komlós, J., Szemerédi, E.: An o(n log n) sorting network. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing. p. 1–9. STOC '83, Association for Computing Machinery, New York, NY, USA (1983). https://doi.org/10.1145/800061.808726

[2] Arasu, A., Kaushik, R.: Oblivious query processing. In: Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014. pp. 26–37. OpenProceedings.org (2014). https://doi.org/10.5441/002/icdt.2014.07

[3] Asharov, G., Komargodski, I., Lin, W., Nayak, K., Peserico, E., Shi, E.: Optorama: Optimal oblivious RAM. In: Advances in Cryptology – EUROCRYPT 2020. pp. 403–432. Springer (2020). https://doi.org/10.1007/978-3-030-45724-2_14

[4] Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. p. 307–314. AFIPS '68 (Spring), Association for Computing Machinery, New York, NY, USA (1968). https://doi.org/10.1145/1468075.1468121

[5] Borodin, A., Hopcroft, J.E.: Routing, merging, and sorting on parallel models of computation. J. Comput. Syst. Sci. **30**(1), 130–145 (1985). https://doi.org/10.1016/0022-0000(85)90008-X

[6] Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000). https://doi.org/10.1007/s001459910006

[7] Chan, T.H., Katz, J., Nayak, K., Polychroniadou, A., Shi, E.: More is less: Perfectly secure oblivious algorithms in the multi-server setting. In: Peyrin, T., Galbraith, S.D. (eds.) Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11274, pp. 158–188. Springer (2018). https://doi.org/10.1007/978-3-030-03332-3_7

[8] Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 1243–1255. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134061

[9] Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Kiribuchi, N., Pinkas, B.: An efficient secure three-party sorting protocol with an honest majority. IACR Cryptol. ePrint Arch. p. 695 (2019), `https://eprint.iacr.org/2019/695`

[10] Cristofaro, E.D., Tsudik, G.: Practical private set intersection protocols with linear complexity. In: Sion, R. (ed.) Financial Cryptography and Data Security, 14th International

Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6052, pp. 143–159. Springer (2010). https://doi.org/10.1007/978-3-642-14577-3_13

[11] Debnath, S.K., Dutta, R.: Secure and efficient private set intersection cardinality using bloom filter. In: Information Security. pp. 209–226. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23318-5_12

[12] Eskandarian, S., Zaharia, M.: Oblidb: Oblivious query processing for secure databases. Proc. VLDB Endow. **13**(2), 169–183 (oct 2019). https://doi.org/10.14778/3364324.3364331

[13] Falk, B.H., Nema, R., Ostrovsky, R.: A linear-time 2-party secure merge protocol. In: Dolev, S., Katz, J., Meisels, A. (eds.) Cyber Security, Cryptology, and Machine Learning - 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30 - July 1, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13301, pp. 408–427. Springer (2022). https://doi.org/10.1007/978-3-031-07689-3_30

[14] Falk, B.H., Ostrovsky, R.: Secure merge with o(n log log n) secure operations. In: 2nd Conference on Information-Theoretic Cryptography (ITC 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 199, pp. 7:1–7:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ITC.2021.7

[15] Freedman, M.J., Ishai, Y., Pinkas, B., Reingold, O.: Keyword search and oblivious pseudorandom functions. In: Theory of Cryptography. pp. 303–324. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), https://doi.org/10.1007/978-3-540-30576-7_17

[16] Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing oram and using it efficiently for secure computation. In: Privacy Enhancing Technologies. pp. 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39077-7_1

[17] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing. p. 218–229. STOC '87, Association for Computing Machinery, New York, NY, USA (1987). https://doi.org/10.1145/28395.28420

[18] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996). https://doi.org/10.1145/233551.233553

[19] Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious ram simulation. In: Automata, Languages and Programming. pp. 576–587. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22012-8_46

[20] Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. IACR Cryptol. ePrint Arch. p. 121 (2014), http://eprint.iacr.org/2014/121

[21] Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multiparty sorting protocols from comparison sort algorithms. In: International Conference on Information Security and Cryptology. pp. 202–216. Springer (2013), https://doi.org/10.1007/978-3-642-37682-5_15

[22] Hayashi, T., Nakano, K., Olariu, S.: Work-time optimal k-merge algorithms on the PRAM. IEEE Trans. Parallel Distributed Syst. **9**(3), 275–282 (1998). https://doi.org/10.1109/71.674319

[23] Hong, Z., Sedgewick, R.: Notes on merging networks (prelimiary version). In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing. p. 296–302. STOC '82, Association for Computing Machinery, New York, NY, USA (1982). https://doi.org/10.1145/800070.802204

[24] Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012. The Internet Society (2012), `https://www.ndss-symposium.org/ndss2012/private-set-intersection-are-garbled-circuits-better-custom-protocols`

[25] Huberman, B.A., Franklin, M., Hogg, T.: Enhancing privacy and trust in electronic communities. In: Proceedings of the 1st ACM Conference on Electronic Commerce. p. 78–86. EC '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/336992.337012

[26] Ishai, Y., Kushilevitz, E., Lu, S., Ostrovsky, R.: Private large-scale databases with distributed searchable symmetric encryption. In: Topics in Cryptology - CT-RSA 2016. pp. 90–107. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-29485-8_6

[27] Jiang, Y., Wei, J., Pan, J.: Publicly verifiable private set intersection from homomorphic encryption. In: Security and Privacy in Social Networks and Big Data. pp. 117–137. Springer Nature Singapore, Singapore (2022). https://doi.org/10.1007/978-981-19-7242-3_8

[28] Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. IACR Cryptol. ePrint Arch. p. 122 (2011), `http://eprint.iacr.org/2011/122`

[29] Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious prf with applications to private set intersection. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. p. 818–829. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978381

[30] Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)Security of Hash-Based Oblivious RAM and a New Balancing Scheme, p. 143–156. SODA '12, Society for Industrial and Applied Mathematics, USA (2012). https://doi.org/10.1137/1.9781611973099.13

[31] Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious ram lower bound! In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 523–542. Springer International Publishing, Cham (2018)

[32] Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Information Security. pp. 262–277. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24861-0_18

[33] Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. In: Theory of Cryptography. pp. 377–396. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_22

[34] Meadows, C.A.: A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In: Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 7-9, 1986. pp. 134–137. IEEE Computer Society (1986). https://doi.org/10.1109/SP.1986.10022

[35] Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer Publishing Company, Incorporated, 1 edn. (2008), https://doi.org/10.1007/978-3-540-77978-0

[36] Nishide, T., Ohta, K.: Constant-round multiparty computation for interval test, equality test, and comparison. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **90-A**(5), 960–968 (2007). https://doi.org/10.1093/ietfec/e90-a.5.960

[37] Patel, S., Persiano, G., Raykova, M., Yeo, K.: Panorama: Oblivious ram with logarithmic overhead. In: 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS). pp. 871–882 (2018). https://doi.org/10.1109/FOCS.2018.00087

[38] Pinkas, B., Reinman, T.: Oblivious ram revisited. In: Advances in Cryptology – CRYPTO 2010. pp. 502–519. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_27

[39] Pinkas, B., Schneider, T., Zohner, M.: Faster private set intersection based on OT extension. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014. pp. 797–812. USENIX Association (2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pinkas

[40] Pinkas, B., Schneider, T., Zohner, M.: Scalable private set intersection based on ot extension. ACM Trans. Priv. Secur. **21**(2) (jan 2018). https://doi.org/10.1145/3154794

[41] Shamir, A.: On the power of commutativity in cryptography. In: Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings. Lecture Notes in Computer Science, vol. 85, pp. 582–595. Springer (1980). https://doi.org/10.1007/3-540-10003-2_100

[42] Shi, E., Chan, T.H., Stefanov, E., Li, M.: Oblivious RAM with o((logn)3) worst-case cost. In: Advances in Cryptology – ASIACRYPT 2011. Lecture Notes in Computer Science, vol. 7073, pp. 197–214. Springer (2011). https://doi.org/10.1007/978-3-642-25385-0_11

[43] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: An extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. p. 299–310. CCS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2508859.2516660

[44] Tajima, A., Sato, H., Yamana, H.: Outsourced private set intersection cardinality with fully homomorphic encryption. In: 2018 6th International Conference on Multimedia Computing and Systems (ICMCS). pp. 1–8 (2018). https://doi.org/10.1109/ICMCS.2018.8525881

[45] Valiant, L.G.: Parallelism in comparison problems. SIAM Journal on Computing **4**(3), 348–355 (1975). https://doi.org/10.1137/0204030

[46] Wang, X., Chan, H., Shi, E.: Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 850–861. CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813634

[47] Yao, A.C., Yao, F.F.: Lower bounds on merging networks. J. ACM **23**(3), 566–571 (1976). https://doi.org/10.1145/321958.321976

[48] Zheng, W., Dave, A., Beekman, J.G., Popa, R.A., Gonzalez, J.E., Stoica, I.: 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. pp. 283–298. USENIX Association (2017), `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng`