

On Secure Sorting and Its Application to Privacy-Preserving Graph Pattern Matching

Marcel Keller 

CSIRO Technology, Australia

Abstract. Sorting in secure computation has been studied almost as long as there have been implementations of multi-party computation (Wang et al., AsiaCCS’10). More recent advances include completely secure implementations (revealing nothing other than the list size) that have used Quicksort (Araki et al., CCS’21) and radix sort (Asharov et al., CCS’22). For the first time, we extensively compare the two variants across a range of parameters and for both honest- and dishonest-majority computation. We then apply secure sorting to improve sparse matrix multiplication *without* oblivious RAM based on recent work by Damie et al. (CODASPY’26). Our main contribution here is to produce an output in the same format as the inputs to the multiplication. This is needed for repeated multiplication of graph adjacency matrices, which we use to find cycles in graphs. Such cycles have been associated with money laundering, and this also applies to another graph pattern dubbed scatter-gather, which we can detect with the same approach. Our work therefore can be applied to privacy-preserving anti-money laundering detection.

1 Introduction

Multi-party computation (MPC) is one of only a few privacy-enhancing technologies, the remaining ones being homomorphic encryption, differential privacy, and trusted execution environments. Unlike with the others, the privacy in MPC comes from secret sharing, where any sensitive information is stored across several parties in a way that any unqualified subset cannot reconstruct it. Most secret sharing schemes used in practical contexts are perfectly secure, which means that an unqualified party cannot do better than guessing a secret value.

MPC allows computation on secret-shared values by adding protocols that create new secret-shared values from existing ones without revealing them. These protocols use the mathematics of the secret sharing domain, resulting in functionality that can compute polynomials of values in secret. This has led to the definition of the *Arithmetic Black Box* [DN03], where parties are given access to a black box that can receive, hold, manipulate, and reveal secret values based on the parties’ instructions. The values are addressed by public handles. While this does not rule out the structured storage of secret values in itself, it poses a hindrance to *data-dependent* algorithms and thus many data structures. For example, when arranging data in a tree, the position likely depends on the data. The parties can form a tree of public handles, but for complete secrecy, the handle should not give away any information about the data. Trivially, any algorithm can be converted to a circuit, but this comes at a potentially considerable cost. For example, an element of an array can be accessed by a secret index using a linear scan, which increases the cost by the size of the array.

E-mail: marcel.keller@csiro.au, mks.keller@gmail.com (Marcel Keller)

Ever since implementations of MPC have emerged [MNPS04, DGKN09, PSSW09], a line of work has investigated how algorithms and data structures can be adapted to the limitations explained above without incurring large increases in cost. An early example in this direction is the work by Wang et al. [WLG⁺10]. They proposed two ways of sorting numbers in MPC: using sorting networks for an accurate result or a cheaper, potentially inaccurate randomized algorithm. Sorting networks are fixed networks of compare-and-swap gates. Such gates have two input and two output wires, and they output the input values in an ordered manner, that is, one output gate will always carry the lower value. While there are sorting networks [AKS83] with optimal asymptotic complexity $O(n \log n)$, they are known to underperform less optimal networks [Bat68] with complexity $O(n \log^2 n)$ for practical parameters. Later works suggested using an adapted version of Quicksort [AFO⁺21] or radix sort [BLT14]. Both have better asymptotic complexity in n , but there are trade-offs between them in other parameters. We explore these in a unified implementation, and we find that neither is optimal in all circumstances. A crucial aspect is in particular the reliance of radix sort on oblivious shuffling, whose cost heavily depends on the security model. For two-party computation, we have implemented the solution by Peceny et al. [PRRS25], which relies on earlier work by Chase et al. [CGP20]. The two works realize the shuffling through secure evaluation of a pseudo-random function, for which Peceny et al. use the MPC-friendly one by Alamati et al. [APRR24]. The latter only uses computation modulo two and three, both of which can efficiently be implemented using oblivious transfer extensions. In the context of three-party computation with an honest majority, we use the simple approach by Araki et al. [AFO⁺21], which involves the parties conducting the shuffle pairwise. Peceny et al. propose their own version of radix sort, which we show to be incorrect.

A more general solution for the issue of making data structures and algorithms oblivious and thus compatible with the arithmetic black box comes in the form of oblivious random access memory (ORAM). Originally developed to obscure the access pattern in the client-server model [Gol87], it was later adapted to the MPC context [OS97, GKK⁺12, KS14]. However, ORAM not only increases the cost by a poly-logarithmic factor (in the data structure size), but it also comes with a considerable constant factor. As a result, ORAM is often avoided in favor of a more specialized solution. For example, Damie et al. [DHPR26] have proposed such an MPC scheme for sparse matrix multiplication. A sparse matrix has a large share of zero entries. Therefore, storing the complete matrix is not efficient. Instead, the matrix is represented by a list of tuples for non-zero entries indicating row and column indices as well as the entry. On a normal computer, a dictionary map would be a reasonable data structure to use for finding entries. As outlined above, it is not known how to create an oblivious dictionary without ORAM.

For their matrix multiplication, Damie et al. store one input matrix in row-wise lists of index-value tuples and the other one in column-wise lists. This allows them to efficiently construct the product matrix as a list of tuples with both indices and values. However, they do not provide a method to convert the output back to the input format. This prevents repeated multiplication, which we need for our application. In Section 4, we provide an efficient oblivious algorithm for this. Note that this is not straightforward due to the secrecy requirements. Our approach avoids ORAM while instead deploying the group-wise data structure introduced by Hamada et al. [HIKC23]. Its core idea is to split a list into sub-lists by secret entry markers, that is, a list of the same length whose entries are one whenever a new sub-list starts, and zero otherwise. Together with sophisticated algorithms using secure shuffling and parallel programming techniques, Hamada et al. use this to represent the training set in decision tree training while hiding the number of entries per tree node.

Finally, we consider applying sparse matrix multiplication to graph analysis. It is well known that graphs can be represented as matrices, and graphs with low degree in

sparse matrices. In Section 5, we present a method for using sparse matrix multiplication to detect cycles in graphs. It relies on the correspondence of graph paths and non-zero elements in the matrix representation. While the correspondence is not perfect, we will show that it suffices to find all cycles in a synthetic anti-money laundering dataset.

2 Preliminaries

2.1 Notation

We use $[\cdot]$ to indicate secret-shared values and $\langle \cdot \rangle$ for oblivious shuffles introduced in Section 2.3. The exception is $[n]$ denoting the set $\{0, \dots, n-1\}$. We also use boldface for vectors, both cleartext and secret-shared, and x_i for the i -th element of $\mathbf{x}$, and $M_{i,j}$, $M_{\cdot,j}$, and $M_{i,\cdot}$ for elements, rows, and columns of a matrix M , respectively.

2.2 The Black Box Model of Multi-Party Computation

We use the arithmetic black box model of MPC to demonstrate the capabilities and limitations. This assumes that MPC implements a figurative black box that contains all secret values and that provides input and output facilities as well as basic operations in some domain, most commonly addition and multiplication modulo some integer. There is a rich literature on how to implement further basic operations such as simple (e.g., comparison) and more complex (e.g., exponentiation) non-linear mathematical operations. The feasibility of the latter is given by the existence of bit decomposition operations and the universality of addition and multiplication, going back to Damgård et al. [DFK⁺06]. For efficiency, it is advantageous to combine the arithmetic black-boxes for several domains with conversions between them, most notably one large modulus and one modulo 2 for bit-wise operations [DSZ15, EGK⁺20]. We will refer to the former as arithmetic and the latter as binary. More generally, we refer the reader to introductory textbooks on MPC [CDN15, EKR18] for the basics of MPC and to Keller [Kel20] for practical aspects.

A notable protocol in the black box model is Mux, the result of which is defined as follows:

$$\text{Mux}([c], [x], [y]) = \begin{cases} x & c = 1 \\ y & c = 0. \end{cases}$$

The generic implementation simply uses multiplication:

$$\text{Mux}([c], [x], [y]) \leftarrow [c] \cdot ([x] - [y]) + [y].$$

This can easily be run in the black box due to the availability of arithmetic, and its correctness is trivial. In some protocols, this is the best-known implementation. For two-party computation, however, there is an optimized variant using oblivious transfer due to Rathee et al. [RRK⁺20]. The protocol is needed because the black box model does not allow branching, that is, the parties cannot instruct it to execute an assignment or an operation based on a secret value.

2.3 Data Structure Operations in MPC

The MPC black box is operated by the parties with publicly known handles for the values inside. This allows storing lists of values as well because there is no limit on the number of handles. However, the secrecy requirement limits the use of many algorithms because the requirements of data structure are in conflict with the fact that the handles are publicly known. For example, Quicksort operates by dividing the input list in ever smaller partitions according to their comparison with a fixed element for every partition. The black box

Protocol 1: Secure Permutation (RevealSort [HICT14])

Input $[\mathbf{k}]$ and $[\mathbf{D}]$ of length n such that $\mathbf{k}$ is a permutation of $[n]$

Output $[\mathbf{D}]$ permuted according to the sorting of $\mathbf{k}$

- 1: $\langle\pi\rangle \leftarrow \text{GenShuffle}()$
- 2: $\mathbf{k}' \leftarrow \text{Reveal}(\text{Apply}(\langle\pi\rangle, [\mathbf{k}]))$
- 3: $[\mathbf{D}] \leftarrow \text{Apply}(\langle\pi\rangle, [\mathbf{D}])$
- 4: Permute $[\mathbf{D}]$ according to $\mathbf{k}'$
- 5: **return** $[\mathbf{D}]$

Protocol 2: Secure Inverse Permutation (RevealSortInverse)

Input $[\mathbf{k}]$ and $[\mathbf{D}]$ of length n such that $\mathbf{k}$ is a permutation of $[n]$

Output $[\mathbf{D}]$ permuted according to the inverse of the sorting of $\mathbf{k}$

- 1: $\langle\pi\rangle \leftarrow \text{GenShuffle}()$
- 2: $\mathbf{k}' \leftarrow \text{Reveal}(\text{Apply}(\langle\pi\rangle, [\mathbf{k}]))$
- 3: Permute $[\mathbf{D}]$ according to the inverse of $\mathbf{k}'$
- 4: $[\mathbf{D}] \leftarrow \text{Unapply}(\langle\pi\rangle, [\mathbf{D}])$
- 5: **return** $[\mathbf{D}]$

model requires the partitioning to be public, but simply outputting the result of the required comparisons potentially reveals information about the input values. This can be fixed, as shown in Section 3.1.

Nevertheless, the requirement for public data handles makes it impossible to simply use more data-dependent data structures such as dictionaries. Oblivious RAM has been proven to lift this limitation, both in a dedicated manner [KS14] and in a general manner [Kel19]. However, this comes with a considerable overhead. As a result, there is a line of research investigating a more MPC-specific approach. An early example of this is due to Hamada et al. [HICT14]. They proposed to use oblivious shuffling to permute a list according to a secret-shared permutation. Oblivious shuffling has emerged as a primitive due to its efficient implementation in some contexts. With semi-honest, honest-majority three-party computation, for example, the shuffling is created by switching to two-party secret sharing and running one random shuffling for every subset of size two [AHI⁺22a]. The resulting shuffling is unknown to any party because of one of the three parts. The shuffling can be repeated and reversed, which leads to three relevant algorithms: **GenShuffle**, **Apply**, and **Unapply**. We will use $\langle\pi\rangle$ for a permutation generated in such a way. Protocol 1 shows Hamada et al.'s permutation. The correctness follows from the fact that, if ρ is the permutation represented by $\mathbf{k}$, the output is $\mathbf{D}$ permuted by $(\rho \circ \pi^{-1}) \circ \pi = \rho$. By representation, we mean that a vector containing $(0, \dots, n-1)$ such that using it as a sort key executes the permutation. Meanwhile, the security follows from the fact that the only information revealed is $\pi \circ \rho$ where π is a freshly generated random permutation.

Using the same techniques, it is straightforward to construct the inverse version shown in Protocol 2. Whereas **RevealSort** sorts $\mathbf{D}$ according to the sorting order of $\mathbf{k}$, **RevealSortInverse** uses the explicit permutation of $\mathbf{k}$. Consider the following example to understand the difference. For inputs $\mathbf{k} = (1, 2, 0)$ and $\mathbf{D} = (10, 20, 30)$, **RevealSort** outputs $\mathbf{D} = (30, 10, 20)$, but **RevealSortInverse** outputs $\mathbf{D} = (20, 30, 10)$. The correctness can be deduced similarly to the above.

2.3.1 Implementing Oblivious Shuffling

Keller and Scholl [KS14] proposed a generic approach to oblivious shuffling based on Waksman networks after earlier works in two-party computation [HEK12] and homomorphic encryption [GHS12]. Waksman networks [Wak68] are switching circuits, that is, they consist of wires and gates with two input and output wires that switch the two values depending on a configuration bit. This makes them immediately usable in the context of secure computation. However, they have complexity $O(n \log n)$, and the generic approach requires one such network per party. As a result, there have been efforts to achieve linear complexity, for example by Araki et al. [AFO⁺21] for the honest-majority case with three parties or Chase et al. [CGP20] for two parties. The latter has been refined by Peceny et al. [PRRS25] based on the MPC-friendly PRF by Alamiati et al. [APRR24], which we use for two-party computation in this work. The PRF is based on evaluating linear functions in $\mathbb{F}_3$ and $\mathbb{F}_2$. There is no communication cost for linear functions when the values are secret-shared in the right domain. As a result, communication is mainly required for the conversion from $\mathbb{F}_3$ to $\mathbb{F}_2$, for which Alamiati et al. suggested an efficient protocol based on oblivious transfer. Its core idea is to combine the facts that OT allows oblivious selection on the receiver's input, and that shares in $\mathbb{F}_3$ consist of only two bits.

2.3.2 Oblivious List Partition

A more sophisticated application of oblivious shuffling in multi-party computation has been demonstrated in the decision tree learning algorithm by Hamada et al. [HIKC23]. They have proposed a set of oblivious algorithms that operate on a list that is secretly divided. In their case, the secret division refers to splitting training samples across nodes. The division is stored as a secret-shared list of zeros and ones such that every one marks the start of a new division. One of their algorithms, **GroupSum**, computes for example the sum of all entries in a division and produces a list where the result is stored in all positions corresponding to a division. For example, the group-wise sum of

$$(1, 2, 3, 4, 5, 6),$$

where the division is $(3, 1, 2)$ represented as

$$(1, 0, 0, 1, 1, 0),$$

results in

$$(6, 6, 6, 4, 11, 11).$$

Algorithm 3 shows a trivial algorithm using only limited branching as facilitated by oblivious selection. Its downside is that it takes $O(n)$ communication rounds when executed with multi-party computation because oblivious selection generally incurs a communication round. This can be reduced to $O(\log n)$ using techniques from parallel programming [LF80].

Hamada et al. proposed to use oblivious shuffling instead, which results in a protocol that only uses a constant number of rounds in addition to a constant number of invocations of oblivious shuffling. In some security models, such as honest-majority semi-honest three-party computation, oblivious shuffling has linear cost with a constant number of rounds [AHI⁺22a], leading to an overall constant-round protocol. Protocol 4 shows Hamada et al.'s algorithm (called **GroupSum2** therein).

The algorithm uses several algorithms related to prefix summation. **PrefixSum** denotes the plain prefix sum, that is, $\{\sum_{j=1}^i x_j\}_{i=1,\dots,n}$, and **PrefixSumR** denotes the prefix sum starting from the end, that is, $\{\sum_{j=i}^n x_j\}_{i=1,\dots,n}$, both for input $[\mathbf{x}]$. PrefixSum^{-1} and PrefixSumR^{-1} denote the inverse functions. PrefixSum^{-1} can be computed as $\{x_i - x_{i-1}\}_{i=1,\dots,n}$ with $x_0 = 0$, and PrefixSumR^{-1} can be computed as $\{x_i - x_{i+1}\}_{i=1,\dots,n}$ with $x_{n+1} = 0$.

Algorithm 3: Trivial group-wise sum**Inputs** $\mathbf{x}$, group indicator $\mathbf{g}$, both of length n **Output** Group-wise sum of $\mathbf{x}$ according to $\mathbf{g}$

```

for  $i = 0, \dots, n - 1$  do
  if  $g_i = 1$  or  $i = 0$  then
     $y_i \leftarrow x_i$ 
  else
     $y_i \leftarrow y_{i-1} + x_i$ 
  end if
end for
for  $i = n - 1, \dots, 0$  do
  if  $i = n$  or  $g_{i+1} = 1$  then
     $y_i \leftarrow y_i$ 
  else
     $y_i \leftarrow y_{i+1}$ 
  end if
end for
return  $\mathbf{y}$ 

```

Protocol 4: Group-wise sum based on oblivious shuffling (GroupSum)**Inputs** $[\mathbf{x}]$, group indicator $[\mathbf{g}]$, both of length n **Output** Group-wise sum of $[\mathbf{x}]$ according to $[\mathbf{g}]$

```

 $[\mathbf{p}] \leftarrow \text{PrefixSumR}([\mathbf{x}]) * [\mathbf{g}]$ 
 $[\mathbf{k}] \leftarrow \text{GenBitPerm}(\neg[\mathbf{g}])$ 
 $[\mathbf{p}]_1 \leftarrow \text{RevealSort}([\mathbf{k}], [\mathbf{p}]_1)$ 
 $[\mathbf{s}]_1 \leftarrow \text{PrefixSumR}^{-1}([\mathbf{p}]_1)$ 
 $[\mathbf{d}] \leftarrow \text{RevealSortInverse}([\mathbf{k}], \mathbf{d}_1) * \mathbf{g}$ 
return  $\text{PrefixSum}([\mathbf{d}])$ 

```

$\triangleright *$ denotes the Schur product

Meanwhile, all of the communication cost falls within the permutation-related parts, `GenBitPerm`, `RevealSort`, and `RevealSortInverse`. The first creates a secret permutation from a bit vector such that the permutation stably sorts the bit vector according to value, and the other two compute this permutation or its inverse on a vector. Protocol 5 has the details of `GenBitPerm`, an optimization of `DestComp` [HICT14], and `RevealSort` and `RevealSortInverse` are explained in Protocols 1 and 2, respectively.

3 Secure Sorting Algorithms

In this section, we will introduce the two sorting algorithms used in this work and consider aspects relevant for their usage in secure computation, namely the choice of secret sharing domain and asymptotic complexity.

Protocol 5: Secret permutation from bits (GenBitPerm [CHI⁺19])

Input $[b]$ of length n with elements in $\{0, 1\}$

Output $[k]$ stably sorting $[b]$

```

1:  $[f_0] \leftarrow 1 - [b]$ 
2:  $[f_1] \leftarrow [b]$ 
3:  $[s_{0,i}] \leftarrow \sum_{k=0}^i [f_{0,j}]$  for  $i = 0, \dots, n-1$ 
4:  $[s_{1,i}] \leftarrow \sum_{k=0}^i [f_{1,j}] + s_{0,n-1}$  for  $i = 0, \dots, n-1$ 
5:  $[k_i] \leftarrow \text{Mux}(b_i, s_{1,i}, s_{0,i})$  for  $i = 0, \dots, n-1$ 
6: return  $[k]$ 

```

3.1 Quicksort

Quicksort [Hoa61] is an established sorting algorithm that achieves expected optimal time when used with randomness. The core is to split the numbers to be sorted in ever smaller partitions based on comparison with an element per partition, the so-called pivot. Randomness ensures that this pivot splits every partition evenly on average, thus achieving one element per partition as quickly as possible. In the context of secure computation, random shuffling has to be used for privacy together with padding to make the numbers unique. This is because the comparison results have to be revealed to allow for the partitioning to happen in cleartext. Oblivious computation of this would be prohibitively expensive. Algorithm 6 and Protocol 7 show our implementation in MP-SPDZ, with an emphasis on the vector notation that is amenable to the virtual machine design. This presentation deviates from prior presentations in that it avoids data structures like sets. Correctness and security have been established by prior works [HKI⁺13, AFO⁺21]. The security primarily relies on the security of the arithmetic black box and the fact that the comparison results are independent of the data due to the shuffling and uniqueness of the elements.

Complexity The most expensive aspect of Protocol 7 is the main loop starting at line 9. Baum et al. [BBM⁺25] have established that the number of expected loop runs is in $O(\log n)$. Furthermore, every loop run involves n secure comparisons with bit length $m + \lceil \log_2(n) \rceil$ as the dominant factor, resulting in an overall complexity of $O(n \log n (m + \log n)) + S(n)$, where $S(n)$ denotes the cost of shuffling n elements. Algorithm 6 does not contribute to the cost in a relevant way for two reasons. First, it does not involve communication, and secondly, its complexity is in $O(n)$, assuming that all integers fit into registers. This is a reasonable assumption given that most computers today have 64-bit registers.

Choice of secret sharing domain Quicksort is not very sensitive to the secret sharing domain because many protocols offer efficient comparison even in the arithmetic domain by temporarily switching to the binary domain, for example using Rep3 [Kel25] for three-party computation or ABY for two-party computation [DSZ15].

3.2 Radix Sort

Unlike most sorting algorithms, radix sort does not use comparison. Instead, it requires that the elements can be split into digits (for example, bit decomposition for integers), which are then used to sort digit by digit. The core operations thus are permutations. Bogdanov et al. [BLT14] have created a blueprint for using radix sort in secure computation,

Algorithm 6: Quicksort Permutation QsPerm**Input** n pivot indicators $\mathbf{g}$ and comparison results $\mathbf{c}$ **Output** Updated pivot indicators $\mathbf{h}$ and permutation $\mathbf{k}$ to sort elements into continuous partitions

```

1: for  $i = 0, \dots, n - 1$  do
2:    $h_i \leftarrow g_i$ 
3:    $k_i \leftarrow i$ 
4: end for
5:  $a, b \leftarrow 0$ 
6: for  $i = 0, \dots, n - 1$  do
7:   if  $g_i = 1$  then                                     ▷ New existing partition
8:     if  $i \neq 0$  then                                       ▷ Wrap up previous partition
9:        $k_a, k_b \leftarrow k_b, k_a$ 
10:      if  $a = i - 1$  then
11:         $h_{i-1} = 1$ 
12:      else
13:         $h_{a+1} = 1$ 
14:      end if
15:    end if
16:     $a, b \leftarrow i$ 
17:  else
18:    if  $c_i = 0$  then
19:       $a \leftarrow a + 1$ 
20:       $k_a, k_b \leftarrow k_a, k_b$ 
21:    end if
22:  end if
23: end for
24:  $k_a, k_b \leftarrow k_b, k_a$ 
25: if  $c = n - 1$  then
26:    $h_{n-1} \leftarrow 1$ 
27: else
28:    $h_{c+1} \leftarrow 1$ 
29: end if

```

which has been improved by further works [CHI⁺19, AHI⁺22a]. At its core, the approach uses two components, an algorithm to create a secret permutation sorting a list of digits in a stable manner and the using this secret permutation. Bogdonav et al. and Chida et al. [CHI⁺19] have used bits as digits, but Asharov et al. [AHI⁺22b] have extended this to multi-bit digits. Protocol 8 shows this extension, and Protocol 9 shows the overall algorithm. Protocol 8 uses Demux as defined by Launchbury et al. [LADM14]. It converts integers into unary representations, that is, the ℓ -bit number x is converted into a vector containing 1 in the x -th position and 0 otherwise. Correctness and security are covered by the aforementioned works.

Complexity GenMultiBitPerm involves $O(n2^\ell)$ multiplications, both when computing $[\mathbf{f}_i]$ and $[k_i]$. Therefore, the cost of radix sort comes down to $O(m/\ell \cdot (n2^\ell + S(n)))$, where $S(n)$ denotes the cost of generating and executing a random shuffle of size n . This presents a trade-off regarding the choice of ℓ depending on $S(n)$. Asharov et al. have not suggested a choice. A further trade-off comes in comparison with Quicksort, whose complexity features

Protocol 7: Secure Quicksort

Input Secret shared vector $[x]$ of length n , bit length m

Output Sorted secret shared vector $[y]$, secret sorting permutation $[\pi]$

```

1: for  $i = 0, \dots, n - 1$  do
2:    $x_i \leftarrow x_i \cdot 2^{\lceil \log_2(n) \rceil} + i$  ▷ Unique numbers maintaining order
3: end for
4:  $[x], \langle \rho \rangle \leftarrow \text{Apply}(\text{GenShuffle}(), [x])$  ▷ Shuffle for public comparison results and random pivots
5: for  $i = 1, \dots, n - 1$  do
6:    $g_i \leftarrow 1$ 
7: end for
8:  $g_0 \leftarrow 1$  ▷ Initialize pivot indicators
9: while  $\sum_i g_i < n$  do
10:  for  $i = 0, \dots, n_1$  do
11:   if  $g_i = 1$  then ▷ True for  $i = 0$  by design
12:     $[p_i] \leftarrow [x_i]$ 
13:   else
14:     $[p_i] \leftarrow [p_{i-1}]$ 
15:   end if
16:  end for
17:  for  $i = 0, \dots, n - 1$  do
18:    $c_i \leftarrow \text{Reveal}([p_i] < [x_i])$  ▷  $(m + \lceil \log_2(n) \rceil)$ -bit comparison
19:  end for
20:   $\mathbf{g}, \mathbf{k} \leftarrow \text{QsPerm}(\mathbf{g}, \mathbf{c})$ 
21:  Permute  $[x]$  according to  $\mathbf{k}$ 
22: end while
23: for  $i = 0, \dots, n - 1$  do
24:   $[y_i] \leftarrow \lfloor [x_i] / 2^{\lceil \log_2(n) \rceil} \rfloor$ 
25:   $[p'_i] \leftarrow [x_i] \bmod 2^{\lceil \log_2(n) \rceil}$ 
26: end for

```

a factor of $n \log^2 n$ but no dependency on $mS(n)$. We will later show that there is no universal optimum. Instead, the optimal choice depends on both m and n as well as the protocol and thus the security model.

Choice of secret sharing domain When using the binary domain shuffling approach by Peceny et al. [PRRS25], it is natural to use the binary domain for Protocol 9. However, this domain does not lend itself to summation required by **GenMultiBitPerm**. We therefore convert to the arithmetic domain for the dot product before converting back. Furthermore, note that the necessary multiplications can be done using the **Mux** protocol because $f_{i,j}$ are bits by definition. The **Demux** protocol clearly lends itself to the binary domain. As a result, we convert $[f_{i,j}]$ from binary to arithmetic and $[\mathbf{k}]$ back to binary.

On the other hand, the three-party protocols we use offer shuffling for the arithmetic domain, and they also offer an efficient dot product. It is therefore beneficial to use the arithmetic domain throughout.

Protocol 8: Permutation generation from multiple bits (GenMultiBitPerm)**Input** $[x]$ of length n with ℓ -bit elements**Output** Permutation $[k]$ sorting x in a stable manner

- 1: $[f_i] \leftarrow \text{Demux}(x_i, \ell)$ for all $i \in [n]$
- 2: $[k_i] \leftarrow \text{DotProduct}(\{[f_{i,j}]\}_{j \in [2^\ell]}, \{\sum_{k \leq i \vee l < j} [f_{k,l}]\}_{j \in [2^\ell]}) - 1$ for $i \in [n]$
- 3: **return** $[k]$

Protocol 9: Radix sort**Input** $[x]$ of length n with m -bit elements, digit length ℓ **Output** Permutation $[k]$ sorting x in a stable manner

- 1: $[k_i] \leftarrow i$ for all $i \in [n]$
- 2: **for** $j = 0, \dots, \lceil m/\ell \rceil$ **do**
- 3: Let d_i be the j -th least significant digit of x_i for all $i \in [n]$.
- 4: **if** $j > 0$ **then**
- 5: $[d], \langle \rho \rangle \leftarrow \text{RevealSort}([k], [d])$
- 6: **end if**
- 7: $[k] \leftarrow \text{GenMultiBitPerm}([d])$
- 8: **if** $j > 0$ **then**
- 9: Permute k by $\langle \rho^{-1} \rangle$.
- 10: **end if**
- 11: **end for**
- 12: **return** $[k]$

3.2.1 Issues with Sorting by Peceny et al. [PRRS25]

Peceny et al. present a radix-like sorting algorithm using their random shuffling protocol. In this section, we present two correctness issues with their algorithm.

Partition-Add is the name given by Peceny et al. to an algorithm that is supposed to achieve the same as GenMultiBitSort by Asharov et al. [AHI⁺22b], namely taking a list of ℓ -bit values and outputting a permutation that will sort the list in a stable manner. However, it avoids the most expensive part, namely the multiplication. Algorithm 10 has the details.

It is easy to see that the algorithm does not achieve its promise. On input $X = (1, 0)$, for example, it outputs $\pi = (2, 3)$ rather than a permutation indicating a switch between the two input values as required. More concretely, input $X = (1, 0)$ and $\ell = 1$ implies that

$$\begin{aligned}
 (f_{1,1}, f_{1,2}) &= (0, 1) & (f_{1,2}, f_{2,2}) &= (1, 0) \\
 (s_{1,1}, s_{1,2}) &= (0, 1) & (s_{1,2}, s_{2,2}) &= (2, 2) \\
 \pi_1 = s_{1,1} + s_{2,1} &= 2 & \pi_2 = s_{2,1} + s_{2,2} &= 3.
 \end{aligned}$$

Algorithm 10: Partition-Add**Input** $\{X_i\}_{i \in [n]}$ where X_i are ℓ -bit values**Claimed output** Permutation π such that π sorts $\{X_i\}_{i \in [n]}$

- 1: For each $i \in [n]$, set $(f_{i,1}, \dots, f_{2^\ell,i})$ to the one-hot expansion of X_i .
- 2: For each $j \in [2^\ell], i \in [n] : s_{j,i} \leftarrow \sum_{i',j': j' < j \vee (j'=j \wedge i' \leq i)} f_{i',j'}$
- 3: For each $i \in [n] : \pi_i \leftarrow \sum_{j \in [2^\ell]} s_{j,i}$
- 4: **return** π

Algorithm 11: Incorrect radix sort**Input** $\{X_i\}_{i \in [n]}$ where X_i are ℓ -bit values, block size t **Claimed output** Permutation π such that π sorts $\{X_i\}_{i \in [n]}$

- 1: Let X'_i be the most significant t bits of X_i for $i \in [n]$
- 2: $\pi \leftarrow \text{Partition-Add}(\{X'_i\}_{i \in [n]})$
- 3: **for** $j \in [1, \lceil \ell/t \rceil]$ **do**
- 4: Let X'_i be the most significant unused t bits of X_i for $i \in [n]$
- 5: $\rho \leftarrow \text{Partition-Add}(\{X'_i\}_{i \in [n]})$
- 6: $\pi \leftarrow \pi \circ \rho$
- 7: **end for**
- 8: **return** π

More generally, π is monotonous by definition. For $k < k'$,

$$\begin{aligned}
\pi_{k'} - \pi_k &= \sum_{j \in [2^\ell]} s_{j,k'} - \sum_{j \in [2^\ell]} s_{j,k} \\
&= \sum_{j \in [2^\ell]} s_{j,k'} - s_{j,k} \\
&= \sum_{j \in [2^\ell]} \left(\sum_{i',j': j' < j \vee (j'=j \wedge i' \leq k')} f_{i',j'} - \sum_{i',j': j' < j \vee (j'=j \wedge i' \leq k)} f_{i',j'} \right) \\
&= \sum_{j \in [2^\ell]} \sum_{j'=k}^{k'} f_{i',j} \geq 0,
\end{aligned}$$

where the last equality follows from $f_{i,j} \in \{0,1\}$ by definition for all i and j . As a result, π can never represent a permutation that changes any positions.

Peceny et al. also present a high-level radix sort algorithm that is easily proven to be incorrect as well. Algorithm 11 details a simplified version that avoids conversion between secret domains.

We now present a simple counterexample, assuming a correct implementation of Partition-Add. Let $t = 1$ and $X = (1, 2)$. It follows that $\ell = 2$ suffices. The most significant bits of (X_1, X_2) are $(0, 1)$. Assuming a correct Partition-Add implementation, this results π being the identity permutation. The remaining bits of (X_1, X_2) are $(1, 0)$, resulting in ρ being the permutation that swaps the two values. Clearly, $\pi \circ \rho$ is also the “swap” permutation. However, $(1, 2)$ is already sorted correctly, and thus, the returned permutation should be the identity permutation. The more established pattern for secure

radix sort, used by Asharov et al. [AHI⁺22b], differs in two ways: First, it processes the bits from least to most significant, so that the most significant bits have more influence over the result. Second, the partial permutation (here the result of Partition-Add) is also used to sort the next chunk of bits before generating a new partial permutation. Note that even changing the order of processing cannot fix the example above because even swapping π and ρ gives the same wrong result. We therefore conclude that there is no straightforward fix to the radix sort by Peceny et al.

3.3 Benchmarks

We have implemented our protocols using MP-SPDZ [Kel20], and we are committed to providing the code as open source. MP-SPDZ already contained radix sort with $\ell = 1$ and some general and efficient oblivious shuffling, but we have added over 5000 lines of code, mostly for two-party computation. The most notable addition is the oblivious shuffling by Peceny et al. [PRRS25] based on the OT extension facility in MP-SPDZ. While Peceny et al. provide their own implementation [Rin26b], we have found that it is based on an OT extension with unknown security [Rin26a].

Tables 1–6 show that the optimal choice of the sorting protocol depends on the setting. This is most likely due to the competing cost structures of the variants.

Table 1: Time in seconds to sort n m -bit numbers with honest-majority semi-honest three-party computation.

m	Variant	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	Radix, $\ell = 1$	0.04	0.04	0.08	0.28	2.95
	Quicksort	0.11	0.14	0.29	2.73	33.76
2	Radix, $\ell = 1$	0.05	0.05	0.11	0.49	5.42
	Radix, $\ell = 2$	0.04	0.05	0.11	0.41	4.42
	Quicksort	0.11	0.14	0.29	2.73	34.14
	Radix, $\ell = 1$	0.06	0.07	0.17	0.90	10.29
4	Radix, $\ell = 2$	0.05	0.06	0.17	0.72	7.97
	Radix, $\ell = 3$	0.05	0.07	0.27	1.32	14.04
	Radix, $\ell = 4$	0.04	0.07	0.29	1.20	11.90
	Quicksort	0.10	0.14	0.30	2.81	35.25
	Radix, $\ell = 1$	0.09	0.11	0.30	1.68	19.75
8	Radix, $\ell = 2$	0.06	0.09	0.27	1.33	14.98
	Radix, $\ell = 3$	0.06	0.09	0.37	1.89	20.46
	Radix, $\ell = 4$	0.05	0.09	0.44	2.08	21.65
	Radix, $\ell = 5$	0.06	0.11	0.79	3.60	36.64
	Radix, $\ell = 6$	0.07	0.17	1.28	5.94	59.72
	Quicksort	0.09	0.14	0.31	2.96	36.72
	Radix, $\ell = 1$	0.16	0.19	0.54	3.54	38.77
16	Radix, $\ell = 2$	0.10	0.14	0.49	2.54	28.95
	Radix, $\ell = 3$	0.09	0.14	0.66	3.60	39.34
	Radix, $\ell = 4$	0.08	0.13	0.76	3.97	41.58
	Radix, $\ell = 5$	0.09	0.19	1.36	6.60	70.45
	Radix, $\ell = 6$	0.08	0.22	1.84	8.55	86.95
	Quicksort	0.09	0.13	0.28	3.20	38.85
	Radix, $\ell = 1$	0.28	0.35	1.03	6.40	76.91
32	Radix, $\ell = 2$	0.17	0.24	0.90	4.93	57.09
	Radix, $\ell = 3$	0.15	0.23	1.12	6.37	70.69
	Radix, $\ell = 4$	0.12	0.23	1.41	7.65	81.73
	Radix, $\ell = 5$	0.12	0.29	2.21	11.56	119.53
	Radix, $\ell = 6$	0.13	0.38	3.38	16.57	168.40
	Quicksort	0.09	0.13	0.29	3.03	35.61

Table 2: Communication in MB to sort n 32-bit numbers with honest-majority semi-honest three-party computation.

m	Variant	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	Radix, $\ell = 1$	0	2	19	192	1,920
	Quicksort	1	10	145	1,944	26,430
2	Radix, $\ell = 1$	0	4	43	433	4,328
	Radix, $\ell = 2$	0	3	34	338	3,375
	Quicksort	1	11	152	2,033	27,466
4	Radix, $\ell = 1$	1	9	92	915	9,155
	Radix, $\ell = 2$	1	7	73	725	7,250
	Radix, $\ell = 3$	1	12	121	1,209	12,088
	Radix, $\ell = 4$	1	9	92	919	9,188
	Quicksort	1	12	167	2,210	29,536
	Radix, $\ell = 1$	2	19	188	1,880	18,801
8	Radix, $\ell = 2$	2	15	150	1,499	14,991
	Radix, $\ell = 3$	2	18	184	1,840	18,400
	Radix, $\ell = 4$	2	19	189	1,887	18,868
	Radix, $\ell = 5$	3	31	314	3,142	31,420
	Radix, $\ell = 6$	5	52	516	5,163	51,626
	Quicksort	1	14	197	2,565	33,677
	Radix, $\ell = 1$	4	38	382	3,815	38,150
16	Radix, $\ell = 2$	3	31	305	3,053	30,530
	Radix, $\ell = 3$	4	37	373	3,730	37,300
	Radix, $\ell = 4$	4	38	383	3,829	38,286
	Radix, $\ell = 5$	6	63	634	6,337	63,367
	Radix, $\ell = 6$	8	78	777	7,770	77,698
	Quicksort	1	16	217	3,252	41,699
	Radix, $\ell = 1$	8	77	768	7,680	76,800
32	Radix, $\ell = 2$	6	62	616	6,156	61,560
	Radix, $\ell = 3$	7	69	688	6,881	68,813
	Radix, $\ell = 4$	8	77	771	7,707	77,070
	Radix, $\ell = 5$	11	111	1,113	11,127	111,271
	Radix, $\ell = 6$	16	156	1,559	15,592	155,915
	Quicksort	2	23	302	3,995	47,730

Table 3: Time in seconds to sort n m -bit numbers with honest-majority malicious three-party computation. $\perp$ indicates memory exhaustion.

m	Variant	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
2	Radix, $\ell = 1$	0.46	0.49	1.20	5.03	51.67
	Radix, $\ell = 2$	0.42	0.49	1.18	3.35	31.53
	Quicksort	0.51	0.59	1.50	13.48	141.80
4	Radix, $\ell = 1$	0.47	0.66	1.87	9.75	107.08
	Radix, $\ell = 2$	0.44	0.62	1.65	6.42	66.08
	Radix, $\ell = 3$	0.46	0.69	2.73	10.78	$\perp$
	Radix, $\ell = 4$	0.44	0.59	2.68	9.35	$\perp$
	Quicksort	0.51	0.62	1.57	14.55	148.34
8	Radix, $\ell = 1$	0.52	0.77	3.43	19.40	217.07
	Radix, $\ell = 2$	0.48	0.76	3.08	12.68	134.51
	Radix, $\ell = 3$	0.43	0.73	3.75	15.24	$\perp$
	Radix, $\ell = 4$	0.46	0.75	4.23	16.35	$\perp$
	Radix, $\ell = 5$	0.42	1.07	6.79	25.23	$\perp$
	Radix, $\ell = 6$	0.51	1.46	11.26	40.93	$\perp$
	Quicksort	0.46	0.65	1.65	15.33	163.18
16	Radix, $\ell = 1$	0.57	1.21	6.24	39.03	442.73
	Radix, $\ell = 2$	0.52	1.04	5.81	25.35	274.70
	Radix, $\ell = 3$	0.52	1.02	6.70	29.75	$\perp$
	Radix, $\ell = 4$	0.47	1.15	7.55	30.31	$\perp$
	Radix, $\ell = 5$	0.49	1.67	12.18	47.41	$\perp$
	Radix, $\ell = 6$	0.51	1.75	15.28	59.16	$\perp$
	Quicksort	0.46	0.70	1.83	17.43	190.29
32	Radix, $\ell = 1$	0.81	2.14	12.21	77.27	888.92
	Radix, $\ell = 2$	0.69	1.67	11.20	50.11	550.63
	Radix, $\ell = 3$	0.63	1.73	12.25	53.02	$\perp$
	Radix, $\ell = 4$	0.64	1.88	14.19	58.53	$\perp$
	Radix, $\ell = 5$	0.68	2.27	20.03	81.57	$\perp$
	Radix, $\ell = 6$	0.82	2.91	27.60	110.47	$\perp$
	Quicksort	0.46	0.59	1.40	10.51	117.00

Table 4: Communication in MB to sort n m -bit numbers with honest-majority malicious three-party computation. $\perp$ indicates memory exhaustion.

m	Variant	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
2	Radix, $\ell = 1$	17	32	190	1,773	17,569
	Radix, $\ell = 2$	17	30	166	1,531	15,096
	Quicksort	20	163	2,154	29,080	319,105
4	Radix, $\ell = 1$	19	51	382	3,720	36,917
	Radix, $\ell = 2$	19	47	336	3,231	31,983
	Radix, $\ell = 3$	20	67	539	5,282	$\perp$
	Radix, $\ell = 4$	20	54	406	3,941	$\perp$
	Quicksort	20	177	2,379	31,738	343,956
8	Radix, $\ell = 1$	22	90	766	7,604	75,394
	Radix, $\ell = 2$	22	81	674	6,618	65,526
	Radix, $\ell = 3$	23	95	828	8,043	$\perp$
	Radix, $\ell = 4$	23	95	814	8,029	$\perp$
	Radix, $\ell = 5$	28	141	1,285	12,622	$\perp$
	Radix, $\ell = 6$	35	210	1,986	19,663	$\perp$
	Quicksort	20	217	2,831	37,064	393,638
16	Radix, $\ell = 1$	30	167	1,556	15,463	153,555
	Radix, $\ell = 2$	28	148	1,379	13,476	133,819
	Radix, $\ell = 3$	31	176	1,652	16,253	$\perp$
	Radix, $\ell = 4$	31	176	1,658	16,286	$\perp$
	Radix, $\ell = 5$	40	269	2,588	25,439	$\perp$
	Radix, $\ell = 6$	45	309	2,988	29,612	$\perp$
	Quicksort	21	230	3,271	47,386	489,928
32	Radix, $\ell = 1$	46	322	3,121	31,027	307,469
	Radix, $\ell = 2$	43	285	2,769	27,046	267,996
	Radix, $\ell = 3$	46	314	3,067	29,915	$\perp$
	Radix, $\ell = 4$	49	341	3,328	32,669	$\perp$
	Radix, $\ell = 5$	61	461	4,531	44,588	$\perp$
	Radix, $\ell = 6$	75	607	6,002	59,349	$\perp$
	Quicksort	20	163	1,303	16,888	221,055

Table 5: Time in seconds to sort n m -bit numbers with semi-honest two-party computation using oblivious transfer (SoftSpokenOT). $\perp$ indicates omitted runs to save resources.

m	Variant	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	Radix, $\ell = 1$	0.15	0.29	1.89	15.87	160.18
	Quicksort	0.68	2.67	6.08	41.48	639.70
2	Radix, $\ell = 1$	0.22	0.57	3.99	34.87	236.82
	Radix, $\ell = 2$	0.20	0.53	3.75	33.94	200.53
	Quicksort	0.72	2.80	6.12	42.26	649.77
4	Radix, $\ell = 1$	0.21	0.60	3.78	27.97	321.77
	Radix, $\ell = 2$	0.22	0.63	4.26	35.59	245.90
	Radix, $\ell = 3$	0.22	0.70	4.73	37.91	392.56
	Radix, $\ell = 4$	0.19	0.63	4.45	36.04	227.61
	Quicksort	0.79	3.07	6.17	44.48	496.02
8	Radix, $\ell = 3$	0.27	0.89	5.51	41.58	428.80
	Radix, $\ell = 4$	0.26	0.88	5.67	41.19	425.96
	Radix, $\ell = 5$	0.28	1.03	7.34	45.58	476.02
	Quicksort	0.84	2.99	6.00	44.56	547.25
16	Radix, $\ell = 3$	0.38	1.17	7.78	53.03	575.23
	Radix, $\ell = 4$	0.35	1.16	7.96	50.41	545.89
	Radix, $\ell = 5$	0.38	1.44	11.08	59.52	659.95
	Quicksort	1.06	2.62	5.53	56.27	807.94
32	Radix, $\ell = 1$	0.84	2.78	22.66	155.37	$\perp$
	Radix, $\ell = 2$	0.53	1.84	14.00	90.92	$\perp$
	Radix, $\ell = 3$	0.46	1.66	12.52	79.83	809.18
	Radix, $\ell = 4$	0.43	1.69	13.28	71.49	733.29
	Radix, $\ell = 5$	0.49	2.14	17.86	84.68	846.08
	Radix, $\ell = 6$	0.56	2.78	25.10	93.81	$\perp$
	Quicksort	0.68	11.08	32.95	41.48	270.96

Table 6: Communication in MB to sort n m -bit numbers with semi-honest two-party computation using oblivious transfer (SoftSpokenOT). $\perp$ indicates omitted runs to save resources.

m	Variant	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	Radix, $\ell = 1$	55	183	1,832	18,456	186,403
	Quicksort	107	834	6,566	76,673	994,341
2	Radix, $\ell = 1$	65	283	2,846	28,265	286,240
	Radix, $\ell = 2$	56	188	1,884	18,676	188,394
	Quicksort	115	879	6,810	79,602	1,028,480
4	Radix, $\ell = 1$	47	474	4,735	47,910	486,417
	Radix, $\ell = 2$	29	294	2,899	28,731	290,710
	Radix, $\ell = 3$	30	300	2,972	29,428	297,264
	Radix, $\ell = 4$	21	205	2,000	19,756	198,726
	Quicksort	130	954	7,298	85,467	1,097,560
8	Radix, $\ell = 3$	44	426	4,111	40,610	410,569
	Radix, $\ell = 4$	36	338	3,204	31,548	317,961
	Radix, $\ell = 5$	38	362	3,463	33,990	341,588
	Quicksort	140	1,009	8,289	97,151	1,234,760
16	Radix, $\ell = 3$	81	747	7,357	72,822	737,432
	Radix, $\ell = 4$	64	568	5,543	54,697	552,181
	Radix, $\ell = 5$	69	619	6,062	59,589	599,518
	Quicksort	190	989	8,217	120,241	1,498,790
32	Radix, $\ell = 1$	322	3,179	32,053	325,926	$\perp$
	Radix, $\ell = 2$	177	1,696	16,981	172,531	$\perp$
	Radix, $\ell = 3$	135	1,253	12,469	126,478	1,283,810
	Radix, $\ell = 4$	109	1,002	9,917	100,412	1,017,700
	Radix, $\ell = 5$	107	980	9,678	97,855	989,953
	Radix, $\ell = 6$	110	996	9,830	99,223	$\perp$
	Quicksort	107	5,380	10,453	76,673	451,720

4 Sparse Matrix Multiplication

4.1 Sparse matrix multiplication without ORAM

Damie et al. [DHP26] presented a sparse multiplication algorithm without ORAM. It takes the inputs as column-wise and row-wise lists for the left- and right-hand inputs X and Y , respectively, such that every list contains index-value tuples. For example, if the first row of matrix M is represented by

$$(50, 10), (200, 20),$$

that would mean that $M_{0,50} = 10$ and $M_{0,200} = 20$, and that all other entries in the row are zero. Having the matrices in said format allows for pairing all entries for element-wise multiplication, that is, creating a list of tuples

$$(i, j, M_{i,k} \cdot M_{k,j})$$

where i and j correspond to the row and column indices of the output, respectively. Note that the sparsity means that not all possible i and j will appear, but all k for which the input lists are non-empty.

This list can then be sorted according to the row and column indices, putting all products that have to be summed for the output together. At this stage, Damie et al. use a linear scan for the summation, detecting a new output entry whenever the row or column index changes. They also set the coordinates in entries made obsolete by the summation to a special symbol $\perp$. The latter makes the protocol linear in the number of rounds, which they suggest improving to logarithmic by using the parallel-programming approach by Ladner and Fischer [LF80]. In Protocol 14, we will adapt this to a constant-round protocol when using oblivious shuffling, thus obtaining a constant-round protocol in settings where oblivious shuffling permits it.

In a last step, Damie et al. compare all coordinate entries and reveal an oblivious shuffling of the comparison result. This effectively reveals the number of relevant entries in the output, and it allows removing obsolete entries. What remains is a list representing the output matrix Z in the format

$$(i, j, Z_{i,j})$$

such that for all row indices i and column indices j there exists at least one k such $X_{i,k}$ and $Y_{k,j}$ were present in the respective input representation.

The issue with this format is that it cannot easily be reversed to the row- or column-wise representation required for input. There are no guarantees for the frequency of any particular entry, and thus, splitting into publicly known parts is not straightforward. Solving this is at the heart of our approach, which enables computing the higher powers required by graph cycle detection.

4.2 Our Protocol Completion

In this section, we will demonstrate our core protocol using an example. Consider the following list of length nine:

$$(0, 0, 0), \dots, (0, 0, 0), (0, 1, 1), (1, 2, 1), (2, 1, 1), (2, 2, 1). \quad (1)$$

For simplicity, we use triples here because the value entries do not matter (as opposed to the non-zero indicators in the last position). The first two entries of every tuple indicate a row-column index pair. For non-zero entries, they are all unique, and the list is sorted according to them. In terms of row indices, the list is not balanced. There are at least three entries with index 0, one with index 1, and two with index 2. The aim of the algorithm is

to obtain a list that has an equal number across all row indices, so that it can be split by row obviously.

By comparing adjacent row indices, we can form a group indicator list as described earlier:

$$1, 0, \dots, 0, 1, 1, 0.$$

Using this and `GroupSum`, we can create a list that counts the number of valid entries with the same row index:

$$1, 1, \dots, 1, 1, 2, 2.$$

Using the group indicator once more, we can then sort this such that the first three entries correspond to the valid entry number per index:

$$1, 1, 2, 1, 1, 1, 1, 2.$$

No, we take these first three entries and repeat them by subtracting 1 with every repetition:

$$0, 0, 1, -1, -1, 0, -2, -2, -1. \quad (2)$$

Consider now the repetition of possible indices $(0, 1, 2)$:

$$0, 1, 2, 0, 1, 2, 0, 1, 2.$$

Sorting this according to whether (2) is non-negative gives

$$0, 1, 2, 2, 0, 1, 0, 1, 2.$$

We can already see that the beginning of this corresponds to the relevant entries we started out with. What remains is to place the last five indices into the right positions of the original list. To do so, we sort (1) according to the inverse of the non-zero bits. This means the irrelevant entries will all be at the end, and thus, placing the indices is straightforward. Lastly, we sort the resulting list according to the row indices, which gives the desired result.

4.3 Algorithm Specification

Protocol 12 follows the blueprint by Damie et al. [DHP26] with two major changes. We operate with global sparsity parameters to hide the individual non-zero counts, as the latter potentially give away considerable amounts of information. More consequentially, we output a row-wise sparse matrix instead of a cell-wise sparse matrix. To do so, we first assign cyclical row numbers to all empty entries in step 13. This ensures that every row has at least one entry, which is a requirement for the group-wise data structure in our row-forming algorithm run after the aggregation.

If the inputs of an $n \times m \times l$ matrix multiplication have sparsity parameters μ and λ , that is, the maximum number of non-zero entries for the columns of the left-hand multiplicand and the right-hand multiplicand, respectively, the intermediate list Z has size $m\nu\lambda$. The $\nu\lambda$ factor comes from the fact that the algorithm first considers the products of all combinations of elements from the left-side column and the right-side row. The aggregation protocol then sums all relevant entries. For example, if $C_{i,j} = A_{i,k}B_{k,j} + A_{i,k'}B_{k',j}$ for non-zero $(A_{i,k}, A_{i,k'}, B_{k,j}, B_{k',j})$ and $k \neq k'$, this would only be reflected after the aggregation in step 17. If the rows of the product C have at most μ non-zero entries, the row-forming algorithm only has to consider $n\mu$ entries. In the most general case, dismissing the remaining entries should only happen after the aggregation. However, we found that in our use case, a stricter condition applies, which allows reducing before the aggregation. More concretely, we impose the condition not only on the row but on all relevant element

Protocol 12: Sparse matrix multiplication with row-wise output
Input

1. Sparse matrix $[A]$ ($n \times m$, column-wise, at most ν non-zero entries per column)
2. Sparse matrix $[B]$ ($m \times l$, row-wise, at most λ non-zero entries per row)
3. Integer μ such that $\mu \leq \nu \cdot \lambda$

Output Sparse matrix $[C]$ of dimension $n \times l$ (row-wise representation, at most μ non-zero entries per row) such that $C = AB$ if

1. AB has at most μ non-zero entries per row
2. For every $i = 0, \dots, n-1$, $\{A_{i,k} \cdot B_{k,j} : k = 0, \dots, m-1, j = 0, \dots, l-1\}$ has size at most μ (early reduction only)

```

1:  $[Z] \leftarrow \emptyset$ 
2: for  $k = 0, \dots, m-1$  do
3:   for  $i = 0, \dots, \nu-1$  do
4:     for  $j = 0, \dots, \lambda-1$  do
5:       Append  $(\text{coord}(A_i^{(k)}), \text{coord}(B_j^{(k)}), \text{val}(A_i^{(k)}) \cdot \text{val}(B_j^{(k)}), \text{nz}(A_i^{(k)}) \cdot$ 
          $\text{nz}(B_j^{(k)}))$  to  $[Z]$ 
6:     end for
7:   end for
8: end for
9:                                      $\triangleright [Z]$  is now a matrix of dimension  $(m\nu\lambda) \times 4$ 
10: Sort  $[Z]$  by the last column
11: Keep only the last  $n\mu$  rows in  $[Z]$                                       $\triangleright$  Early reduction for efficiency
12: for  $k = 0, \dots, m\mu-1$  do
13:    $Z_{k,0} \leftarrow \text{Mux}(Z_{k,3}, Z_{k,0}, k \bmod n)$ 
14:                                      $\triangleright$  Assign cyclic row number to zero entries
15: end for
16: Sort  $[Z]$  by the first two columns                                      $\triangleright$  Row and column numbers
17:  $[Z] \leftarrow \text{AggEqualCoord}([Z])$ 
18: return  $\text{MakeRowMatrix}([Z])$ 

```

products in general, that is, for every i , there are at most μ non-zero products $A_{i,k} \cdot B_{k,l}$ across all k and l .

Protocol 13 presents our adaptation of the aggregation protocol by Damie et al. Unlike them, we make use of the potentially constant-round primitives going back to Hamada et al. [HIKC23]. This includes group-wise binary OR, which can be constructed from group-wise sum using the fact that

$$\bigvee_i x_i = \left(\sum_i x_i \right) \bmod 2$$

if the domain is large enough.

Protocol 14 shows our approach to convert the cell-wise matrix representation produced earlier back to row-wise presentation. The main difficulty is to sort the cell-wise representation into equal-size blocks per row such that it is easy to split them obliviously. This has to happen while the row indices are kept secret.

Protocol 13: Equal coordinate aggregation (AggEqualCoord)

Input $L \times 4$ matrix $[Z]$ sorted by the first two as produced in Protocol 12.

Output A four-column matrix such that for every coordinate pair (a, b) there is at most one row $(a, b, v, 1)$ such that $v = \sum v_i$ for all v_i where there is a row $(a, b, v_i, 1)$ in $[Z]$. All other rows $(a, b, \cdot, \cdot)$ will be $(a, b, 0, 0)$.

```

1:  $[g_1] \leftarrow 1$ 
2: for  $i = 2, \dots, L$  do
3:    $[g_i] \leftarrow ([Z_{i,1}] \stackrel{?}{=} [Z_{i+1,1}]) \wedge ([Z_{i,1}] \stackrel{?}{=} [Z_{i+1,1}])$ 
4: end for                                 $\triangleright$  Group indicator for coordinate tuples
5:  $[v] \leftarrow \text{GroupSum}([g], [Z_{\cdot,2}])$        $\triangleright$  Total value for coordinate tuples
6:  $[w] \leftarrow \text{GroupOr}([g], [Z_{\cdot,3}])$        $\triangleright$  Non-zero indicator for coordinate tuples
7: for  $i = 1, \dots, L$  do
8:    $[Z_{i,2}] \leftarrow [v_i] \cdot [g_i]$            $\triangleright$  Non-zero only once per coordinate tuple
9:    $[Z_{i,3}] \leftarrow [w_i] \cdot [g_i]$            $\triangleright$  Non-zero only once per coordinate tuple
10: end for
11: return  $[Z]$ 

```

Complexity The asymptotic complexity of Protocol 12, excluding sub-protocols, is dominated by the single-bit sorting of $m\nu\lambda$ in step 10 and the coordinate-wise sorting in step 16. Protocol 13 involves an oblivious shuffling of size $O(n\mu)$ or $O(m\nu\lambda)$, depending on the use of early reduction, as well as the same number of further operations. Finally, Protocol 14 uses coordinate-wise sorting of $n\mu$ elements, which dominates all other operations. As a result, the overall asymptotic cost corresponds to single-bit sorting of $m\nu\lambda$ elements and $(\log n + \log l)$ -bit sorting of $n\mu$ elements in the case of early reduction. This emphasizes the relevance of sorting because our benchmarks show that this dominance extends to the concrete costs.

Security The security of Protocols 12–14 follows from the fact that they are almost entirely run within the arithmetic black box. The only additional information revealed is the number of groups in Protocol 14. However, this number is predictable if the inputs achieve the condition in Protocol 12, that is, if there are too many non-zero entries in the matrix product.

Protocol 14: Row matrix creation (MakeRowMatrix)**Input**

1. $(n \cdot \mu) \times 4$ matrix $[Z]$ as produced by `AggEqualCoord`, sorted by first entry of rows
2. $n, l \in \mathbb{N}$ such that $l \mid \mu$

Output Row-wise sparse $n \times l$ matrix C

```

1:  $[g_1] \leftarrow 0$ 
2: for  $i = 1, \dots, L$  do
3:    $[g_i] \leftarrow \neg([Z_{i,0}] \stackrel{?}{=} [Z_{i-1,0}])$ 
4: end for ▷ Create groups for row coordinates
5: Reveal  $\sum_i [g_i]$  and abort if it is not  $n$ 
6:  $[c] \leftarrow \text{GroupSum}([g], [Z_{\cdot,3}])$ 
7: Sort  $[c]$  according to  $\neg[g]$  ▷  $[c_i]$  is the count of  $i$  in  $Z$  for  $i = 1, \dots, n$ 
8: for  $i = 1, \dots, n$  do
9:   for  $j = 1, \dots, \mu$  do
10:     $k \leftarrow (i-1) \cdot \mu + j$ 
11:     $[d_k] \leftarrow [c_i] \stackrel{?}{<} j$ 
12:     $[e_k] \leftarrow j$ 
13:   end for
14: end for
15: Sort  $[d]$  and  $[e]$  according to  $[d]$ 
16: Sort  $[Z]$  according to the negation of its last column
17: ▷ All non-zero entries come first now
18: for  $i = 1, \dots, n\mu$  do
19:    $[Z_{i,0}] \leftarrow \text{Mux}([d_i], [e_i], [Z_{i,0}])$  ▷ Replace row indices for zero entries
   according to determined frequency
20: end for
21: Sort  $[Z]$  by its first column
22: ▷ First column is now  $\mu$  repetitions of every row index
23: for  $i = 1, \dots, n$  do
24:   for  $j = 1, \dots, \mu$  do
25:     $k \leftarrow (i-1) \cdot \mu + j$ 
26:     $[C_{i,j}] = ([Z_{k,1}, [Z_{k,2}], [Z_{k,3}]])$  ▷ Split into blocks by row
27:   end for
28: end for
29: return  $[C]$ 

```

5 Graph Pattern Matching

Cleartext graph algorithms often make heavy use of RAM techniques such as breadth- or depth-first search. While it is possible to translate these to the setting of secure computation using oblivious RAM, the latter comes with a considerable overhead [KS14]. It therefore can be advantageous to consider other approaches.

One such approach derives from the fact that graphs can be described as matrices. If $V = \{v_i\}_{i \in [n]}$ for a graph $G = (V, E)$, let M be the $n \times n$ -dimensional adjacency matrix defined as follows:

$$M_{i,j} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Now consider the product $P = M \cdot M$. By definition,

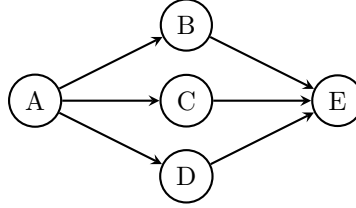
$$P_{i,j} = \sum_{k \in [n]} M_{i,k} \cdot M_{k,j}.$$

Given the definition of $M_{i,j}$,

$$P_{i,j} = |\{k \mid (v_i, v_k) \in E \wedge (v_k, v_j) \in E\}|.$$

In many real-world contexts, such as financial transactions, graphs have low degrees. Sparse matrix multiplication can thus be used to compute P .

A Simple Graph Pattern. The following pattern in transaction graphs has been identified as being used for money laundering [STZ⁺21, ABvN⁺23], dubbed scatter-gather:



From the above definitions, it follows trivially that the values in P correspond to the number of intermediate nodes in the scatter-gather pattern. Using a synthetic anti-money laundering dataset, we have confirmed this correspondence experimentally, in that our implementation of sparse matrix multiplication found every instance of this pattern marked as suspicious as well as all false negatives.

5.1 Cycle Detection

A cycle in graph $G = (V, E)$ is a list of vertices $\{v_i\}_{i \in [l]}$ that form a cycle as follows:

$$(v_i, v_{(i+1) \bmod l}) \in E.$$

If the list is disjoint, the cycle is called a simple cycle. The identification of such cycles has been studied for a long time [Joh75]. This implies sparse matrix multiplications when using Algorithm 15.

In other words, the entries of P indicate the number of paths of length two from one vertex to another, and the diagonal is non-zero for all vertices that are part of a cycle of length two. A simple induction argument shows that this generalizes to M^n for cycles of length n , giving rise to Algorithm 15.

Algorithm 15 also outputs repeated cycles. For example, if v_i is part of k cycles of length two, $(M^2)_{i,i} = k$, and thus, $(M^4)_{i,i} \geq k^2$. A partial remedy to this increase is

Algorithm 15: Cycle detection using matrix multiplication

Inputs Adjacency matrix M for graph $G = (V, E)$ with $n = |V|$, $m \in \mathbb{N}$

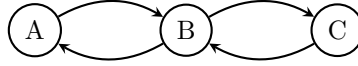
Output $(m-1) \times n$ -dimensional matrix C such that $C_{i-2,j}$ is the number of cycles of length i involving vertex j

```

 $P \leftarrow M$ 
for  $i \in [2, m]$  do
   $P \leftarrow P \cdot M$ 
  for  $j \in [n]$  do
     $C_{i-2,j} \leftarrow P_{j,j}$ 
  end for
end for

```

to set the diagonal of P to all zeros before every multiplication. However, this does not completely solve the issue, as exemplified by the following graph:



The matrix M formed by this graph is

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

The smallest powers of M can be computed as follows:

$$M^2 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad M^3 = \begin{pmatrix} 0 & 2 & 0 \\ 2 & 0 & 2 \\ 0 & 2 & 0 \end{pmatrix} \quad M^4 = \begin{pmatrix} 2 & 0 & 2 \\ 0 & 4 & 0 \\ 2 & 0 & 2 \end{pmatrix}.$$

It is easy to see that M^n represents the number of paths of length n from one vertex to another. For example, the value 4 in M^4 represents the following paths:

1. B-A-B-A-B
2. B-A-B-C-B
3. B-C-B-C-B
4. B-C-B-A-B

None of these paths is a simple cycle, but combinations of smaller cycles. Consider now the adaptation of Algorithm 15 with setting the diagonal to zero. More concretely, let $P_1 = M$, $P_i = P'_{i-1} \cdot M$, and $P'_i = z(P_i)$, where z denotes the function that sets the diagonal to zero. It follows that

$$P_2 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad P_3 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad P_4 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

P_4 represents a reduced number of paths, but it still exposes non-simple cycles, namely A-B-C-B-A and C-B-A-B-C. Looking at P_3 , the reason can be seen in that P_3 represents the path A-B-C-B, which escapes the “annihilation” in the construction of P'_2 .

Algorithm 16: Improved cycle detection using matrix multiplication**Inputs**

- Adjacency matrix M for graph $G = (V, E)$ with $n = |V|$
- Adjacency matrix M' for graph $G' = (V, E')$ where $E' \subset E$ such that $E \setminus E' = \{(u, v) : (u, v) \in E \wedge (v, u) \in E\}$
- $m \in \mathbb{N}$

Output $(m-1) \times n$ -dimensional matrix C such that $C_{i-2,j}$ is the number of cycles of length i involving vertex i such that, for the relevant edge (j, u) in the cycle, $(u, j) \notin E$, unless $i = 2$.

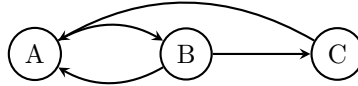
```

 $P_1 \leftarrow z(M)$ 
for  $j \in [2, m]$  do
  if  $i = 2$  then
     $P \leftarrow P_1 \cdot P_1$ 
  else
     $P \leftarrow P_{i-1} \cdot M'$ 
  end if
  for  $j \in [n]$  do
     $C_{i-2,j} \leftarrow P_{i,i}$ 
  end for
  if  $i = 2$  then
     $P_2 \leftarrow z(M \cdot M')$ 
  else
     $P_i \leftarrow z(P)$ 
  end if
end for

```

We now propose an improved algorithm that avoids this expansion of cycles of length two to create cycles of length of any even number. The core idea is to discard all edges that are part of length-two cycles when computing further cycles. This is achieved by constructing two adjacency matrices. Note that this is realistic in a privacy-preserving setting if all edges involving a vertex are known to the same party, which then can simply detect all cycles of length two. Algorithm 16 presents the details.

We have analyzed the graph implied by the transactions in a synthetic dataset for money laundering detection [OCA⁺23], and we have found that Algorithm 16 correctly identifies all nodes in simple cycles, see Section 5.1.3. This is because no edge in a two-cycle is also part of a longer cycle. However, there are graphs where Algorithm 16 does not correctly identify all nodes. Consider the following graph:



The adjacency matrices for this graph are

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad M' = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix},$$

and it follows that the relevant matrix product is

$$P_2 \cdot M' = z(M \cdot M') \cdot M' = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

While it does not include all relevant nodes, as the “correct” result would be a diagonal of all ones, it still includes the first node as a member of a cycle of length three.

Theorem 1. *For any vertex i , Algorithm 16 outputs the number of all non-repeating paths of length j from i to i that do not contain two-cycle edges other than the one going out from i .*

Proof. The case of $i = 2$ is trivial. We thus use $i = 3$ as the base case. Consider the diagonal of $P_2 \cdot M' = z(M \cdot M') \cdot M'$. $M \cdot M'$ counts all paths between two vertices not ending with a two-cycle edge due to the definition of M' , and $z(M \cdot M')$ counts all such non-cycle paths. The multiplication with M' then counts all paths of length three that do not start with a two-cycle and that do not end with a two-cycle edge, which proves the case for $i = 3$. A similar argument proves the induction. $\square$

5.1.1 A parametrizable algorithm

We now confront the issue of non-simple cycles in a different way, still focusing on two-cycles attached to each other giving rise to non-repetitive paths. So far, we have only used simple adjacency matrices, that is, only containing the values 0 and 1. Based on the assumption that it is possible to treat edges in two-cycles differently, we now propose to give a different weight to them. For a graph $G = (V, E)$. Let $w \in (0, 1)$ and the matrix M_w be defined as

$$M_{i,j} = \begin{cases} 1 & (i, j) \in E \wedge (j, i) \notin E \\ w & (i, j) \in E \wedge (j, i) \in E \\ 0 & \text{else} \end{cases} \quad (3)$$

Clearly, a path that passes k times through a two-cycle edge will have weight w^k rather than one. Using the synthetic AML dataset [OCA⁺23], we have found that $w = 0.16$ prevents false positives up to cycle length 20. However, this might be of limited use, given that ignoring all two-cycles does not lead to any false negatives, as seen earlier.

5.1.2 Benchmarks

We have implemented Algorithm 16 in MP-SPDZ. To gauge its cost, we have run it with semi-honest honest-majority three-party computation on AWS c5.9xlarge instances in the same region. We have used radix sort with $\ell = 2$, as Tables 1 and 2 suggest its optimality across a range of parameters. Note that the sort key length for graph size n is at most $O(\log n)$, which makes Quicksort unlikely to be optimal for the parameters used. Table 7 shows that the cost is primarily influenced by the graph size and the maximum degree. This is expected, as the main cost lies in the size and the sparsity of the sparse matrix multiplication.

5.1.3 Dataset Results

We have tested Algorithm 16 on a publicly available synthetic dataset for money laundering detection [OCA⁺23]. One type of suspicious activity in said dataset is cyclic transactions, where money is sent over several links until it ends up at the original account. Accounts and transactions can be modelled as nodes and edges in a directed graph, and cyclic transactions correspond to graph cycles in this model. The dataset contains 855,460 nodes. Table 8 shows that Algorithm 16 finds all nodes in cycles with no false positives or negatives.

Table 7: Time and total communication for cycle detection up to length 10 in semi-honest 3PC. Reach means the maximum number of nodes reachable from any node, which corresponds to μ in the matrix multiplication in Protocol 12.

Graph size	Max. degree	Max. reach	Time	Communication
100	10	20	2 s	0 GB
1,000	10	20	5 s	2 GB
1,000	100	20	5 s	2 GB
1,000	100	100	23 s	9 GB
10,000	10	20	34 s	21 GB
10,000	100	20	35 s	22 GB
10,000	100	100	210 s	110 GB
100,000	10	20	409 s	261 GB
100,000	100	20	419 s	271 GB
100,000	100	100	2434 s	1308 GB

Table 8: Cycles in synthetic anti-money laundering dataset.

Cycle length	#Cycles	#Nodes in dataset	#Nodes found by Alg. 16
2	46917	66155	66155
3	0	0	0
4	0	0	0
5	3	15	15
6	1	6	6
7	2	14	14
8	3	24	24
9	5	45	45
10	7	70	70
11	3	33	33
12	3	36	36
13	5	65	65
14	1	14	14
15	3	45	45
16	0	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0

Algorithm 17: Parametrizable cycle detection using sparse matrix multiplication
Inputs

- $w \in (0, 1)$
- Adjacency matrix M_w for graph $G = (V, E)$ with $n = |V|$ as defined in Equation (3).
- $m \in \mathbb{N}$

Output $(m - 1) \times n$ -dimensional matrix C such that $C_{i-2,j}$ is an approximate number of simple cycles of length i involving vertex j .

```

 $P_1 \leftarrow z(M)$ 
for  $i \in [2, m]$  do
   $P \leftarrow P_{i-1} \cdot M$ 
  for  $j \in [n]$  do
     $C_{i-2,j} \leftarrow \lfloor P_{j,j} \rfloor$ 
  end for
   $P_i \leftarrow z(P)$ 
end for

```

Acknowledgements

We would like to acknowledge helpful suggestions by Sevvandi Kandanaarachchi and Ashfaqur Rahman.

References

- [ABvN⁺23] Erik Altman, Jovan Blanuša, Luc von Niederhäusern, Béni Egressy, Andreea Anghel, and Kubilay Atasu. Realistic synthetic financial transactions for anti-money laundering models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*, Red Hook, NY, USA, 2023. Curran Associates Inc.
- [AFO⁺21] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 610–629. ACM Press, November 2021. doi:10.1145/3460120.3484560.
- [AHI⁺22a] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 125–138. ACM Press, November 2022. doi:10.1145/3548606.3560691.
- [AHI⁺22b] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. Cryptology ePrint Archive, Report 2022/1595, 2022. URL: <https://eprint.iacr.org/2022/1595>.

- [AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, January 1983. doi:[10.1007/BF02579338](https://doi.org/10.1007/BF02579338).
- [APRR24] Navid Alamati, Guru-Vamsi Policharla, Srinivasan Raghuraman, and Peter Rindal. Improved alternating-moduli PRFs and post-quantum signatures. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VIII*, volume 14927 of *LNCS*, pages 274–308. Springer, Cham, August 2024. doi:[10.1007/978-3-031-68397-8_9](https://doi.org/10.1007/978-3-031-68397-8_9).
- [Bat68] Kenneth E. Batchier. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery. doi:[10.1145/1468075.1468121](https://doi.org/10.1145/1468075.1468121).
- [BBM⁺25] Eli Baum, Sam Buxbaum, Nitin Mathai, Muhammad Faisal, Vasiliki Kalavri, Mayank Varia, and John Liagouris. Orq: Complex analytics on private data with strong security guarantees. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 802–833, New York, NY, USA, 2025. Association for Computing Machinery. doi:[10.1145/3731569.3764833](https://doi.org/10.1145/3731569.3764833).
- [BLT14] Dan Bogdanov, Sven Laur, and Riivo Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *Secure IT Systems: 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15–17, 2014, Proceedings*, page 59–74, Berlin, Heidelberg, 2014. Springer-Verlag. doi:[10.1007/978-3-319-11599-3_4](https://doi.org/10.1007/978-3-319-11599-3_4).
- [CDN15] Ronald Cramer, Ivan Damgård, and Jesper Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 08 2015. doi:[10.1017/CB09781107337756](https://doi.org/10.1017/CB09781107337756).
- [CGP20] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 342–372. Springer, Cham, December 2020. doi:[10.1007/978-3-030-64840-4_12](https://doi.org/10.1007/978-3-030-64840-4_12).
- [CHI⁺19] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. An efficient secure three-party sorting protocol with an honest majority. Cryptology ePrint Archive, Report 2019/695, 2019. URL: <https://eprint.iacr.org/2019/695>.
- [DFK⁺06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Berlin, Heidelberg, March 2006. doi:[10.1007/11681878_15](https://doi.org/10.1007/11681878_15).
- [DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 160–179. Springer, Berlin, Heidelberg, March 2009. doi:[10.1007/978-3-642-00468-1_10](https://doi.org/10.1007/978-3-642-00468-1_10).
- [DHPR26] Marc Damie, Florian Hahn, Andreas Peter, and Jan Ramon. Secure sparse matrix multiplications and their applications to privacy-preserving machine learning. In *Proceedings of the Sixteenth ACM Conference on Data and*

- Application Security and Privacy*, CODASPY '26, page 71–84, New York, NY, USA, 2026. Association for Computing Machinery. doi:[10.1145/3800506.3803493](https://doi.org/10.1145/3800506.3803493).
- [DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 247–264. Springer, Berlin, Heidelberg, August 2003. doi:[10.1007/978-3-540-45146-4_15](https://doi.org/10.1007/978-3-540-45146-4_15).
- [DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015. doi:[10.14722/ndss.2015.23113](https://doi.org/10.14722/ndss.2015.23113).
- [EGK⁺20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Cham, August 2020. doi:[10.1007/978-3-030-56880-1_29](https://doi.org/10.1007/978-3-030-56880-1_29).
- [EKR18] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2(2-3):70–246, 12 2018. arXiv:<https://securecomputation.org>, doi:[10.1561/33000000019](https://doi.org/10.1561/33000000019).
- [GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 465–482. Springer, Berlin, Heidelberg, April 2012. doi:[10.1007/978-3-642-29011-4_28](https://doi.org/10.1007/978-3-642-29011-4_28).
- [GKK⁺12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012. doi:[10.1145/2382196.2382251](https://doi.org/10.1145/2382196.2382251).
- [Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987. doi:[10.1145/28395.28416](https://doi.org/10.1145/28395.28416).
- [HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.
- [HICT14] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint Archive, Report 2014/121, 2014. URL: <https://eprint.iacr.org/2014/121>.
- [HIKC23] Koki Hamada, Dai Ikarashi, Ryo Kikuchi, and Koji Chida. Efficient decision tree training with new data structure for secure multi-party computation. *PoPETs*, 2023(1):343–364, January 2023. doi:[10.56553/popets-2023-0021](https://doi.org/10.56553/popets-2023-0021).
- [HKI⁺13] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC 12*, volume 7839 of *LNCS*, pages 202–216. Springer, Berlin, Heidelberg, November 2013. doi:[10.1007/978-3-642-37682-5_15](https://doi.org/10.1007/978-3-642-37682-5_15).

- [Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, July 1961. doi:[10.1145/366622.366644](https://doi.org/10.1145/366622.366644).
- [Joh75] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975. doi:[10.1137/0204007](https://doi.org/10.1137/0204007).
- [Kel19] Marcel Keller. The oblivious machine - or: How to put the C into MPC. In Tanja Lange and Orr Dunkelman, editors, *LATINCRYPT 2017*, volume 11368 of *LNCS*, pages 271–288. Springer, Cham, September 2019. doi:[10.1007/978-3-030-25283-0_15](https://doi.org/10.1007/978-3-030-25283-0_15).
- [Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020. doi:[10.1145/3372297.3417872](https://doi.org/10.1145/3372297.3417872).
- [Kel25] Marcel Keller. Rep3 reloaded: On the cost of function-dependent preprocessing in semi-honest 3PC with honest majority. Cryptology ePrint Archive, Report 2025/919, 2025. URL: <https://eprint.iacr.org/2025/919>.
- [KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 506–525. Springer, Berlin, Heidelberg, December 2014. doi:[10.1007/978-3-662-45608-8_27](https://doi.org/10.1007/978-3-662-45608-8_27).
- [LADM14] John Launchbury, Dave Archer, Thomas DuBuisson, and Eric Mertens. Application-scale secure multiparty computation. In Zhong Shao, editor, *Programming Languages and Systems*, pages 8–26, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:[10.1007/978-3-642-54833-8_2](https://doi.org/10.1007/978-3-642-54833-8_2).
- [LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, oct 1980. doi:[10.1145/322217.322232](https://doi.org/10.1145/322217.322232).
- [MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *USENIX Security 2004*, pages 287–302. USENIX Association, August 2004. URL: <http://www.usenix.org/publications/library/proceedings/sec04/tech/malkhi.html>.
- [OCA⁺23] Berkan Oztas, Deniz Cetinkaya, Festus Adedoyin, Marcin Budka, Huseyin Dogan, and Gokhan Aksu. Enhancing anti-money laundering: Development of a synthetic transaction monitoring dataset. In *2023 IEEE International Conference on e-Business Engineering (ICEBE)*, pages 47–54, 2023. doi:[10.1109/ICEBE59045.2023.00028](https://doi.org/10.1109/ICEBE59045.2023.00028).
- [OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303. ACM Press, May 1997. doi:[10.1145/258533.258606](https://doi.org/10.1145/258533.258606).
- [PRRS25] Stanislav Peceny, Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. Efficient permutation correlations and batched random access for two-party computation. In Tibor Jager and Jiaxin Pan, editors, *PKC 2025, Part IV*, volume 15677 of *LNCS*, pages 76–109. Springer, Cham, May 2025. doi:[10.1007/978-3-031-91829-2_3](https://doi.org/10.1007/978-3-031-91829-2_3).
- [PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Berlin, Heidelberg, December 2009. doi:[10.1007/978-3-642-10366-7_15](https://doi.org/10.1007/978-3-642-10366-7_15).

- [Rin26a] Peter Rindal. Private communication, 2026.
- [Rin26b] Peter Rindal. Secure join. <https://github.com/ladnir/secure-join/>, 2026.
- [RRK⁺20] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CryptFlow2: Practical 2-party secure inference. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 325–342. ACM Press, November 2020. doi:10.1145/3372297.3417274.
- [STZ⁺21] Michele Starnini, Charalampos E. Tsourakakis, Maryam Zamanipour, André Panisson, Walter Allasia, Marco Fornasiero, Laura Li Puma, Valeria Ricci, Silvia Ronchiadin, Angela Ugrinoska, Marco Varetto, and Dario Moncalvo. Smurf-based anti-money laundering in time-evolving transaction networks. In Yuxiao Dong, Nicolas Kourtellis, Barbara Hammer, and Jose A. Lozano, editors, *Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track*, pages 171–186, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-86514-6_11.
- [Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968. doi:10.1145/321439.321449.
- [WLG⁺10] Guan Wang, Tongbo Luo, Michael T. Goodrich, Wenliang Du, and Zutao Zhu. Bureaucratic protocols for secure two-party sorting, selection, and permuting. In Dengguo Feng, David A. Basin, and Peng Liu, editors, *ASIACCS 10*, pages 226–237. ACM Press, April 2010. doi:10.1145/1755688.1755716.