A Linear-Time 2-Party Secure Merge Protocol

Brett Hemenway Falk[1], Rohit Nema[2](✉), and Rafail Ostrovsky[2]

[1] University of Pennsylvania, Philadelphia, USA
fbrett@cis.upenn.edu
[2] UCLA, Los Angeles, USA
rnema@ucla.edu, rafail@cs.ucla.edu

Abstract. We present a linear-time, space and communication *data-oblivious* algorithm for securely merging two private, sorted lists into a single sorted, secret-shared list in the *two* party setting. Although merging two sorted lists can be done *insecurely* in linear time, previous *secure* merge algorithms all require super-linear time and communication. A key feature of our construction is a novel method to *obliviously* traverse permuted lists in sorted order. Our algorithm only requires black-box use of the underlying Additively Homomorphic cryptosystem and generic secure computation schemes for comparison and equality testing.

Keywords: Secure computation · Homomorphic encryption · Oblivious protocols

1 Introduction

Securely merging two sorted lists into a single, globally sorted list with the same asymptotic complexity as in the insecure setting has been a long-standing open problem. It is a fundamental tool in many machine learning and data-processing applications [6,42,57], Oblivious RAM [31,45], and Private Set Intersection (PSI) [36]. A series of works [1,13,32,33] have shown that securely *sorting* a list can be done with the same asymptotic complexity as insecure sorting. On the other hand, for *merging*, a gap remains. In the past, it has been solved with complicated techniques that either run in super-linear time or communication, or make unnatural assumptions.

In the insecure setting, and in the three-party ORAM setting, where there are three servers and a *trusted* client, merging two sorted lists of length n can be done in $O(n)$ time, [10], whereas in the *secure* setting, the best existing 2-party secure merge algorithm requires $O(n \log\log n)$ communication [26].

Our main result is to close this gap. More explicitly, we show

Theorem 1 (Main Theorem). *There exists a 2-party protocol for merging two locally sorted lists in linear-time, space and communication that provides*

B. H. Falk, R. Nema and R. Ostrovsky—Work done while consulting for Stealth Software Technologies, Inc.

S. Dolev et al. (Eds.): CSCML 2022, LNCS 13301, pp. 408–427, 2022.
https://doi.org/10.1007/978-3-031-07689-3_30

security against semi-honest adversaries. The protocol only requires black-box use of an Additively-Homomorphic cryptosystem and a generic secure computation protocol for comparison and equality-testing on secret shares.

Secure 2-party merge protocols arise naturally, since the two participants can each sort their list locally before the protocol begins. Three-party protocols for secure merge are less natural, since there are still only two lists being merged, but these lists are secret-shared amongst the *three* computation parties. If the two lists being merged were initially held in the clear by two parties, then it's unnatural to require a third party to aid in the secure merge procedure. On the other hand, if the two lists were initially secret-shared among two parties (e.g. as the output of a previous 3-party computation) it becomes less natural to assume that they are pre-sorted (since they cannot have been sorted locally).

One application of two-party merge protocols is in Private Set Intersection (PSI). There are many PSI protocols, but most output the intersection *in the clear* (e.g. [11,12,16,20–22,24,28,35,37–41,47,48,51–53]). In many applications, however, PSI is only a first step in a larger computation, and in these settings the PSI must return *secret shares* of the intersection, rather than the list itself – but these secret-shared PSI protocols (e.g. [17,49,50]) tend to be less efficient than protocols that reveal the intersection in the clear. One of the earliest methods for secret-shared PSI is the *sort-compare* paradigm [36], where the participants sort their joint list, then compare adjacent elements in a linear pass, deleting singletons. The problem with this approach is that the initial sorting step takes $O(n \log n)$ communication. Using our novel linear-time secure merge protocol, the sort-compare paradigm gives a simple, efficient *linear-communication* secret-shared PSI protocol.

Our protocol is inspired by the 3-server ORAM merge protocol of [10], where the two sorted lists are treated as linked lists, then each linked list is shuffled with a collection of "dummy" elements using a linear-time three-party secure shuffle [43]. Thereafter, the trusted client can traverse the shuffled linked lists, comparing one element at a time, as in the standard insecure merge protocol.

There are several obstacles that need to be overcome in order to eliminate the trusted client and one of the servers from the [10] merge protocol. We can use a linear-time 2-party secure shuffle [26] to replace the 3-party shuffle, but updating the pointers in the shuffled lists is challenging without a trusted client.

To overcome this obstacle, we develop a technique for converting values encrypted under the key of one participant into additive secret shares of the same underlying plaintext (See Sect. 5.2). This conversion process is extremely efficient, and only relies on the cryptosystem being additively homomorphic. Moreover, the trusted client in [10] can easily switch from the real to dummy list obliviously once the real list is exhausted; however, this is non-trivial in our 2-party setup since obviously neither party should learn when a real list has been exhausted. We combat this issue by creating a unique, partially circular linked list (Sect. 5.1 and Fig. 1) such that the protocol can seamlessly switch from the real to dummy list.

Using this novel linked list construction and ciphertext-to-secret-sharing tool, we give a two party secure merge protocol, where each participant treats their input as a linked list, then allows the other participant to shuffle this linked list (while updating the pointers). The parties then re-share these permuted linked lists, and compare elements one at a time (using a secure comparison protocol), while the exact sequence of data accesses from each list is independent of the underlying data. See Sect. 5 for the full construction.

The detailed security proofs and analysis are presented in the full version of the paper [25].

2 Previous Work

2.1 Secure Sorting

Merging two sorted lists can be seen as a special case of sorting, and thus any sorting protocol is also a merge protocol. When security is not required, a simple counting argument shows that any comparison-based sorting algorithm requires $O(n \log n)$ comparisons, whereas two sorted lists can be merged using only $O(n)$ comparisons. Although secure merge protocols are a building block for many secure multiparty computations, most applications focus on the more general (and more difficult) problem of secure sorting.

One route for building a secure sorting protocol is to securely implement a data-oblivious *sorting network* using a generic circuit-based secure multiparty computation (MPC) protocol (e.g. GMW [30], BGW [7] or Garbled Circuits [59,60]). Asymptotically, the best sorting network is the AKS network [1], which requires $O(n \log n)$ comparisons. Although the AKS network is asymptotically optimal, the hidden constants are *extremely* large [2], and so the AKS network has little practical value. In practice, Batcher's bitonic sort [5] which requires $O(n \log^2 n)$ comparisons is much faster and is widely implemented in practice, including in the ABY [23], Obliv-C [61] and EMP [58] compilers. Batcher's sorting network is defined recursively, and thus when using Batcher's network to merge two pre-sorted input lists, all but the final level of the recursion can be omitted. Unfortunately, this does not improve the asymptotic complexity, but it does increase the concrete performance by about a factor of 2.

One problem with implementing traditional sorting algorithms (e.g. quick-sort, mergesort, radix sort) using generic secure computation, is that the they are not data-oblivious – even if the comparisons are implemented securely, the *data movement* depends on the underlying values being sorted. The *shuffle-then-sort* paradigm [13,32,33], solves this problem by first *obliviously shuffling* the input lists, then securely executing a traditional sorting algorithm. The initial shuffle ensures that the data movement (which is not hidden by the secure computation) is independent of the underlying data. These techniques yield an asymptotically optimal ($O(n \log n)$) sorting algorithms, that are also efficient in practice.

The efficiency of the shuffle-then-sort paradigm rests on the efficiency of the secure shuffle protocol. In the 3-party setting there are linear-time secure shuffles (based on one-way functions) [43], and in the 2-party there are linear-time secure shuffles (based on additively homomorphic encryption) [29].

Applying the shuffle-then-sort paradigm to the problem of merging immediately yields $O(n \log n)$-communication oblivious merge protocols, but does *not* achieve the $O(n)$-time merging that is possible in the insecure setting. In fact, the $\Omega(n \log n)$ lower bound on comparison-based sorting means that this approach will never yield a linear-time secure *merge* algorithm – unless we can take advantage of the fact that the initial lists being merged are pre-sorted.

Alternative sorting schemes (e.g. Radix sort) avoid the $\Omega(n \log n)$ lower bounds on comparison-based sorting. Another example is [34] in the randomized setting which sorts integers in $O(n\sqrt{\log \log n})$ expected running time. These sorting algorithms are efficient, but rely on the RAM model of computation, and their data-dependent access patterns cannot be efficiently implemented in the circuit model. One exception is [4], which uses non-comparison based techniques to beat the $\Omega(n \log n)$ lower bound, but still remains in the circuit model.

2.2 Secure Merging

Secure, multiparty merge protocols have been studied separately from secure sorting protocols, and just as in the insecure case, focusing on the problem of *merging* allows us to circumvent the $\Omega(n \log n)$ lower bound for sorting.

The first secure merge protocol with (asymptotically) less communication than a corresponding secure sort was given in the 3-server ORAM setting (which requires 3-servers and a trusted client), where there is an information-theoretic secure merge protocol with only $O(n)$ communication [10]. In general, any k-server ORAM protocol, the client can be emulated using secure multiparty computation (MPC), thus the protocol of [10] also yields a 3-server secure merge protocol. Unfortunately, using MPC to securely emulate an ORAM client can dramatically hurt performance since the ORAM client may not be "MPC friendly", e.g. the client may have a very large circuit complexity, which leads inefficiencies when emulating the ORAM client under MPC.

The key idea of [10] is to apply "shuffle-then-sort" [13,32,33] to the idea of merging. Essentially, the participants shuffle the two (sorted) linked-lists – updating the pointers to each element's new, shuffled location. Then the participants apply a standard (non-oblivious) merge protocol to traverse these shuffled linked lists (without needing to hide the data movement). These techniques yield a linear-communication secure merge protocol, but the construction of [10] only works in the 3-party ORAM setting, i.e., when there are four parties, three servers and a trusted client.

The "shuffle-then-merge" paradigm is a bit more delicate than the "shuffle-then-sort" paradigm, since the input lists in a merge are pre-sorted, and the merge protocol must process them in this sorted order (even after the oblivious shuffle). To overcome this difficulty, the pre-sorted input lists can be turned into *linked lists*, and the oblivious shuffle can update each item's pointer to point to the permuted position of its successor [10].

In the two-party setting, [26] gives a protocol based on additively homomorphic encryption that securely merges two lists using $O(n \log \log n)$ communication. The key idea of [26] is that since the input lists are pre-sorted, we can divide the entire list into poly-logarithmic sized blocks, and focus on moving these blocks into (nearly) the correct positions. Once the large blocks are in place, the small number of remaining "strays" that are out of place, can be identified and moved efficiently.

Although our solution is fundamentally different, like [26], we also rely on a linear-time 2-party shuffle.

Our protocol follows a shuffle-then-merge paradigm that is similar to [10], but in order to adapt this to the two-party setting, we create a new protocol for shuffling linked lists in the two-party setting (which can be seen as an extension of the two-party oblivious shuffle of [26,29]).

3 Overview

3.1 Challenges

In the insecure setting, two parties can merge their locally sorted lists by simply comparing their smallest elements and advancing the list with the smaller element. This operation is linear in the length of the two lists. The core issue in translating this linear-time merge algorithm to a secure version is that advancing a list is not *data-oblivious* – it reveals which list contained the smaller element.

Protocol 1. A basic, *data-dependent* merge.

```
Input: Two sorted input lists $A$, $B$ of lengths $n_A$ and $n_B$
Output: A sorted output list $C$ of length $n_A + n_B$
 1: Initialize $i_A = i_B = i_C = 0$
 2: while $i_C < n_A + n_B$ do
 3:     if $A[i_A] < B[i_B]$ or $i_B \geq n_B$ then
 4:         $C[i_c] = A[i_A]$
 5:         $i_A = i_A + 1$
 6:     else
 7:         $C[i_c] = B[i_B]$
 8:         $i_B = i_B + 1$
 9:     end if
10:     $i_C = i_C + 1$
11: end while
```

There are two key challenges when trying to adapt the non-oblivious naïve merge protocol (Protocol 1), into an oblivious variant.

1. **Which list is being accessed**: Whether the algorithm reaches Line 4 or Line 7 reveals which *list* is being accessed.
2. **Which location is being accessed**: When the algorithm reaches Line 4 (resp. Line 7), it reveals which element of A's (resp. B's) list is being accessed at iteration i_C.

We also face an additional challenge: we have only *two* participants in the protocol unlike these prior works which had three, either two servers and a trusted client [44] or three servers and trusted client [10].

3.2 Intuition and Construction Overview

Oblivious Shuffle with Linked List: To address challenge 2, we rely on an oblivious permutation. In the multiparty setting, it is possible to perform efficient (linear-time), oblivious shuffles of secret-shared lists [43]. Similarly, in the two-party scenario, the participants can use additively homomorphic encryption to obliviously shuffle ciphertexts in linear time [26,29]. These linear-time multiparty shuffles are a key building block of many secure multiparty sorting protocols [13,32,33], and secure merge algorithms [10,26].

By viewing each participant's sorted input as a linked list, then shuffling that list, the parties can decouple the locations being accessed from the iteration of the loop – for example, at Line 4 the protocol would read location $\Pi_A(i_A)$ for some random permutation Π_A, instead of directly reading i_A.

There are some subtleties here, as the parties need to obliviously permute their linked lists, and then obliviously traverse them.

In order to allow the parties to traverse the permuted linked lists in the original (sorted) order, the parties must also update the pointers. Thus if π is a permutation of $[n]$, and the original list is $(v[0], \ldots, v[n-1])$, the parties will create two new arrays

$$\begin{aligned} w &= \left(v\left[\pi^{-1}(0)\right], \ldots, v\left[\pi^{-1}(n-1)\right]\right) && \text{Permuted data} \\ t &= \left(\pi\left(\pi^{-1}(0)+1\right), \ldots, \pi\left(\pi^{-1}(n-1)+1\right)\right) && \text{Permuted tags} \end{aligned}$$

With $t[\pi(n-1)] = \perp$. Thus if $w[i] = v[j]$, then $w[t[i]] = v[j+1]$.

This structure allows the parties to traverse the permuted list, w, by first revealing $\pi(0)$ and then, selectively revealing elements of t, starting with $t[\pi(0)]$, $t[\pi(1)]$, ...

Our goal is for each party to achieve a secret-shared, permutation of their own list permuted (as well as the updated pointers) by the other party. In our construction, the second party acts as a *permuting* party for the first and generates both the permuted list and the corresponding linked list to traverse it. To maintain privacy of the data and obliviousness of the memory accesses, the second party's permutation, and the first party's data must remain private.

Now, if the permuting party holds on to its share of the owner party's list, it is not clear how to obliviously traverse the permutation since the permuting party knows the position of each accessed share, and thus each element.

When there are three participants this can be done information-theoretically, by having each participant generate a permutation and secret-share to the other two participants [10]. In the two party setting, we can use additively homomorphic encryption to (obliviously) permute a private list [26,29], but we cannot use those constructions in a black-box manner, since they do not allow us to create the shared tags needed to traverse the permuted list.

Instead, we recombine the shares at the owner party but to maintain obliviousness, i.e. to hide the data itself so as to not leak the permutation, both parties somehow convert their shares into shares encrypted using the *permuting party's* public key. The owner party can then use the additive homomorphism of the encryption scheme to add the encrypted shares and obtain an encryption of the element under the *other* (permuting) party's public key. Therefore, it cannot decrypt to learn the underlying value (and thus, permutation).

Adding Dummies and Oblivious Pointer Advancement: To address challenge 1, we add "dummy" elements to each party's list so that we are able to advance both lists every iteration of the loop. For simplicity, suppose both parties' lists are of size n. Then, both parties can generate n dummy elements and maintain two separate pointers to keep track of the real and dummy elements respectively. These dummies are interspersed with the real elements to create a list of $2n$ elements. At every iteration, the party with the smaller element advances its real pointer, while the other party advances its dummy pointer. This ensures that an element is consumed from both lists every iteration of the merge.

Finally, we are left with two more operations: (1) comparing encrypted real values efficiently and (2) advancing lists obliviously. We achieve (1) using a trick to convert ciphertexts into secret shares which can be passed to any *state-of-the-art* 2-party comparison protocol [18,55] to avoid executing an expensive decryption circuit jointly; and we accomplish (2) by a clever construction of the linked list. The detailed shuffle and merge protocol is shown in Sect. 5.

4 Preliminaries

4.1 Secret Sharing

Our protocol makes use of an additive secret sharing scheme, where a secret $x \in \mathcal{G}$ is shared as $(x - r, r)$, for some random $r \leftarrow \mathcal{G}$ where $\mathcal{G}$ is the finite group that parameterizes the Group Homomorphic Encryption scheme. In the two-party setting all linear secret-sharing schemes are essentially equivalent [19], so we can focus on this scheme without loss of generality.

As is standard, we use the notation $[\![x]\!]$ to denote a secret sharing of the plaintext x. Using the linearity of the secret sharing scheme, the participants can compute $[\![x + y]\!]$ from $[\![x]\!]$ and $[\![y]\!]$ with *no communication*.

For more complex calculations on shares, we rely on secure multiparty computation (MPC), described below.

4.2 Secure Computation

Our protocol makes use of a few simple primitives for processing on secret shares, *comparisons*, *multiplexing* and *equality tests*. These basic primitives are implemented in essentially every secure computation framework, including ABY [23], EMP [58], SCALE-MAMBA [3] and MPyC [54].

We assume that there is an underlying ordering on the elements of $\mathcal{G}$ – this is a necessary assumption since the parties want to *sort* their elements.

Our construction is compatible with both arithmetic and boolean secure computation protocols, although comparisons and equality tests are likely to be more efficient in boolean-circuit-based secure computation protocols.

4.3 Additively Homomorphic Encryption

Our construction makes use of a semantically secure, additively homomorphic cryptosystem, $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add})$. Our system is compatible with classical

Comparisons

$$[\![x < y]\!] = \begin{cases} [\![0]\!] & \text{if } x \geq y \\ [\![1]\!] & \text{if } x < y \end{cases}$$

Multiplexing

$$\mathsf{mux}\left([\![b]\!], [\![x]\!], [\![y]\!]\right) = \begin{cases} [\![x]\!] & \text{if } b = 0 \\ [\![y]\!] & \text{if } b = 1 \end{cases}$$

Multiplexes are often implemented as a simple multiplication

$$\mathsf{mux}\left([\![b]\!], [\![x]\!], [\![y]\!]\right) = [\![x]\!] + [\![b]\!] \cdot \left([\![y]\!] - [\![x]\!]\right)$$

Equality tests

$$[\![x = y]\!] = \begin{cases} [\![0]\!] & \text{if } x \neq y \\ [\![1]\!] & \text{if } x = y \end{cases}$$

additively homomorphic schemes like Paillier [46], or lattice-based schemes that natively work over $\mathbb{Z}/2\mathbb{Z}$, e.g. BFV [9,27] or CGGI [14,15], both of which are widely supported by current FHE implementations [56]. Note that the security we require for the $\mathsf{Add}(\cdot,\cdot,\cdot)$ is much weaker than full circuit privacy [8], since in our application the summations being computed are known to both parties, and only the *summands* are private.

In order for our final merge protocol to achieve linear communication, the underlying additively homomorphic cryptosystem must have *constant ciphertext expansion.*

4.4 Notation

As there are only two parties, and each party has a unique public key (for the additively homomorphic cryptosystem), when we say "key i" we mean the public key of party i, pk_i.

We denote each party as P_i where $i \in \{0, 1\}$. As all our protocols are two-party protocols (and most are completely symmetric), we take all subscripts modulo 2, thus if P_i is one party, P_{i+1} is the other party.

Several protocols below must be run twice, one time for each party, so we give such protocols an index with respect to which we write the steps within the protocol. For example, $\mathsf{Protocol}_i$ will be called twice, for $i \in \{0, 1\}$ and we use index i within the protocol to identify the parties. Similarly, we use the same index to define the ideal functionality.

We introduce some more notation in Table 1.

Additively Homomorphic Encryption

Semantic security: for all $x, y \in \mathcal{G}$

$$\left\{ (pk, c_x) : \begin{array}{l} pk, sk \leftarrow \mathsf{KeyGen}\left(1^\lambda\right) \\ c_x \leftarrow \mathsf{Enc}(pk, x) \end{array} \right\} \approx_c \left\{ (pk, c_y) : \begin{array}{l} pk, sk \leftarrow \mathsf{KeyGen}\left(1^\lambda\right) \\ c_y \leftarrow \mathsf{Enc}(pk, y) \end{array} \right\}.$$

Security of Add**:** for all $x, y \in \mathcal{G}$

$$\left\{ \begin{array}{c} c, \\ c_x, c_y, \\ pk, sk \end{array} : \begin{array}{l} pk, sk \leftarrow \mathsf{KeyGen}\left(1^\lambda\right) \\ c_x \leftarrow \mathsf{Enc}(pk, x) \\ c_y \leftarrow \mathsf{Enc}(pk, y) \\ r \leftarrow \mathcal{G} \\ c_r \leftarrow \mathsf{Enc}(pk, r) \\ c \leftarrow \mathsf{Add}(pk, c_x, c_r) \end{array} \right\} \approx_c \left\{ \begin{array}{c} c, \\ c_x, c_y, \\ pk, sk \end{array} : \begin{array}{l} pk, sk \leftarrow \mathsf{KeyGen}\left(1^\lambda\right) \\ c_x \leftarrow \mathsf{Enc}(pk, x) \\ c_y \leftarrow \mathsf{Enc}(pk, y) \\ r \leftarrow \mathcal{G} \\ c_r \leftarrow \mathsf{Enc}(pk, r) \\ c \leftarrow \mathsf{Add}(pk, c_y, c_r) \end{array} \right\}.$$

Decrypting the sum of two ciphertexts yields nothing about the individual summands.

Correctness: for any $x, y \in \mathcal{G}$, and $c > 0$

$$\Pr\left[\left\{ \mathsf{Dec}(sk, c_{x+y}) : \begin{array}{l} pk, sk \leftarrow \mathsf{KeyGen}\left(1^\lambda\right) \\ c_x \leftarrow \mathsf{Enc}(pk, x) \\ c_y \leftarrow \mathsf{Enc}(pk, y) \\ c_{x+y} \leftarrow \mathsf{Add}(pk, c_x, c_y) \end{array} \right\} = x + y \right] > 1 - O\left(\lambda^{-c}\right)$$

Table 1. More notation

$[\![x]\!]$	A secret sharing of the value x
$[\![x]\!]_i$	Party i's secret share of the value x
$\langle\!\langle m \rangle\!\rangle_i$	An encryption of the message m under public key of party i

5 Construction and Protocol Definitions

In this section we describe our construction. First, we present a two-party algorithm for creating and shuffling linked lists. Second, we present a technique for converting encryptions (encrypted by one party) into secret shares. Third, we show how to combine these tools into our main construction which is a linear-communication secure merge protocol.

We assume that party i has a key pair (pk_i, sk_i) for an additively homomorphic cryptosystem $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add})$.

5.1 Obliviously Shuffling Input Lists

In this section, we describe our novel two-party protocol for padding and shuffling private linked lists. $\mathsf{ShuffleLL}_i$ (Protocol 2). The goal of the $\mathsf{ShuffleLL}_i$ protocol is for party i to achieve a random permutation of its input list with dummies encrypted under party $(i+1)$'s public key. The protocol takes a parameter, m, defining how many "dummy" elements are created. Although $\mathsf{ShuffleLL}_i$ takes m as a parameter, in our final merge protocol, P_1 should set m equal to the length of its input list. The $\mathsf{ShuffleLL}_i$ protocol realizes the ideal functionality, $\mathcal{F}^i_{\text{shuffle}}$ below.

Ideal Functionality $\mathcal{F}^i_{\text{shuffle}}$

1. *Input:* P_i with sorted list v of size n, and P_{i+1} with permutation $\pi\colon [m+n] \to [m+n]$ for some $m > 0$.
2. Create v' by concatenating m dummy elements to the end of v and shuffle v' using π, $w[j] \leftarrow v'\left[\pi^{-1}(j)\right]$ for $j \in \{0, \ldots, n+m-1\}$.
3. Create linked list t to traverse w, such that if $w[j] = v[k]$, then $w[t[i]] = v[k+1]$.
4. For $j \in \{0, \ldots, n+m-1\}$, *output* $\langle\!\langle w[j] \rangle\!\rangle_{i+1}$, and $\langle\!\langle t[j] \rangle\!\rangle_{i+1}$ to P_i, and $\perp$ to P_{i+1}.
5. *Output* $(\llbracket \pi(n+1) \rrbracket_i, \llbracket \pi(0) \rrbracket_i)$ to P_i, and $(\llbracket \pi(n+1) \rrbracket_{i+1}, \llbracket \pi(0) \rrbracket_{i+1})$ to P_{i+1}.
6. *Output* $\llbracket \pi(n) \rrbracket_i$ to P_i and $\llbracket \pi(n) \rrbracket_{i+1}$ to P_{i+1}.

In the second last step, we output a 2-tuple which are secret shares of the head pointers (positions) of the dummy and real list respectively. In the last step, we output the secret share of the position of a special *end-of-list* dummy element. This special element is used to obliviously switch between the real and dummy list. It is explained in detail in Sect. 5.1 and 5.3.

Below, we describe the shuffle for party P_0 but in the final protocol they also swap positions and rerun. Assume that P_0 holds a sorted list v of length n, and P_1 generates a random permutation π over $[m+n]$. Then, the protocol proceeds as follows,

1. *Encrypt sorted list:* To hide its real elements (input list), P_0 encrypts each element using its public key pk_0 and sends the list of ciphertexts (in sorted order of the underlying value) to P_1.
2. *Generate shares:* Given a value v', party 1 can create an additive sharing of v' as $(v'-r, r)$ for some random value $r \in \mathcal{G}$. In our setting, however, P_1 does not have the plaintext value, v', but instead has an encryption $\langle\!\langle v' \rangle\!\rangle_0$.
Using the additively homomorphism, given a ciphertext $\langle\!\langle v' \rangle\!\rangle_0$, party 1 creates the encrypted pair $(\langle\!\langle v'-r \rangle\!\rangle_0, \langle\!\langle r \rangle\!\rangle_1)$. See Line 2.
3. *Concatenate encrypted dummies:* Party P_1 creates a special dummy known as the *end-of-list* element, and $m-1$ random dummy elements. The *end-of-list* element marks the end of both the real and dummy list but also points to the first element of the dummy list. Therefore, the *end-of-list* element along with the dummy elements form a cycle. The *end-of-list* element stores the largest real value in sorted order instead of a random number as its value. P_1 easily constructs the *end-of-list* element encrypted under pk_0 by

just duplicating $\langle\!\langle v\,[n-1]\rangle\!\rangle_0$. Instead of a linked list terminating by pointing to $\perp$, we will have it point to the this *end-of-list* element. The purpose of the special element becomes apparent when either party's real list is exhausted and we must obliviously *switch* to traversing the dummy list while we access the remaining real elements from the other party (See Sect. 5.3).

4. *Permute ciphertexts and create linked list:* Party P_1 permutes the pair of shares using π by assigning the k^{th} element of the permuted list to the $\pi^{-1}\,(k)^{\text{th}}$ element of the concatenated list as shown in Line 5. To traverse the permuted list in sorted order, P_1 also generates a linked list such that the i^{th} element is the position of the *next* element in sorted order (see Line 6). We also point the *last* dummy element to the *end-of-list* element. Therefore, the real (resp. dummy) list reaches the *end-of-list* element after n (resp. m) steps. See Fig. 1 below which illustrates this construction.
   To hide the linked list from party P_0 (and thus, the underlying permutation), P_1 encrypts each element of the linked list using its public key, pk_1. Finally, it secret shares the position of the first dummy and the first real element as a 2-tuple *head pointer*, *and* the *end-of-list* element.
5. *Recombine shares:* P_1 sends both the shuffled ciphertext pairs and the encrypted linked list to P_0. Party P_0 first decrypts the ciphertexts which were encrypted under its own public key, pk_0 and then *re-encrypts* them using pk_1, P_1's public key. Using the additive property of the encryption scheme, P_0 adds the newly obtained ciphertexts to their corresponding ciphertexts in the pair. Due to the homomorphic property, P_0 obtains an encryption of the sum of the underlying value which is in fact, the original set of real/dummy elements as the pairs were constructed precisely from those values.

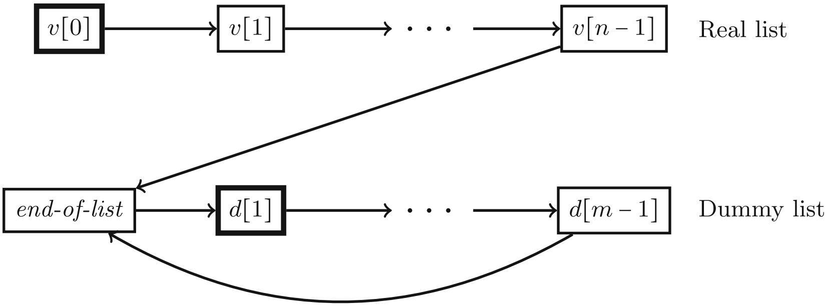

Fig. 1. Construction of the linked list. $d[1]$ and $v[0]$ (as pair of encrypted shares) are the at the head of the dummy and real pointer respectively. Both the last real element, $v[n-1]$ and last dummy element, $d[m-1]$ point to the *end-of-list* element.

Therefore, at the end of Protocol 2, P_0 obtains a permutation (oblivious to itself) of its original list with dummies encrypted under P_1's public key, along with an encrypted linked list to traverse it. Note that the *end-of-list* element is treated as a *dummy* element but stores a *real* value which is crucial in proceeding obliviously when either party exhausts its real list. We further elaborate on this in Sect. 5.3.

We prove $\mathsf{ShuffleLL}_i$ securely computes the ideal functionality $\mathcal{F}^i_{\text{shuffle}}$ in [25].

Protocol 2. $\mathsf{ShuffleLL}_i$: Pad and Permute Linked Lists

Input: Party P_i holds sorted list v of size n; P_{i+1} holds random permutation $\pi\colon [m+n] \rightarrow [m+n]$ for some $m > 0$.

Output: P_i obtains a permutation (under π) of its elements (with m dummies) and linked list, both encrypted using P_{i+1}'s public key.

```
(index $j \in \{0, \ldots, n+m-1\}$)
For $k \in \{0, \ldots, n-1\}$, $P_i$ encrypts $c[k] \leftarrow \langle\langle v[k] \rangle\rangle_i$, and sends $c$ to $P_{i+1}$
For $k \in \{0, \ldots, n-1\}$, $P_{i+1}$ generates random value $r_k \leftarrow \mathcal{G}$, and creates $c_i[k] \leftarrow (c[k] - \langle\langle r_k \rangle\rangle_i, \langle\langle r_k \rangle\rangle_{i+1})$    ▷ 2-tuples of the form $(c_i[k][0], c_i[k][1])$
$P_{i+1}$ generates random $r \leftarrow \mathcal{G}$ and sets $c'_i[0] \leftarrow (c[n-1] - \langle\langle r \rangle\rangle_i, \langle\langle r \rangle\rangle_{i+1})$
    ▷ end-of-list element. $c[n-1] - \langle\langle r \rangle\rangle_i = \langle\langle v[n-1] - r \rangle\rangle_i$
For $k \in \{1, \ldots, m-1\}$, $P_{i+1}$ generates dummies, $d[k] = d_0[k] + d_1[k]$ where $d_0[k], d_1[k] \leftarrow \mathcal{G}$ are random, and creates, $c'_i[k] \leftarrow (\langle\langle d_i[k] \rangle\rangle_i, \langle\langle d_{i+1}[k] \rangle\rangle_{i+1})$
$P_{i+1}$ permutes, $c^\pi_i[j] \leftarrow (c_i \| c'_i)\left[\pi^{-1}(j)\right]$
$P_{i+1}$ creates linked list, $t'[\pi(j)] \leftarrow \pi(j+1)$ with $t'[\pi(n+m-1)] = \pi(n)$    ▷ point the *last* dummy to the *end-of-list* element
$P_{i+1}$ encrypts $t_i[j] \leftarrow \langle\langle t'[j] \rangle\rangle_{i+1}$
$P_{i+1}$ secret shares $\mathfrak{p}_i = (\pi(n+1), \pi(0))$    ▷ head pointers tuple
$P_{i+1}$ secret shares $\mathfrak{e}_i = \pi(n)$    ▷ end-of-list element
$P_{i+1}$ sends $c^\pi_i$, and $t_i$ to $P_i$
$P_i$ recombines $c^\pi[j] \leftarrow c^\pi_i[j][1] + \langle\langle \mathsf{Dec}(sk_0, c^\pi_i[j][0]) \rangle\rangle_{i+1}$
```

5.2 Converting Ciphertexts to Secret Shares

In this section, we give an efficient 2-party protocol for converting ciphertexts from an additively homomorphic cryptosystem into secret shares of the same underlying value. A similar idea was used implicitly for creating "blinded permutations" [29].

In principle, a general-purpose MPC protocol can always be used to convert ciphertexts to secret shares by evaluating the decryption circuit for the encryption scheme within the MPC, but, in general, this is extremely inefficient. $\mathsf{EncToSS}_i$ (Protocol 3) gives an extremely efficient two-party protocol for achieving the same result when the underlying cryptosystem is additively homomorphic. $\mathsf{EncToSS}_i$ realizes the ideal functionality, $\mathcal{F}^i_{\text{decrypt}}$ defined below.

Ideal Functionality $\mathcal{F}^i_{\text{decrypt}}$

1. *Input:* P_i with ciphertext, $\langle\!\langle v \rangle\!\rangle_{i+1}$.
2. *Output* secret shares of value v: $[\![v]\!]_i$ to P_i, and $[\![v]\!]_{i+1}$ to P_{i+1}.

In our setting, party i holds a ciphertext $c = \langle\!\langle v \rangle\!\rangle_{i+1}$ of a private value, v, encrypted under party $(i+1)$'s key. At the end of the protocol, the parties hold additive secret shares of the underlying value v, and neither party learns anything about v.

We prove that $\mathsf{EncToSS}_i$ securely computes $\mathcal{F}^i_{\text{decrypt}}$ in [25].

Protocol 3. $\mathsf{EncToSS}_i$: Convert Ciphertext to Secret Share

Input: Party P_i inputs ciphertext, $c = \langle\!\langle v \rangle\!\rangle_{i+1}$ (encrypted using pk_{i+1}).
Output: Returns secret sharing of the underlying plaintext, v.

```
$P_i$ generates random value, $r_i \leftarrow \mathcal{G}$
$P_i$ encrypts $\langle\!\langle r_i \rangle\!\rangle_{i+1}$
$P_i$ uses the additive homomorphism to compute $\langle\!\langle v + r_i \rangle\!\rangle_{i+1}$
$P_i$ sends $c' = \langle\!\langle v + r_i \rangle\!\rangle_{i+1}$ to $P_{i+1}$
$P_{i+1}$ decrypts $v' \leftarrow \mathsf{Dec}(sk_{i+1}, c')$
$P_{i+1}$ shares $v'$
$P_i$ sets $[\![v'']\!]_i = [\![v']\!]_i - r_i$
return $[\![v'']\!]$
```

5.3 Securely Merging Obliviously Shuffled Lists

We are finally ready to *securely* merge the two parties' lists. Our Merge protocol realizes the ideal functionality, $\mathcal{F}_{\text{merge}}$ defined below.

Ideal Functionality $\mathcal{F}_{\text{merge}}$

1. *Input:* For $i \in \{0, 1\}$, P_i with list v_i of size n_i.
2. $\mathcal{F}_{\text{merge}}$ merges the two lists v_1 and v_2 such that the resultant list, v is sorted.
3. *Output* secret shares of each element of v, $[\![v[j]]\!]_0$ to P_0, and $[\![v[j]]\!]_1$ to P_1, for $j \in \{0, \ldots, n_0 + n_1\}$.

Suppose party P_i holds list v_i of size n_i. The protocol proceeds as described below.

1. *Obliviously shuffle padded list with linked list:* First, both parties call $\mathsf{ShuffleLL}_i$ (for $i \in \{0, 1\}$ (as described in Protocol 2) to obtain an encrypted, permuted version of their input list padded with dummies (including the *end-of-list* element). $\mathsf{ShuffleLL}_i$ also outputs an encrypted linked list that party i

later uses to traverse their list without leaking the accessed positions to party $i+1$ (who knows the permutation).

2. *Access elements from shuffled list:* The parties maintain a secret-shared bit for each party, $[\![b_i]\!]$, and $b_i = 1$ at iterations where P_i needs to access a real element, and $b_i = 0$ at iterations where P_i needs to access a dummy element. In the first step, both parties access their first real element, in all subsequent steps $b_0 \neq b_1$ since only one party advances its real list.[1] The bit, b_i, allows the parties to select and update the appropriate values obliviously using the mux operation (e.g. Protocol 5 line 9).

   At every step in the protocol, the parties also maintain a secret sharing of the last observed real value in P_i's list, cur_i. In any iteration where a dummy element must be consumed from party i's list, we use b_i to obliviously select cur_i over the dummy value, effectively discarding it in place of the actual real value to be compared. See Line 14 of Protocol 5.
3. *Compare real values:* Using b_i, we obtain the real values at the head of each real list. To find the smaller element, we use a generic comparison protocol (Sect. 4.2) which returns a (secret-shared) bit equal to 1 if party 0's real value was smaller than party 1's. Therefore, we set b_0 to the result of the comparison protocol (line 15) and $b_1 \leftarrow 1-b_0$ (line 16) allowing us to appropriately update the head pointer for the next step.
4. *Update head pointer:* Now, we advance one party's real list and the other party's dummy list as follows. First, we find the next position from the encrypted linked list using $\mathsf{EncToSS}_i$. Then, we update the appropriate entry of the head pointer using bit, b_i (line 1). If $b_i = 1$, then this means that P_i's real value was smaller and we must advance the real (resp. dummy) pointer to obtain the next real (resp. dummy) value from P_i's (resp. P_{i+1}'s) list. Protocol 4 details how the head pointer is advanced. We prove in [25] that that every memory location in the shuffled list is accessed exactly once, which makes the overall access pattern independent of the underlying data.
5. *Switching from an exhausted list:* When either party exhausts their real list, we must somehow *notify* the protocol and secret-share the remaining values of the other real list.

   We keep track of when a real list is exhausted by checking when the real pointer reaches the *end-of-list* element. We do so securely using a generic equality testing MPC protocol as described in Sect. 4.2. We maintain another secret-shared bit, fin initialized to 0, which acts like a boolean flag and is inverted as soon as either real pointer reaches its corresponding *end-of-list* element. See line 10 of Protocol 5.

   Without loss of generality, suppose that party 0 exhausted its real list first. This implies that $b_0 = 1$ (and $b_1 = 0$) from the previous iteration, and the real pointer has been advanced to store the position of the *end-of-list* element. Recall that the underlying value of the *end-of-list* element is exactly the same as the largest real value, i.e., the most recent element that party 0 accessed in

[1] Since $b_0 = \neg b_1$ at every iteration after the first, we could increase efficiency by storing only a single bit, but the exposition is simpler if we forego this minor optimization.

the previous iteration. So on Line 14, val_0 will equal the *end-of-list* element i.e., the largest real value of party 0, and val_1 will equal cur_1, the most recent real value from party 1 that has not been advanced and secret-shared yet. Therefore, essentially, we will perform the same comparison as the previous iteration and conclude that val_0 is smaller. However, val_0 is a duplicate of the most recent real value that was secret-shared in the previous iteration. This is where we use the fin bit to "reverse" the bits so that we instead select val_1 as the next real value, and advance the real pointer of party 1 (and dummy pointer of party 0) as required since we're only left with real values from party 1's list. As val_0 is smaller than every remaining real value in party 1's list, every comparison hereafter will always return $b_0 = 1$ which we always *invert* hereafter using fin. We prove fin remains 1 once set in [25], thus proving the correctness of the algorithm. In summary, performing these *dummy* comparisons allows the protocol to remain oblivious by still accessing elements from the permuted list, and using the fin bit allows the protocol to *correctly* compute the merge.

Lastly, notice that if party 1 exhausts it real list first, then by construction, party 0's dummy pointer will reach the *end-of-list* element as we consume one dummy for each real element after the first one and thus, cycle back from the last dummy element to the *end-of-list* element. And since party 1 just exhausted its real list, we know $b_0 = 0$ and $b_1 = 1$. So, pos_0 is equal to the position of the dummy pointer, i.e., the position of the *end-of-list* element. Therefore, in either case (whether party 0 or 1 exhausts a real list), pos_0 will always equal the position of the *end-of-list* element and it is sufficient to only test pos_0 for setting fin (line 10).

Protocol 4. $\mathsf{UpdateHead}_i$: Update Head Pointer to Linked Lists

Input: Bit, $[\![b]\!]$; Head pointer tuple, $[\![\mathfrak{p}]\!]$; linked list, t held by party P_i.

Output: Head pointer tuple updated with the next real or dummy position from t according to bit, b.

```
$[\![pos]\!] \leftarrow \mathsf{mux}\,([\![b]\!], [\![\mathfrak{p}[0]]\!], [\![\mathfrak{p}[1]]\!])$
$\mathsf{Reveal}_i\,(pos)$        ▷ The revealed $pos$ is an index in the shuffled list
$[\![next]\!] \leftarrow \mathsf{EncToSS}_i\,(t[pos])$
$[\![\mathfrak{p}_{new}[1]]\!] \leftarrow \mathsf{mux}\,([\![b]\!], [\![\mathfrak{p}[1]]\!], [\![next]\!])$
$[\![\mathfrak{p}_{new}[0]]\!] \leftarrow \mathsf{mux}\,([\![b]\!], [\![next]\!], [\![\mathfrak{p}[0]]\!])$
return $[\![\mathfrak{p}_{new}]\!]$
```

In the end, both parties obtain element-wise secret shares of the merge of their two sorted lists such that the resulting list is also in sorted order. We prove Merge securely computes $\mathcal{F}_{\text{merge}}$ in [25].

Our algorithm runs in time linear in the length of the two lists requires only linear communication between the two parties assuming the underlying encryption scheme produces ciphertexts with constant factor expansion. The concrete costs are outlined in [25].

Protocol 5. Merge: Securely Merge Sorted Lists

Input: Party P_i holds input list v_i of size n_i.

Output: Parties obtain a secret sharing of the merge of the lists in sorted order.

```
For $i \in \{0,1\}$, $P_i$ locally generates random permutation, $\pi_i\colon [n_0+n_1] \rightarrow [n_0+n_1]$.
For $i \in \{0,1\}$, run $\mathsf{ShuffleLL}_i\,(v_i, \pi_{i+1})$ so that $P_i$ obtains ciphertext list, $c_i$, linked list, $t_i$ and secret shares, $\llbracket \mathfrak{p}_j \rrbracket_i$ and $\llbracket \mathfrak{e}_j \rrbracket_i$ for $j \in (0,1)$.
For $i \in \{0,1\}$, $\llbracket b_i \rrbracket \leftarrow \llbracket 1 \rrbracket$          ▷ $b_i$ indicates real or dummy list
For $i \in \{0,1\}$, $\llbracket cur_i \rrbracket \leftarrow \llbracket \perp \rrbracket$          ▷ $cur_i$ is the current value in the *real list*
$\llbracket end \rrbracket \leftarrow \llbracket \mathfrak{e}_0 \rrbracket$          ▷ position of the *end-of-list* element
$\llbracket fin \rrbracket \leftarrow \llbracket 0 \rrbracket$          ▷ $fin = 1$ if either real list is exhausted
$k \leftarrow 0$
while $k < n_0 + n_1$ do
    For $i \in \{0,1\}$, $\llbracket pos_i \rrbracket \leftarrow \mathsf{mux}\,(\llbracket b_i \rrbracket, \llbracket \mathfrak{p}_i[0] \rrbracket, \llbracket \mathfrak{p}_i[1] \rrbracket)$          ▷ Choose $pos_i$ based on $b_i$
    $\llbracket fin \rrbracket \leftarrow \llbracket fin \rrbracket \oplus \llbracket pos_0 = end \rrbracket$          ▷ If $fin = 1$ it will remain 1
    For $i \in \{0,1\}$, $\mathsf{Reveal}_i\,(\llbracket pos_i \rrbracket)$
    For $i \in \{0,1\}$, $\llbracket \mathfrak{p}_i \rrbracket \leftarrow \mathsf{UpdateHead}_i(\llbracket b_i \rrbracket, \llbracket \mathfrak{p}_i \rrbracket, t_i)$          ▷ Move to new head
    For $i \in \{0,1\}$, $\llbracket temp_i \rrbracket \leftarrow \mathsf{EncToSS}_i\,(c_i\,[pos_i])$          ▷ Access next position
    For $i \in \{0,1\}$, $\llbracket val_i \rrbracket \leftarrow \mathsf{mux}\,(\llbracket b_i \rrbracket, \llbracket cur_i \rrbracket, \llbracket temp_i \rrbracket)$          ▷ Choose real values
    $\llbracket b_0 \rrbracket \leftarrow \llbracket val_0 < val_1 \rrbracket \oplus \llbracket fin \rrbracket$          ▷ Compare real values
    $\llbracket b_1 \rrbracket \leftarrow \llbracket 1 - b_0 \rrbracket$
    $\llbracket l[k] \rrbracket \leftarrow \mathsf{mux}\,(\llbracket b_0 \rrbracket, \llbracket val_1 \rrbracket, \llbracket val_0 \rrbracket)$          ▷ $l[k]$ is the smaller value
    For $i \in \{0,1\}$, $\llbracket cur_i \rrbracket \leftarrow \llbracket val_i \rrbracket$          ▷ Store most recent real value
    $k \leftarrow k + 1$
end while
return $(\llbracket l\,[0] \rrbracket, \ldots, \llbracket l\,[n_0 + n_1 - 1] \rrbracket)$          ▷ secret-sharing of sorted merged list
```

6 Conclusion

In this paper, we presented the first linear-communication 2-party secure merge protocol. The protocol is asymptotically optimal, and efficient enough for practical applications. To achieve this protocol, we introduced a 2-party method to obliviously traverse a permuted list using a novel linked list construction and an extremely efficient technique to convert ciphertexts to secret shares.

Our secure merge protocol makes only black-box use of an additively homomorphic cryptosystem, and a secure computation protocol supporting comparisons, equality tests, and multiplexing on secret shared values.

References

1. Ajtai, M., Komlós, J., Szemerédi, E.: Sorting in $c\log(n)$ steps. Combinatorica **3**, 1–19 (1983)
2. Al-Haj Baddar, S., Batcher, K.: The AKS sorting network. In: Designing Sorting Networks: A New Paradigm, pp. 73–80. Springer, New York (2011). https://doi.org/10.1007/978-1-4614-1851-1_11
3. Aly, A., Keller, M., Rotaru, D., Scholl, P., Smart, N.P., Wood, T.: SCALE-MAMBA (2019). https://homes.esat.kuleuven.be/~nsmart/SCALE/

4. Asharov, G., Lin, W., Shi, E.: Sorting short keys in circuits of size o(n log n). In: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, 10–13 January 2021. pp. 2249–2268. SIAM (2021)
5. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, pp. 307–314. ACM (1968)
6. Bater, J., Elliott, G., Eggen, C., Goel, S., Kho, A., Rogers, J.: SMCQL: secure querying for federated databases. Proc. VLDB Endow. **10**(6), 673–684 (2017)
7. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC, pp. 1–10. ACM, New York (1988)
8. Bourse, F., Del Pino, R., Minelli, M., Wee, H.: FHE circuit privacy almost for free. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9815, pp. 62–89. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53008-5_3
9. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_50
10. Chan, T.-H.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. (eds.) ASIACRYPT 2018. LNCS, vol. 11274, pp. 158–188. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_7
11. Chen, H., Huang, Z., Laine, K., Rindal, P.: Labeled PSI from fully homomorphic encryption with malicious security. In: CCS, pp. 1223–1237. ACM (2018)
12. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: CCS, pp. 1243–1255 (2017)
13. Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Kiribuchi, N., Pinkas, B.: An efficient secure three-party sorting protocol with an honest majority. IACR ePrint 2019/695 (2019)
14. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 3–33. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53887-6_1
15. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10624, pp. 377–408. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_14
16. Chongchitmate, W., Ishai, Y., Lu, S., Ostrovsky, R.: PSI from ring-OLE. In: CCS 2022. ACM (2022)
17. Ciampi, M., Orlandi, C.: Combining private set-intersection with secure two-party computation. In: Catalano, D., De Prisco, R. (eds.) SCN 2018. LNCS, vol. 11035, pp. 464–482. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98113-0_25
18. Couteau, G.: New protocols for secure equality test and comparison. In: Preneel, B., Vercauteren, F. (eds.) ACNS 2018. LNCS, vol. 10892, pp. 303–320. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93387-0_16
19. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 342–362. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30576-7_19

20. Dachman-Soled, D., Malkin, T., Raykova, M., Yung, M.: Efficient robust private set intersection. In: Abdalla, M., Pointcheval, D., Fouque, P.-A., Vergnaud, D. (eds.) ACNS 2009. LNCS, vol. 5536, pp. 125–142. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01957-9_8
21. De Cristofaro, E., Tsudik, G.: Practical private set intersection protocols with linear complexity. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 143–159. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14577-3_13
22. De Cristofaro, E., Tsudik, G.: Experimenting with fast private set intersection. In: Katzenbeisser, S., Weippl, E., Camp, L.J., Volkamer, M., Reiter, M., Zhang, X. (eds.) Trust 2012. LNCS, vol. 7344, pp. 55–73. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30921-2_4
23. Demmler, D., Schneider, T., Zohner, M.: ABY-a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
24. Dong, C., Chen, L., Wen, Z.: When private set intersection meets big data: an efficient and scalable protocol. In: CCS, pp. 789–800 (2013)
25. Falk, B.H., Nema, R., Ostrovsky, R.: A linear-time 2-party secure merge protocol. Cryptology ePrint Archive, Report 2022/380 (2022)
26. Falk, B.H., Ostrovsky, R.: Secure merge with $o(nloglogn)$ secure operations. In: 2nd Conference on Information-Theoretic Cryptography (ITC 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2021)
27. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR ePrint 2012/144 (2012)
28. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_1
29. Gentry, C., Halevi, S., Jutla, C., Raykova, M.: Private database access with HE-over-ORAM architecture. In: Malkin, T., Kolesnikov, V., Lewko, A.B., Polychronakis, M. (eds.) ACNS 2015. LNCS, vol. 9092, pp. 172–191. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28166-7_9
30. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: STOC, pp. 218–229 (1987)
31. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM (JACM) **43**(3), 431–473 (1996)
32. Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: an efficient sorting algorithm for practical secure multi-party computation. IACR ePrint 2014/121 (2014)
33. Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 202–216. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37682-5_15
34. Han, Y., Thorup, M.: Integer sorting in 0(n sqrt (log log n)) expected time and linear space. In: Proceedings of the 43rd Symposium on Foundations of Computer Science, FOCS 2002, pp. 135–144. IEEE Computer Society (2002)
35. Hazay, C., Lindell, Y.: Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. J. Cryptol. **23**(3), 422–456 (2010)
36. Huang, Y., Evans, D., Katz, J.: Private set intersection: are garbled circuits better than custom protocols? In: NDSS (2012)

37. Jarecki, S., Liu, X.: Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 577–594. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00457-5_34
38. Jarecki, S., Liu, X.: Fast secure computation of set intersection. In: Garay, J.A., De Prisco, R. (eds.) SCN 2010. LNCS, vol. 6280, pp. 418–435. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15317-4_26
39. Kiss, Á., Liu, J., Schneider, T., Asokan, N., Pinkas, B.: Private set intersection for unequal set sizes with mobile applications. PoPETs **4**, 97–117 (2017)
40. Kissner, L., Song, D.: Privacy-preserving set operations. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 241–257. Springer, Heidelberg (2005). https://doi.org/10.1007/11535218_15
41. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: CCS, pp. 818–829 (2016)
42. Laud, P., Pankova, A.: Privacy-preserving record linkage in large databases using secure multiparty computation. BMC Med. Genom. **11**(4), 84 (2018)
43. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 262–277. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24861-0_18
44. Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 377–396. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_22
45. Ostrovsky, R.: Efficient computation on oblivious RAMs. In: STOC, pp. 514–523 (1990)
46. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_16
47. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: SpOT-light: lightweight private set intersection from sparse OT extension. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 401–431. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_13
48. Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: private set intersection using permutation-based hashing. In: USENIX Security Symposium, pp. 515–530 (2015)
49. Pinkas, B., Schneider, T., Tkachenko, O., Yanai, A.: Efficient circuit-based PSI with linear communication. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11478, pp. 122–153. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17659-4_5
50. Pinkas, B., Schneider, T., Weinert, C., Wieder, U.: Efficient circuit-based PSI via cuckoo hashing. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10822, pp. 125–157. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78372-7_5
51. Pinkas, B., Schneider, T., Zohner, M.: Faster private set intersection based on OT extension. In: USENIX, pp. 797–812 (2014)
52. Pinkas, B., Schneider, T., Zohner, M.: Scalable private set intersection based on OT extension. IACR Cryptology ePrint Archive (2016)
53. Rindal, P., Rosulek, M.: Improved private set intersection against malicious adversaries. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 235–259. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_9
54. Schoenmakers, B.: MPyC: secure multiparty computation in Python. Github, February 2019

55. Veugen, T., Blom, F., de Hoogh, S.J., Erkin, Z.: Secure comparison protocols in the semi-honest model. IEEE J. Sel. Top. Signal Process. **9**(7), 1217–1228 (2015)
56. Viand, A., Jattke, P., Hithnawi, A.: SoK: fully homomorphic encryption compilers. arXiv preprint arXiv:2101.07078 (2021)
57. Volgushev, N., Schwarzkopf, M., Getchell, B., Varia, M., Lapets, A., Bestavros, A.: Conclave: secure multi-party computation on big data. In: EuroSys, p. 3. ACM (2019)
58. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: efficient multiparty computation toolkit (2016). https://github.com/emp-toolkit/emp-sh2pc
59. Yao, A.: Protocols for secure computations (extended abstract). In: FOCS 1982, pp. 160–164 (1982)
60. Yao, A.: How to generate and exchange secrets. In: FOCS 1986, pp. 162–167 (1986)
61. Zahur, S., Evans, D.: Obliv-C: a language for extensible data-oblivious computation. IACR Cryptology ePrint Archive 2015/1153 (2015)