Efficient Permutation Correlations and Batched Random Access for Two-Party Computation

Stanislav Peceny[1]*, Srinivasan Raghuraman[2], Peter Rindal[3], and Harshal Shah[3]

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

Abstract. In this work we formalize the notion of a two-party permutation correlation $(A, B), (C, \pi)$ s.t. $\pi(A) = B + C$ for a random permutation π of n elements and vectors $A, B, C \in \mathbb{F}^n$. This correlation can be viewed as an abstraction and generalization of the Chase et al. (Asiacrypt 2020) share translation protocol. We give a systematization of knowledge for how such a permutation correlation can be derandomized to allow the parties to perform a wide range of oblivious permutations of secret-shared data. This systematization immediately enables the translation of various popular honest-majority protocols to be efficiently instantiated in the two-party setting, e.g. collaborative filtering, sorting, database joins, graph algorithms, and many more.

We give two novel protocols for efficiently generating a random permutation correlation. The first uses MPC-friendly PRFs to generate a correlation of n elements, each of size $\ell = \log|\mathbb{F}|$ bits, with $O(n\ell)$ bit-OTs, time, communication, and only three rounds. Similar asymptotics previously required relatively expensive public-key cryptography, e.g. Paillier or LWE. Our protocol implementation for $n = 2^{20}, \ell = 128$ requires just 7 seconds & $\sim 2\ell n$ bits of communication, a respective 40 & $1.1\times$ improvement on the LWE solution of Juvekar at al. (CCS 2018). The second protocol is based on pseudo-random correlation generators and achieves an overhead that is *sublinear* in the string length ℓ, i.e. the communication and number of OTs is $O(n \log \ell)$. The overhead of the latter protocol has larger hidden constants, and therefore is more efficient only when long strings are permuted, e.g. in graph algorithms.

Finally, we present a suite of highly efficient protocols based on permutations for performing various batched random access operations. These include the ability to extract a hidden subset of a secret-shared list. More generally, we give ORAM-like protocols for obliviously reading and writing from a list in a batched manner. We argue that this suite of batched random access protocols should be a first class primitive in the MPC practitioner's toolbox.[1]

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

[1] The authors grant IACR a non-exclusive and irrevocable license to distribute the article under the https://creativecommons.org/licenses/by-nc/3.0/.

1 Introduction

Secure multi-party computation (MPC) is increasingly used to perform complex data intensive tasks while maintaining strong privacy guarantees. Examples include machine learning & data analytics, database joins, sorting and many more. A common thread to these complex tasks is the need to perform some type of random access into the processed data. Unfortunately, this is at odds with how MPC and similar technologies work. Typically, MPC protocols require the target function to be expressed as a circuit where all memory accesses are fixed and independent of the input. The advantage of this model is that it can reduce the task of designing protocols for a complex function to the simpler task of multiplying or adding two encrypted values. However, given only multiplication and addition gates, it is computationally expensive to implement random access memory. Each read/write operation essentially requires reading or writing to all of the memory. One line of work that aims to overcome this is oblivious RAM for MPC [LO13,WCS15,HKO23,PLS23]. While very flexible, this approach introduces a polylog overhead for each access and as such is still comparatively less efficient compared to the plaintext setting.

An alternative model has emerged that we call *batched random access.* Many of the most important applications can be made efficient in a circuit model with access to *random access gates.* In this model, the main part of the circuit is expressed with addition and multiplication gates. The circuit also has access to random access gates that take as input a (possibly secret-shared) selection function σ, and a list of values X. The output of the random access gate are the values of X selected by σ, i.e. $X_{\sigma(1)}, ..., X_{\sigma(m)}$. A similar gate can be defined for writing values into a list. The most ubiquitous example of such a gate is when σ is a permutation, i.e. $X_{\sigma(1)}, ..., X_{\sigma(m)}$ is simply a reordering of X. Permutations have proven extremely useful for a wide range of applications [AKK+23,BDG+22,AHI+22,FO20,CHI+19,NWI+15,MRR20]. We show that it is possible for these permutation gates to be implemented in concretely efficient linear time and constant depth/rounds, sidestepping the polylog overhead of oblivious RAM. We then use our efficient permutation protocols to build up more complex random access gates, culminating in the ability to perform complex batched read and write operations, similar to what oblivious RAM can support.

1.1 Applications of Secret-shared Permutation

Secret-shared permutation is an essential building block of many MPC protocols. Thus, reducing the cost of secret-shared permutation will lead to significant cost reductions in these protocols. We now present a non-exhaustive list of applications.

Shuffle. Two parties jointly shuffle an array and return additive secret sharing of the result. The security guarantee is that no party learns the random permutation corresponding to the shuffle. This technique has primarily been used in the three-party setting [AKK+23,BDG+22,AHI+22], but has also been generalized to n-party shuffle. [FO20] explicates the use of shuffle in MPC.

Merging and Sorting. The most efficient secure sorting algorithms rely on a shuffle-then-sort paradigm [HKI+13] or radix sort [CHI+19]. For the former, the idea is that after the shuffle, the (comparison-based) sorting algorithms can *reveal* the result of each comparison and move data based on the result without compromising security. This is because the comparison bit is independent of the underlying value after shuffling. Radix sort uses different techniques and permutes the data several times, once per bit of the sorting key. Due to the lack of efficient two-party permutation, these protocols have remained costly.

Similarly, permutations are a useful component for secure merging protocols, where two sorted lists are merged so that the final secret-shared list is ordered. E.g., [FO20]'s merge relies on secure shuffle in many subprotocols.

Graph Algorithms. Our improvement is particularly significant in frameworks that rely heavily on oblivious permutations. E.g., GraphSC [NWI+15] is a framework that enables efficient secure implementation of graph-based algorithms. It is inspired by parallelization techniques from Pregel [MAB+10], a programming model for developing parallel algorithms on large-scale graphs. More specifically, it securely implements Pregel's scatter, gather, and apply operations, which can be used to solve several important parallel data mining and machine learning algorithms. Without going into the details of the gather and scatter operations, invoking them requires oblivious permutations on the entire graph. As the graph is large and the operations are called repeatedly, we can get considerable savings by using our permutation.

Extraction and Filtering. A straightforward application of a permutation protocol is to extract elements from an array that satisfy a given condition. For example, consider a secret-shared vector where each entry has an associated flag indicating whether it meets a certain criterion or belongs to a set intersection. To extract the flagged elements (in secret-shared form while preserving security), we can obliviously permute the secret-shared array alongside its corresponding secret-shared flag array. After permuting, the parties can reveal the flag array and retrieve the flagged elements or discard those that do not meet the condition. This technique has broad applicability, including in applications such as [FO20].

Database Joins & Private Set Intersection. Secure database joins have now become an active research area [MRR20,BDG+22,HZF+22,LWDY24]. This line of research can be viewed as a generalization of private set intersection, where the input sets are secret-shared, have associated values, and can contain duplicates. They are essentially SQL tables. One can then consider various joins between these tables. [MRR20,BDG+22] demonstrate that one can compute such joins with $O(n \log n)$ overhead given access to efficient permutation protocol, or in

$O(n)$ time if one table has unique keys [LWDY24]. As there have been no efficient two-party permutation protocols until our work, [MRR20,BDG+22] focus on the honest majority setting.

Random Access Memory in Secure Multi-Party Computation. Commonly, MPC frameworks work in the circuit model where the function to be computed is expressed as a circuit. This is in contrast with the RAM model where memory accesses can be performed in an input-dependent way. To access an (encrypted) value at position i in memory, the parties executing the protocol must know i. If i is computed as a function of the input, then i can leak information about the input. Yet, many important applications such as stable matching [DEs16], sorting [AHI+22,HKI+13,CHI+19], merging [FO20,GRR24], graph scatter-gather algorithms [NWI+15], graph Dijkstra [Ost24], decision trees [TN21], privacy preserving advertising [DWA+21], database joins [BDG+22,HZF+22,LWDY24,CFL+24], private set union [KLS24,CSSW24], ride sharing [KMPP24] are not efficiently expressible in terms of circuits.

To mitigate this, one can equip a circuit-based MPC protocol with a special "random-access" gate, e.g. via garbled RAM [LO13]. This gate takes as input the index i and returns the memory value m_i, without leaking any information about i. A trivial solution for such a gate is linear scan, which reads all $O(n)$ memory locations and saves m_i when it is read. The gate then returns m_i obliviously. However, this solution adds an $O(n)$ factor to the running time. Specialized garbled RAM protocols are able to obliviously access m_i with polylog overhead [LO13,HKO23,PLS23]. Despite significant improvements, they remain expensive and have not been implemented until recently [YPHK23]. For example, the state-of-the-art garbled RAM of [HKO23] requires $O(T \log^3 n \cdot \log\log n)$ time to perform T RAM accesses into a memory of size n.

An alternative, proposed by [DS17], leverages a private information retrieval (PIR) scheme based on point functions [GI14] to construct a random-access gate. Their construction requires an amortized $O(\sqrt{n})$ overhead within the MPC protocol and additionally $O(n)$ work per access that is "outside" the MPC with small constants. Despite having linear overhead this scheme can perform very well for some n. Another downside of this approach is that it requires interaction between the parties, and therefore is, unlike garbled RAM, incompatible with constant-round garbled circuit protocols.

Most schemes that support a random-access gate require the memory accesses to be sequential. Therefore, if the random-access gate is implemented using secret sharing, this introduces a polylog multiplicative overhead to the round complexity of the protocol. This is always true for [DS17] due to its inherent interactiveness. Garbled RAM schemes, such as [HKO23], are non-interactive due to their use of garbled circuits as opposed to secret sharing. However, this comes at a cost as garbled circuits add an additional security parameter overhead to the communication, compared to some secret-sharing based schemes. One could consider combining parallel garbled RAM [BCP16,LO17] techniques with secret sharing. These techniques allow for the evaluation of $t = O(n)$ RAM gates in parallel. However, these schemes thus far have only been proposed for garbled circuits and are not as optimized as, for example, [HKO23]. Moreover, they still impose a polylog overhead in communication.

We observe that many of the most important applications [DEs16,AHI+22] [HKI+13,CHI+19,FO20,GRR24,NWI+15,Ost24,TN21,DWA+21,BDG+22] [HZF+22,LWDY24,CFL+24,KLS24,CSSW24,KMPP24] for MPC that require random access can be efficiently implemented in a batched manner. In particular, instead of reading just one memory location at a time, the prior MPC computation generates a set of indices $\sigma \subseteq [n]$ with $|\sigma| = O(n)$, and the batched RAM gate accesses and returns all memory locations $\{m_{\sigma(i)}\}$. In this amortized setting one can hope to avoid the polylog overhead lower bounds inherent to the former schemes. Indeed, when σ is a permutation of size n, our permutation protocols can implement the batched RAM gate with an amortized $O(1)$ overhead.[2]

1.2 Our Contributions

In our work, we design and present a comprehensive suite of highly efficient protocols secure against semi-honest corruption that make it feasible to use permutations in real-world two-party computation applications. In some cases the protocols we present are novel, e.g. $\Pi_{\mathsf{Prf\text{-}Perm}}$,$\Pi_{\mathsf{Pcg\text{-}Perm}}$, $\Pi_{\mathsf{Batched\text{-}RAM\text{-}Read}}$, $\Pi_{\mathsf{Batched\text{-}RAM\text{-}Write}}$, while others, such as $\Pi_{\mathsf{Derand\text{-}Msg}}$, $\Pi_{\mathsf{Derand\text{-}Perm}}$, should be viewed closer to a systemization of knowledge. For these latter protocols, their core idea appears implicitly or explicitly in prior work, e.g. [CGP20,CHI+19]. We, however, find that they have not been clearly stated in the two-party setting, and in some cases we observe other works under-utilize their potential. As such, we believe the systemization of permutation knowledge for the two-party setting is an independent contribution. We begin with our protocols for generating permutation correlations:

- We present two novel protocols for the $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ functionality (see Figure 4), which generates a random permutation correlation. In particular, one party learns a permutation $\pi : [n] \to [n]$ while the other party learns a random list $A \in \mathbb{F}_{2^\ell}^n$. The parties obtain a secret sharing (B, C) of $\pi(A) = (A_{\pi_1}, \dots, A_{\pi_n})$, i.e., $B + C = \pi(A)$. We note that this functionality was implicit to the share translation protocol of Chase et al. [CGP20]. We now present our novel protocols for realizing this functionality:
  - $\Pi_{\mathsf{Prf\text{-}Perm}}$, defined in Figure 6, makes use of recent advances in MPC-friendly PRFs to construct a protocol that requires $O(n\ell)$ OTs (binary OLEs). We implement this protocol and observe that it can permute a list of size $n = 2^{20}, \ell = 128$ in 7 seconds on a single thread, or 2.8 seconds on 4 threads.
  - $\Pi_{\mathsf{Pcg\text{-}Perm}}$, defined in Figure 7, is the first protocol to achieve "sub-linear" communication with the help of pseudorandom correlation generators.

[2] We assume the read memory is of size at least κ bits.

In particular, for a chosen π this protocol can generate a *random permutation correlation* with $O(n \log \ell)$ communication/OTs. We believe this protocol to be ideal when a large amount of data needs to be permuted.

We then present various permutation protocols that make use of random permutation correlations. In some cases similar protocols have previously been proposed, with our contribution being a systematization and generalization of prior knowledge.

- Building on the work of [CGP20], we formalize the notation of random permutation correlation and its related protocols. In particular, we consider the functionality of $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ (see Figure 8) that on input permutation π and list X, outputs shares of $\pi(X)$. The protocol $\mathit{\Pi}_{\mathsf{Derand\text{-}Msg}}$ (see Figure 10), described by [CGP20] and implicitly by others, allows one to first generate a random permutation correlation via $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ and then derandomize the correlation to output a secret sharing of $\pi(X)$. $\mathit{\Pi}_{\mathsf{Derand\text{-}Msg}}$ is information-theoretic and is extremely efficient; it requires one round, $n\ell$ bits of communication and $2n$ $\mathbb{F}_{2^\ell}$ additions.
- Building on the sublinear nature of $\mathit{\Pi}_{\mathsf{Pcg\text{-}Perm}}$, we observe that it is possible to use a random permutation correlation for ℓ-bit strings to derandomize many lists $X, Y, Z, \ldots$ in an "on-demand" manner given that the string length of $X, Y, Z, \ldots$ adds to at most ℓ bits. This allows us to amortize the cost of $\mathit{\Pi}_{\mathsf{Pcg\text{-}Perm}}$, which is useful in some applications, e.g. GraphSC.
- We then describe a standard extension of these protocols, which also allows the input list X to be secret-shared between the parties. The $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ protocol achieves this with virtually no overhead as the party with π can locally permute their shares of X.
- We then generalize $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ to the setting where π is secret-shared as the composition of two random permutations π_1, π_2 such that $\pi = \pi_2 \circ \pi_1$. The efficiency of this protocol is essentially two invocations of $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$. This generalization is common in honest majority setting [HKI+13,CHI+19,AHI+22].
- Given a random permutation correlation for π (possibly secret-shared), we show that it is possible to compute shares of $\pi^{-1}(X)$ with the same efficiency as $\pi(X)$. Moreover, one can generate shares for $\pi^{-1}(X)$ using the correlated randomness for π, and vice versa. While straightforward, we are not aware of this protocol in prior works.
- To facilitate the ability to *programmatically* generate a permutation within an MPC computation, we give a new protocol $\mathit{\Pi}_{\mathsf{A2C}}$ in Figure 14 that allows converting between an additive secret sharing of π, e.g. XOR shares $\pi = \pi_1 \oplus \pi_2$, to a permutation composition sharing π'_1, π'_2 such that $\pi = \pi'_1 \circ \pi'_2$. We also give the protocol $\mathit{\Pi}_{\mathsf{C2A}}$ in Figure 15 for converting composition shares of π into additive shares. The efficiency of both protocols is essentially one invocation of $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ of n strings of length $\ell = \log n$ or the derandomization of a similar-sized correlated randomness.
- We then show that our protocols can be used to implement a class of protocols for sorting, $\mathit{\Pi}_{\mathsf{Partition}}$,$\mathit{\Pi}_{\mathsf{Radix\text{-}Sort}}$,$\mathit{\Pi}_{\mathsf{Quick\text{-}Sort}}$ (see Section 9). In particular,

we show that one can recast most existing honest majority sorting protocols into our two-party setting with the same asymptotic complexity, i.e. sorting in $O(n \log n)$ time and $O(\log n)$ rounds.

Finally, we build novel protocols for more expressive functionalities such as general batched RAM gate using our permutation correlations. Armed with these functionalities, we believe that many applications with complex random access requirements can be efficiently realized.

- We present a class of protocols $\Pi_{\mathsf{Ext\text{-}Ord\text{-}Pad}}, \Pi_{\mathsf{Ext\text{-}Unord}}, \ldots$ referred to as extraction (see Section 8). Instead of specifying an input permutation, these protocols allow the users to input a secret-shared vector $f \in \{0,1\}^n$ and a secret-shared list $X \in \mathbb{F}^n$ and output a secret-shared list $\{X_i \mid f_i = 1\}$. The output can be padded to a fixed length c and can have the same order as the input. We also give methods for inverting these transformations to map the output back to the input. These protocols are linear time and mostly constant round with one exception requiring $O(\log n)$ rounds.
- Lastly, our protocols $\Pi_{\mathsf{Batched\text{-}RAM\text{-}Read}}, \Pi_{\mathsf{Batched\text{-}RAM\text{-}Write}}$ (see Section 10) generalize our permutation protocols to allow the parties to input a sharing of an arbitrary function $\sigma : [n] \to [n]$. In particular, when performing a read operation, multiple output positions may read the same input location. This protocol runs in time $O(n \log n)$ and $O(\log n)$ rounds. Similarly, we also present $\Pi_{\mathsf{Batched\text{-}RAM\text{-}Write}}$ that allows the parties to write to memory in a generalized manner.

2 Related Work

Few solutions for permuting secret-shared values in the two-party setting exist. The oldest is the folklore solution using additive homomorphic encryption (AHE). The core idea is relatively simple, the sender party holds a permutation π while the other party, the receiver, holds an input list $X = (X_1, \ldots, X_n)$. The receiver with X first encrypts each entry X_i with their own AHE key k and sends the ciphertexts to the sender. The sender then permutes the ciphertexts by π, randomizes them, and adds a random mask C_i to each ciphertext. The resulting ciphertexts are then sent back to the receiver who decrypts them to obtain $B_1, \ldots, B_n$. Observe that $C + B = \pi(X)$, which can be viewed as a secret sharing of $\pi(X)$. This basic solution can easily be extended to the setting where X is secret-shared. The Paillier scheme is an example of such an AHE scheme. However, it has some disadvantages. The first being that many MPC protocols work with binary secret sharing or with some other small modulus p. The Paillier scheme has a large modulus, which necessitates converting between these representations and incurs an additional overhead. Moreover, Paillier encryption is relatively slow resulting is poor practical performance [CGP20].

[JVC18] proposed the Gazelle protocol that makes use of lattice-based (LWE) AHE to implement permutations. It follows a similar outline as above along with some additional complexities. In particular, LWE AHE schemes support

SIMD/batching operations where many, e.g. $n = 2048$, plaintext values, modulo some p, can be packed into a single ciphertext. This results in improved efficiency, i.e. an $n\times$ reduction in communication/computation. However, to support permutations, one must then be able to permute values within and between ciphertexts. [JVC18] proposes such techniques. We compare our efficiency to both Paillier and Gazelle in Section 11.

An alternative is to make use of a Benes permutation network [Ben64] and any generic MPC protocol, for example GMW or a garbled circuit. The input list X consists of n strings, each of length ℓ bits. A Benes permutation network is a circuit of depth $\log n$ and consists of $\ell n \log n$ swap gates. Each gate takes two ℓ bit string as inputs and output them in order or in swapped order. The party with π can program each of the swap gates. Each way to program the $\ell n \log n$ switch gates corresponds to exactly one permutation. This approach benefits from not making extensive use of public key cryptography but still requires significant communication.

Recently, Chase et al. proposed a different scheme for permutations [CGP20]. Their scheme makes clever use of a function secret sharing and punctured PRF to generate small permutations of size $T \leq 256$. They show that these can then be combined using a Benes permutation network where the swap gates are replaced with small permutation gates with T inputs. Their overall running time is $O(\kappa n \log n + \ell n \log n / \log T)$. In practice this gives a sizable improvement over the classic Benes network constructions. More recently, [SYB+23] demonstrated that [CGP20] could be made malicious secure with moderate overhead. We leave as future work if their techniques can be applied to our protocols.

Finally, in the honest majority setting there is a very simple and efficient protocol for implementing permutations. For example, for three parties, each pair of two parties can jointly hold a random permutation (π_1, π_2, π_3) such that the overall permutation is $\pi = \pi_3 \circ \pi_2 \circ \pi_1$. When using replicated secret sharing [AFL+16] each pair of two parties also holds a secret sharing of the input X. The pair of parties holding π_1 can locally permute their shares of X by π_1 and then secret share the result. This can be repeated for π_2 and π_3. The result is then a secret sharing of $\pi(X)$. [CHI+19] make use of this well-known construction to implement sorting. Recently, [AHI+22,AKK+23] extended it to the malicious honest majority setting.

3 Preliminaries

3.1 Notation

Let $\{a, b, c\}$ denote the set containing elements a, b, c. Let (a, b, c) denote the vector with elements a, b, c. The ith element of a set or a vector S is denoted as S_i. For integers a, b, let the notation $[a, b]$ denote the ordered set $(a, a+1, \ldots, b)$. Let $[n]$ be shorthand for $[1, n]$. We denote κ and λ as the computational and statistical security parameters, respectively.

3.2 Permutations & Injective Functions

We define a permutation $\pi : [n] \to [n]$ of size n as a bijection between the set $[n]$ and itself, i.e. $\pi(i)$ returns a distinct value for each $i \in [n]$. There are several ways to represent such a function. Typically, we consider π to be a vector from the space $[n]^n$ such that $\pi(i) = \pi_i$. One can also view π as a matrix Π in the space $\{0,1\}^{n \times n}$ where $\Pi_{i,\pi(i)} = 1$ and otherwise is zero. This is referred to as a permutation matrix and is convenient in the context of linear algebra. In particular, for a vector $X \in \mathbb{F}^n$, let $\pi(X) := \Pi \cdot X$ denote the vector X permuted by π. Equivalently, $\pi(X) = (X_{\pi(1)}, \ldots, X_{\pi(n)})$.

The set of permutations forms a group under function composition. That is, for permutations $\pi, \rho : [n] \to [n]$, the composition $\pi \circ \rho$ is a permutation $\phi : [n] \to [n]$ such that $\phi(i) = \pi(\rho(i))$. Equivalently, let $\Pi, P, \Phi \in \{0,1\}^{n \times n}$ be the matrix representations of π, ρ, ϕ respectively; then $\Phi = \Pi \cdot P$. A permutation π also has an inverse π^{-1} such that $\pi \circ \pi^{-1} = (1, 2, \ldots, n)$ or equivalently $\Pi \cdot \Pi^{-1} = I$ where I is the identity matrix.

Most of our protocols also work for the case of an injective function $\nu : [n] \to [m]$ for some $m > n$. For ν to be an injection, we require that $\nu(i)$ outputs a distinct value for each i. The vector, matrix representation, and function composition are defined in the same way. However, as ν is injective, there is no inverse function/matrix.

3.3 Secret Sharing & Functionalities

We denote a secret sharing of x as $[\![x]\!]$. For concreteness, we assume binary secret sharing where the two parties respectively hold $[\![x]\!]_1, [\![x]\!]_2 \in \mathbb{F}_{2^\ell}$ such that $x = [\![x]\!]_1 \oplus [\![x]\!]_2$. In some places, our protocols will call for the *integer* addition of several shares. We assume this is achieved by converting to secret sharing over an integer modulus, performing the summation, and switching back to binary sharing, see [MR18] for the relevant techniques.

We define ideal functionalities with the $\mathcal{F}$ notation. One can think of calling $\mathcal{F}$ as sending the inputs to a trusted third party that computes the function and returns the result to the parties. These functionalities will be securely realized by cryptographic protocols, which we denote with the Π notation. A protocol is considered secure if there exists a simulator for it that can only interact with the functionality, see [Lin16] for details.

3.4 Oblivious PRF with Shared Output

We make use of an oblivious weak PRF functionality with secret-shared output, $\mathcal{F}_{\mathsf{Sowprf}}$. The definition of a weak PRF $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ states that for a set of random inputs $x_1, \ldots, x_n \in \mathcal{X}$, and for a random key $k \in \mathcal{K}$, the distribution of $F_k(x_1), \ldots, F_k(x_n)$ should be pseudo-random. We formalize in Definition 1.

Definition 1 (Weak Pseudo-random Function). *A function $f : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a weak pseudo-random function if*

$$\begin{aligned} &\{(x_1, \ldots, x_n, F_k(x_1), \ldots, F_k(x_n)) : x_1, \ldots, x_n \leftarrow \mathcal{X}, k \leftarrow \mathcal{K}\} \\ \approx &\{(x_1, \ldots, x_n, y_1, \ldots, y_n) \qquad\qquad\quad : x_1, \ldots, x_n \leftarrow \mathcal{X}, y_1, \ldots, y_n \leftarrow \mathcal{Y}\}. \end{aligned}$$

The $\mathcal{F}_{\mathsf{Sowprf}}$ functionality evaluates a weak PRF F where the sender inputs a key k while the receiver inputs one or more x. For each x, the parties output a secret sharing $[\![F_k(x)]\!]$. This functionality is meant to model the parties evaluating the PRF within an MPC protocol.

This functionality differs from a traditional OPRF protocol in that the output $F_k(x)$ is secret-shared between the parties instead of revealed in the clear to the receiver. To instantiate this building block we make use of recent advances in so-called MPC-Friendly symmetric key primitives. One of the first popular examples is the LowMC pseudo-random permutation [ARS+15] by Albrecht et al. This block cipher is specifically designed to minimize the number of AND gates in its binary circuit representation. In particular, this block cipher repeatedly performs the following three operations: (1) add the key to the current state, (2) perform a non-linear s-box transformation to the state, (3) multiply the state by a public random invertible matrix. The only step in this process that requires MPC interaction is the s-box transformation, which can be instantiated with approximately 1 AND gate per bit of state. Overall, approximately 14 repetitions need to be applied leading to an MPC evaluations of lowMC requiring approximately 14 AND (28 OTs) gates per output bit of the PRF.

Another prominent class of MPC-Friendly primitives is referred to as alternating moduli first proposed by Boneh et al. [BIP+18] and later optimized in [DGH+21,APRR24]. This weak PRF takes a radically different design. The function can be divided into two phases, one that performs a linear transformation of the state mod 3. The state is then reinterpreted mod 2 and then another linear transformation is applied. The security of this construction relies on the assumption that linear options over two different moduli result in a highly non-linear and unpredictable function. To efficiently implement this in MPC, [APRR24] designs a custom protocol that performs secret sharing over the two different moduli and then performs share conversion between the different moduli. This effectively reduces the number of AND gates per output bit to 2.

In particular, the protocol provided by [APRR24], which implements $\mathcal{F}_{\mathsf{Sowprf}}$, must first perform a key-specific setup protocol. By allowing the key to be reused, we can save on the cost of performing the setup. In this case, the round complexity of the protocol is 2 plus the cost to preprocess $2n\kappa$ OTs where κ is the security parameter. These OTs can be preprocessed and performed in 3 additional rounds. The total communication complexity is $\approx 16n\ell$ bits.

Function Secret Sharing. We will make use of *function secret sharing* (FSS) [BGI15] that allows two parties to generate secret shares k_0, k_1 of a function f in some class $\mathcal{F}$. Given that the two parties respectively hold $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\kappa, f)$, it is possible for them to efficiently and non-interactively generate a secret share

Functionality $\mathcal{F}_{\mathsf{Sowprf}}$:

Parameters: Let $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ be a weak pseudo-random function.

$\mathcal{F}_{\mathsf{Sowprf}}(\textsc{Sender} : k, \textsc{Receiver} : x)$:
Upon input k from the sender and input x from the receiver, return a secret sharing $[\![F_k(x)]\!]$ to the parties.

Fig. 1. The $\mathcal{F}_{\mathsf{Sowprf}}$ functionality that returns secret shares of a weak PRF with a key from the sender and input from the receiver.

$[\![f(x)]\!] := (\mathsf{Eval}(k_0, x), \mathsf{Eval}(k_1, x))$ for any public x. The security guarantee is that given either k_0 or k_1, nothing about the function f is revealed, apart from it being a member of $\mathcal{F}$. Formally,

Definition 2 (Function Secret Sharing). *Let $\mathcal{F}$ be a class of functions. For all $f \in \mathcal{F}$ and $p \in \{0, 1\}$, a FSS scheme $(\mathsf{Gen}, \mathsf{Eval})$ is private if there exists a PPT simulator sim s.t.:*

$$\{(k_p, \mathcal{F}) : (k_0, k_1) \leftarrow \mathsf{Gen}(1^\kappa, f, \mathcal{F})\} \approx \{(k_p, \mathcal{F}) : k_p \qquad \leftarrow \mathsf{Sim}(p, 1^\kappa, \mathcal{F})\},$$

and for all x and all $(k_0, k_1) \in \mathsf{Gen}(1^\kappa, \mathsf{f}, \mathcal{F}) : f(x) = \mathsf{Eval}(k_0, x) + \mathsf{Eval}(k_1, x)$.

A trivial solution would be to simply define k_0, k_1 to be the secret sharing of the truth table of f. However, we desire k_0, k_1 be small, i.e. sublinear in the truth table size. The most prominent example of this technique has been FSS for point functions, a function $f : [n] \to \{0, 1\}^\ell$ that is zero for all input but a for which $f(a) = b$. [GI14,BGI15] describe an efficient scheme based on a length-doubling PRG. Their construction has each party expand a binary tree where the children of a node are defined as the outputs of the length-doubling PRG. The leaves of the tree are indexed by the inputs to f. I.e., at leaf a, the random leaf values differ by b, while all other leaves hold the same random values. Each key k_0, k_1 consists of a single κ-bit element per level of the tree that assists in its generation. That is, the size of k_0, k_1 is $O(\kappa \log_2(n) + \ell)$. In the event that $\ell = 1$, it is possible to reformulate the problem as $n' = n/\kappa$ and $\ell' = \kappa$ by defining b as a unit vector of length κ. Overall, the same function is computed but at a cost $O(\kappa \log_2(n/\kappa) + \kappa)$. Doerner and Shelat [DS17] later gave an efficient two-party key generation protocol that allows the parties to input secret shares of a, b and securely generate k_0, k_1. Their protocol requires $O(\log n)$ rounds and makes only black box calls to the PRG, i.e. does not evaluate it in an MPC circuit.

Syndrome Decoding. The syndrome decoding assumption with regular noise states that for a class of matrices $\mathcal{C}$, one can sample a weight $t = O(\kappa)$ vector $e \in \mathbb{G}^n$ that when compressed by a matrix $A \in \mathcal{C}$, results in a shorter pseudo-random vector $r := Ae$. Syndrome decoding can be shown to be equivalent to the Learning Parity with Noise assumption, where e is the error vector. For reasons of efficiency, we restrict the noise distribution of e to be the concatenation of t random unit vectors. More formally, the Regular Syndrome Decoding assumption is stated in Definition 3.

Functionality $\mathcal{F}_{\mathsf{FSS\text{-}Gen}}$:

Parameters: Let $(\mathsf{Gen}, \mathsf{Eval})$ be a function secret sharing scheme for point functions $f_{\alpha,\beta}$ s.t. $f(\alpha) = \beta$ and otherwise $f(*) = 0$.

$\mathcal{F}_{\mathsf{FSS\text{-}Gen}}([\![\alpha]\!], [\![\beta]\!])$:
Upon input $[\![\alpha]\!], [\![\beta]\!]$, compute $(k_1, k_2) \leftarrow \mathsf{Gen}(f_{\alpha,\beta})$. Return k_1 to the first party and k_2 to the second party.

Fig. 2. Function Secret Sharing key generation protocol $\mathcal{F}_{\mathsf{FSS\text{-}Gen}}$ for point functions.

Definition 3 (Regular Syndrome Decoding [BCG⁺19a]). *For some ring $\mathbb{G}$, let $\mathcal{R}$ denote the uniform distribution over $\mathbb{G}^n$ s.t. the all samples are the concatenation of t regular-sized unit vectors. Let C be a probabilistic code generation algorithm such that $\mathsf{C} \to \mathbb{G}^{k\times n}$. For weight $t = t(\kappa)$, dimension $k = k(\kappa)$, number of samples (or block length) $n = n(\kappa)$, and ring $\mathbb{G} = \mathbb{G}(\kappa)$, the regular syndrome decoding assumption $(\mathcal{R}, \mathsf{C}, \mathbb{G})$-RSD states that*

$$\{(A, \vec{b}) : A \leftarrow \mathsf{C}, \vec{e} \leftarrow \mathcal{R}, \vec{b} := A \cdot \vec{e}\} \approx \{(A, \vec{b}) : A \leftarrow \mathsf{C}, \qquad \vec{b} \leftarrow \mathbb{G}^n\}.$$

We refer to [BCG⁺19a,BCG⁺22,RRT23] for parameter selection.

Aggregation Trees. [BDG⁺22] presents a useful functionality called an aggregation tree $\mathcal{F}_{\mathsf{Agg}}$. This functionality takes as input a shared list X and a shared bit vector B. The list is logically divided up into blocks with the start of a block being denoted by $B_i = 0$. For each block, the functionality will independently apply a prefix sum to the block for some associative sum operator $\star$. For example, if a block begins at i and is of size 3, then the output X' will contain $X'_i := X_i, X'_{i+1} := X_i \star X_{i+1}, X'_{i+2} := X_i \star X_{i+1} \star X_{i+2}$. The operator $\star$ can be any associative operator. [BDG⁺22] give a protocol for implementing this that takes $O(n \cdot \star_{\mathsf{time}})$ time and $O(\log n \cdot \star_{\mathsf{rounds}})$ rounds, where $\star_{\mathsf{time}}, \star_{\mathsf{rounds}}$ are the time and round complexity of computing the $\star$ circuit. We will make use of a duplication tree where $\star$ is defined as $\star(x_1, x_2) := x_1$. That is, it simply returns the first argument.

Functionality $\mathcal{F}_{\mathsf{Agg}}$:

$\mathcal{F}_{\mathsf{Agg}}([\![X]\!], [\![B]\!], \star)$:
Upon input vector $[\![X]\!] \in \mathbb{D}^n$ and control bits $[\![B]\!] \in \{0,1\}^n$ and associative operator $\star$ from the parties, define $\mathsf{pre\text{-}ind}(i) \in [i]$ to be the maximum value such that $B_{\mathsf{pre\text{-}ind}(i)} = 0$. Output $[\![X']\!] \leftarrow \mathsf{share}(X')$ where $X'_i := \bigstar_{j=\mathsf{pre\text{-}ind}(i)}^{i} X_j$

Fig. 3. Functionality $\mathcal{F}_{\mathsf{Agg}}$ for secret-shared aggregation.

4 Technical Overview

In this section we aim to give a high level overview of how all our protocols work. In Section 5, we give the formal descriptions of our permutation correlation generation protocols. In Section 6, we formally present how one can derandomize permutation correlations. Section 7 extends our protocols to secret-shared permutations. Section 8 and Section 9 show that our efficient permutations lead to improvements in a class of protocols we refer to as extraction and in sorting protocols, respectively. Section 10 gives a formal description of how to construct batched random access read and write operations. Finally, Section 11 reports on the concrete running times of our protocols compared to prior art.

We begin with $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}, \mathcal{F}_{\mathsf{Gen\text{-}Perm}}$, our most foundational functionalities for performing permutations on secret-shared data. The former, $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$, provides the most natural interface. It takes as input a permutation $\pi : [n] \to [n]$ from a *sender* party and a vector $X \in \mathbb{F}^n$ from a *receiver* party. The parties output a secret sharing $[\![\pi(X)]\!] = ([\![X_{\pi(1)}]\!], \ldots, [\![X_{\pi(n)}]\!])$. However, our most primitive protocols do not implement this functionality. Instead, they are designed to output random permutation correlations as specified by $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$, see Section 4.1. These can then be derandomized to realize the $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ functionality.

For most of our protocols we give two running times, which typically differ by a multiplicative security parameter κ. The first we refer to as *in practice*. This one has the additional κ overhead but achieves better concrete performance (at the time of writing this document). The second setting we refer to as *in theory*. The primary cause of the difference is the overhead of generating a bit OT / binary OLE [BCG+23]. We defer a more detailed discussion to Section 4.7.

4.1 Permutation Correlation Generators

The functionality $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$, defined in Figure 4, generates the random correlation

$$(A, B), (C, \pi) \qquad \text{s.t.} \qquad \pi(A) = B + C,$$

where the receiver holds A, B, the sender holds C, π, and $A, B, C \in \mathbb{F}^n$ are uniform vectors subject to the constraint above. [CGP20] implicitly use this correlation in their share translation protocol but do not suggest it is a standalone correlation. Another way to view this correlation is as the permutation equivalent to correlated randomness used by multiplication gates in traditional MPC, i.e. a random OLE correlation $(a, b), (c, d)$ s.t. $d \cdot a = b + c$. Looking forward, we will then use this correlation to realize the $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ functionality (see Figure 8).

MPC-Friendly PRF Permutation. $\Pi_{\mathsf{Prf\text{-}Perm}}$ (Figure 6), our first protocol that implements the $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ functionality, achieves the best performance for typical use cases where the strings to be permuted are of small to moderate sizes. The core building block is MPC-friendly PRF. In particular, we make use of a weak PRF F that allows for efficient evaluation in MPC when one party knows the key k and the other party knows the input x. The output is a secret sharing

Functionality $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}(\textsc{Sender} : \pi, \textsc{Receiver})$:

Public parameters: Permutation size n, group $\mathbb{F}$ and string length ℓ.
Input: The sender party inputs a permutation $\pi : [n] \to [n]$.
Output: The functionality samples uniformly random $A, B, C \in \mathbb{F}^{n\times\ell}$ s.t. $\pi(A) = B + C$. (A, B) are output to the receiver, C is output to the sender.

Fig. 4. The ideal functionality $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ for generating random permutation correlations.

of $F_k(x)$, where the secret sharing is over the same group as the permutation correlation (see the functionality $\mathcal{F}_{\mathsf{Sowprf}}$ in Figure 1).

The protocol can be described as follows. The receiver first samples a key k for the weak PRF F and defines their A vector as the output of the PRF on the identity permutation. I.e., $A_i = F_k(i)$ for $i \in [n]$. The parties then engage in n PRF evaluations, where the $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ receiver acts as the $\mathcal{F}_{\mathsf{Sowprf}}$ sender with key k. The $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ sender, which acts as the $\mathcal{F}_{\mathsf{Sowprf}}$ receiver, provides $\pi(i)$ as his input to the ith evaluation. $\mathcal{F}_{\mathsf{Sowprf}}$ outputs a secret sharing of $T_i := F_k(\pi(i))$.

The first observation is that $T_i = A_{\pi(i)}$, and therefore $T = \pi(A)$. In other words, T is A permuted by π. Therefore, what remains is to take the secret sharing of T and generate the B, C vectors. However, we make the observation that this step comes for free in the case that the secret sharing output by $\mathcal{F}_{\mathsf{Sowprf}}$ uses the same group as the permutation correlation. In this case, the receiver and the sender define B_i, C_i as their shares of T_i, respectively. Observe that this is precisely what we want,

$$\begin{aligned} A_{\pi(i)} &= T_i = B_i + C_i \\ \pi(A) &= B + C. \end{aligned}$$

If the sharing of T uses a different group than the permutation correlation, one can use standard techniques to convert the shares.

One deficit of this description is that the input to the weak PRF are the integers between 1 and n. However, weak PRFs require the input values to be sampled *uniformly at random*. We resolve this issue by first running the input $\pi(i)$ through a random oracle H. In particular, the receiver will define $A_i := F_k(H(i))$ while the sender will input $H(\pi(i))$ to $\mathcal{F}_{\mathsf{Sowprf}}$. In this way, the simulator will be able to program H to output the input values associated with the weak PRF challenge.

As an additional optimization, instead of resampling the key k each time $\Pi_{\mathsf{Prf\text{-}Perm}}$ is invoked, it is possible to reuse it and instead sample a new random oracle H. This will ensure that the inputs to the PRF will remain unique and random while amortizing any key-specific cost of $\mathcal{F}_{\mathsf{Sowprf}}$. Overall, the running time is $O(n\ell'\kappa)$ in practice or $O(n\ell')$ in theory, where $\ell' := \lceil \ell/\kappa \rceil \kappa$ is the bit length of the element ℓ rounded up to κ. The communication complexity is $\sim 5n\ell'$ bits [APRR24].

PCG Permutation. As an alternative to our PRF-based construction, we show that one can build a permutation correlation generator $\Pi_{\mathsf{Pcg\text{-}Perm}}$ (Figure 7) with communication that is *sublinear* in the string length ℓ. Specifically, for a permutation π of size n, with correlations $A, B, C \in \mathbb{F}_2^{n\times\ell}$, we give a protocol with $O(n\kappa^2 \log \ell)$ bits of communication. At the heart of this construction is the ability to use LPN/syndrome decoding to get a succinct PRG seed [BCG+19b,BCG+19a]. This seed can then be non-interactively expanded in the MPC context. At a high level, the construction works by first permuting n seeds, $s_1, ..., s_n$ by the permutation π, with the result $\pi(s)$ secret-shared. Once permuted, the parties can non-interactively expand these secret-shared seeds to get secret shares of $\mathsf{PRG}(s_{\pi(i)})$. As before, these shares form the C, B components of the correlation.

In more detail, let P_1 hold permutation π. Let P_2 sample a key k for the wPRF F and define A as follows. Let $(p_{i,1}, ..., p_{i,t}) := F_k(H(i))$ where $p_{i,j} \in [\log_2(\ell)+1]$ for $j \in [t]$ denote the t noisy positions of the error vector $\vec{e}_i$. I.e., let $\vec{e}_i \in \{0,1\}^{2\ell}$ be the syndrome decoding weight-t vector such that $e_{i,p_{i,j}} = 1$ for all $j \in [t]$. Let $G \in \{0,1\}^{\ell\times 2\ell}$ be a matrix such that syndrome decoding is hard. Then define $A_i := G\vec{e}_i$. Without knowledge of $\vec{e}$, A will look pseudorandom by the LPN assumption. Using MPC, the parties compute $[\![p'_{i,1}, ..., p'_{i,t}]\!] := F_k(H(\pi(i)))$ for each $i \in [n]$, where $H(\pi(i)$ is input by P_1 and k is input by P_2. For $j \in [t]$, use a distributed point function key generation protocol (see $\mathcal{F}_{\mathsf{FSS\text{-}Gen}}$) to generate keys $K_{i,j,1}, K_{i,j,2}$ for point $p'_{i,j}$, where P_1 learns the former and P_2 learns the latter. The parties each expand their key to get shares $[\![\vec{e}'_{i,j}]\!]$. The parties compute $[\![A'_i]\!] := G[\![\vec{e}'_i]\!]$. Define B, C as the shares of $[\![A']\!]$.

The running time of this protocol consists of generating nt distributed point functions with bit length σ, where $\sigma = \log(\ell/t\kappa), t = O(\kappa)$. The inputs to the point functions can be generated with $nt\sigma/\kappa = O(n\sigma)$ invocations of $\mathcal{F}_{\mathsf{Sowprf}}$, where each costs $O(\kappa)$ in theory or $O(\kappa^2)$ in practice. The key generation for each distributed point function requires $O(\kappa\sigma + \ell/t)$ work. Finally, we can instantiate G as a linear time code, or more commonly, an $O(\ell \log \ell)$ time code. Overall, the running time is $O(n\kappa^2 \log(\ell/\kappa^2) + n\ell)$ and $O(n\kappa^2 \log(\ell/\kappa^2))$ bits of communication.

Generalized Mapping Functions. We note that these protocols do not require π to be a permutation. The $\mathit{\Pi}_{\mathsf{Prf\text{-}Perm}}, \mathit{\Pi}_{\mathsf{Pcg\text{-}Perm}}$ naturally support an arbitrary function $\pi : [m] \to [n]$. However, when π is not a permutation, the function over $([n] \to [m])$ no longer forms a group. This can lead to certain limitations for secret sharing the permutation itself as we will see later.

4.2 Derandomization

In Section 6, we define the $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ functionality, which allows the user to specify the permutation along with the input vector X to be permuted. We then present the $\mathit{\Pi}_{\mathsf{Basic\text{-}Perm}}$ protocol in the $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$-hybrid model, which, given a random permutation correlation, can derandomize it to give the secret shares of $\pi(X)$ as output. More generally, we present several derandomization techniques that allow the parties to derandomize π, and then generate shares of $\pi(X)$ and/or $\pi^{-1}(X)$ from a single random permutation correlation in an on-demand manner.

$\Pi_{\text{Derand-Perm}}$: **Choosing π.** We begin with the standard technique for derandomizing the permutation of a random permutation correlation (ρ, A, B, C). In particular, the sender holds a random $\rho : [n] \to [n]$ and C, while the receiver holds A, B such that $B+C = \rho(A)$. The sender party with π computes $\delta := \pi^{-1} \circ \rho$ and sends it to the receiver who computes $A' := \delta(A)$. Observe that $A = \rho^{-1}(B+C)$, and therefore

$$\begin{aligned} A' &= \delta(A) \\ &= (\pi^{-1} \circ \rho)(\rho^{-1}(B+C)) \\ \pi(A') &= B + C. \end{aligned}$$

As can be seen by the $\mathcal{F}_{\text{Gen-Perm}}$ functionality, it is also possible to directly generate a permutation correlation for the desired permutation π. However, in some cases it is desirable to be able to generate the correlation before π is known, such as during a preprocessing phase or in a "just in time" manner a few rounds before π is determined. This allows the parties to not have to pay for the round complexity associated with the protocol implementing $\mathcal{F}_{\text{Gen-Perm}}$, since it can be performed concurrently with other useful work. Moreover, this derandomization is extremely efficient, requiring $O(n)$ time and sending $n \log n$ bits in a single message from the sender to the receiver.

$\Pi_{\text{Derand-Msg}}$: **Sharing $\pi(X)$.** Now, assume we already hold a permutation correlation (π, A, B, C) for the desired permutation π such that $B+C = \pi(A)$. The receiver party with (X, A, B) sends $\Delta := X - A$ and outputs B. The other party outputs $C' := C + \pi(\Delta)$. Observe that

$$\begin{aligned} B + C' &= C + \pi(\Delta) + B \\ &= C + \pi(X - A) + B \\ &= C + \pi(X - \pi^{-1}(B + C)) + B \\ &= \pi(X). \end{aligned}$$

This protocol is extremely efficient and requires applying the permutation π to the input Δ, performing $2n$ additions, and sending a single message of the same size as X. This derandomization protocol was used by [CGP20] and is relatively standard.

We propose a natural extension to this protocol where the $A, B, C \in \mathbb{F}^{n\times\ell}$ correlations are larger than $X \in \mathbb{F}^{n\times 1}$. One can simply use the first column of the correlation to mask X, saving the rest of the correlation for later use on some other input X'. This simple observation can be extremely useful in applications, where we wish to permute by π several times, such as in the GraphSC framework [NWI+15] and its many follow-up works.

$\Pi_{\text{Derand-Inv-Msg}}$: **Sharing $\pi^{-1}(X)$.** Our last derandomization technique allows us to also permute by the inverse permutation. Let (π, A, B, C) be the correlated randomness such that $B+C = \pi(A)$. The party with (X, A, B) sends $\Delta := X - B$ and outputs $B = A$. The other party computes and outputs $C' := \pi^{-1}(\Delta - C)$. Observe that

$$\begin{aligned} B + C' &= A + \pi^{-1}(\Delta - C) \\ &= A + \pi^{-1}(X - \pi(A)) \\ &= \pi^{-1}(X). \end{aligned}$$

To the best of our knowledge, we are the first to observe that one can use the same correlation for π to permute by both π and π^{-1}. As in $\Pi_{\mathsf{Derand\text{-}Msg}}$, we can use a *single* correlation to permute and unpermute data in an on-demand manner, saving part of the correlation for later. When combined with our sublinear correlation generator $\Pi_{\mathsf{Pcg\text{-}Perm}}$, this results in high performance if in need to permute by π, π^{-1} multiple times. A prime example is the aforementioned GraphSC framework [NWI+15], where for many iterations, the algorithm permutes by π and then π^{-1}.

Secret-shared Input X. It is trivial to extend $\Pi_{\mathsf{Derand\text{-}Msg}}$ and $\Pi_{\mathsf{Derand\text{-}Inv\text{-}Msg}}$ to the setting where the X to be permuted is secret-shared over the same group as the random permutation correlation. In particular, the party with π can simply permute their own share and add the result to the output share that was obtained previously in $\Pi_{\mathsf{Derand\text{-}Msg}}$,$\Pi_{\mathsf{Derand\text{-}Inv\text{-}Msg}}$.

4.3 Secret-shared Permutations

Composed Permutations. We now present the relatively standard technique for allowing π to be secret-shared. We denote a composed sharing of π as $\langle\!\langle \pi \rangle\!\rangle$ and it consists of two permutations $\langle\!\langle \pi \rangle\!\rangle_1, \langle\!\langle \pi \rangle\!\rangle_2$ such that $\pi = \langle\!\langle \pi \rangle\!\rangle_2 \circ \langle\!\langle \pi \rangle\!\rangle_1$. Each party holds exactly one of these permutations. The parties can then generate a secret sharing of $Z = \pi(X)$ by first invoking $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ on $[\![X]\!]$ to obtain $[\![Y]\!] = \langle\!\langle \pi \rangle\!\rangle_1([\![X]\!])$ and then invoking $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ again to compute $[\![Z]\!] = \langle\!\langle \pi \rangle\!\rangle_2([\![Y]\!])$. The cost of this protocol is two invocations of $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ and local additions.

This protocol inherits all of the extensions we present to the core protocols for $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$. In particular, one can first generate the correlations for $\langle\!\langle \pi \rangle\!\rangle_1, \langle\!\langle \pi \rangle\!\rangle_2$ and then derandomize them on demand. Similarly, one can apply the invert π^{-1} to a secret-shared input by first applying $\langle\!\langle \pi \rangle\!\rangle_2^{-1}$ and then $\langle\!\langle \pi \rangle\!\rangle_1^{-1}$.

Additive Permutations. Finally, we present a new protocol for efficiently converting the representation of π to and from a composition format $\langle\!\langle \pi \rangle\!\rangle$ and an additive secret sharing format $[\![\pi]\!]$ where each party holds a share $[\![\pi]\!]_1, [\![\pi]\!]_2 \in \mathbb{Z}_n^n$ such that $\pi = [\![\pi]\!]_1 + [\![\pi]\!]_2$. The utility of π being additively shared is that π can be generated programmatically within a circuit, e.g. as done by radix sorting $\Pi_{\mathsf{Radix\text{-}Sort}}$. Conversely, it is possible to convert a composed permutation into an additive sharing before using it as input to a circuit computation.

If π is additively shared, we generate a composed permutation sharing by having the first party sample $\langle\!\langle\pi\rangle\!\rangle_1$ at random. We then permute the shares of $[\![\pi]\!]$ by $\langle\!\langle\pi\rangle\!\rangle_1$ to get shares of $\langle\!\langle\pi\rangle\!\rangle_2$ which is revealed to the second party. When compared to the natural extension of [CHI+19] to the two-party setting, our construction requires permuting π only once as opposed to twice. Therefore, our protocol is twice as efficient. Additionally, instead of directly converting $[\![\pi]\!]$ into $\langle\!\langle\pi\rangle\!\rangle$, [CHI+19] converts $[\![\pi]\!]$ into the pair $(\rho, \langle\!\langle\pi'\rangle\!\rangle)$ such that ρ is public and $\pi = \rho \circ \pi'$. As such, [CHI+19] requires the parties to perform additional plaintext permutation by ρ.

Generalized Mapping Functions. The protocols in this section can be made to support any injective function $\pi : [m] \rightarrow [n]$, i.e. $\pi(x)$ does not duplicate any of its inputs. The main alteration is to define $\langle\!\langle\pi\rangle\!\rangle_2 : [m] \rightarrow [n]$ as injective, while $\langle\!\langle\pi\rangle\!\rangle_1$ remains a bijective permutation. The conversion protocols $\Pi_{\mathsf{A2C}}, \Pi_{\mathsf{C2A}}$ can then be defined for injective functions in the natural way.

Composition. We can also achieve composition of permutations. If one permutation is public, then the parties can simply locally permute their shares. Given two secret-shared permutations $\langle\!\langle\pi\rangle\!\rangle, [\![\rho]\!]$, one can compute the composition $\theta = \pi \circ \rho$ by invoking $\mathcal{F}_{\mathsf{Comp\text{-}Perm}}$ on input permutation $\langle\!\langle\pi\rangle\!\rangle$ and list $[\![\rho]\!]$.

4.4 Extraction

Now that we introduced our permutation protocols, we present a class of protocols called extractions (formally defined in Section 8). In these protocols, the parties input a secret-shared bitvector $f \in \{0,1\}^n$ and return a permutation that if applied to X, returns the subset $\{X_i \mid f_i = 1\}$.

We further refine this basic idea in two ways:

- **Ordering.** The first concerns the *order* of elements in the output. $\Pi_{\mathsf{Ext\text{-}Unord}}$ outputs X_i (s.t. $f_i = 1$) in random order; $\Pi_{\mathsf{Ext\text{-}Ord}}$ outputs X_i in the original order within X.
- **Padding.** The second concerns the *size* of the output. The output can either contain only the X_i s.t. $f_i = 1$ ($\Pi_{\mathsf{Ext\text{-}Unord}}$, $\Pi_{\mathsf{Ext\text{-}Ord}}$) or it can be padded to a fixed length ($\Pi_{\mathsf{Ext\text{-}Unord\text{-}Pad}}$, $\Pi_{\mathsf{Ext\text{-}Ord\text{-}Pad}}$). In this case, the protocols receive one additional input c, which specifies the output size. This is especially useful when the parties do not know the size $|\{X_i \mid f_i = 1\}|$. They can determine the upper bound on the output size to ensure obliviousness in secure computation.

All of these protocols are based on the same fundamental idea that if we permute f according to a random permutation π (unknown to parties), then it is secure to reveal $f' = \pi(f)$, or a padded version of $\pi(f)$, in the clear. This is because now we cannot correlate any f'_i with f_j. Once f' is revealed we simply compose the (injective) permutation that returns all $[\![X_i]\!]$ such that $f'_i = 1$

with π. This is exactly how $\Pi_{\textsf{Ext-Unord}}$ is implemented. The remaining extraction protocols use $\Pi_{\textsf{Ext-Unord}}$ as a subprotocol, and hence are derived from the same idea.

To implement $\Pi_{\textsf{Ext-Unord-Pad}}$ (unordered extraction with padding), we pad f with c elements. We mark a subset of these elements so that the total number of marked elements (in X and the c appended elements) is c. Once padded, we use $\Pi_{\textsf{Ext-Unord}}$ to extract them.

To implement $\Pi_{\textsf{Ext-Ord}}$ (i.e. output the marked elements in their original order), we additionally mark each X_i with the number of elements with $f_{j<i} = 1$. This mark stores the original order of X_i with $f_i = 1$ and can be computed locally via simple additions. Then we invoke $\Pi_{\textsf{Ext-Unord}}$ to extract all the X_i with $f_i = 1$. Recall the output is in random order as $\Pi_{\textsf{Ext-Unord}}$ uses a random permutation. We place the extracted elements in their original order by opening the additional mark and ordering the elements based on this mark.

$\Pi_{\textsf{Ext-Ord-Pad}}$ combines the ideas from $\Pi_{\textsf{Ext-Ord}}$ and $\Pi_{\textsf{Ext-Unord-Pad}}$. The main difference is that we need to do some extra accounting to ensure the padded elements are placed at the end of the output list after calling $\Pi_{\textsf{Ext-Unord}}$.

4.5 Sorting

Additionally, we show that many of the existing ideas for implementing sorting in the honest-majority three-party setting [AHI+22,CHI+19,HKI+13] directly translate to our two-party framework (formally presented in Section 9). As with our previous protocols, the output of these functionalities is a secret-shared permutation $\langle\!\langle \pi \rangle\!\rangle$ that when applied to the input would sort it. $\mathcal{F}_{\textsf{Stable-Sort}}$ defines the ideal functionality.

We begin with our $\Pi_{\textsf{Partition}}$ protocol that can be viewed as a sorting protocol for single-bit elements. This protocol takes inspiration from [CHI+19] and makes the following observation. Let $[\![X]\!]$ be the input vector. One can generate the inverse sorting permutation using three transformations $x \to f \to c \to s \to \pi$. For example, if $X = (0, 1, 1, 0, 1)$ then:

$$f = \begin{bmatrix} 1\,0\,0\,1\,0 \\ 0\,1\,1\,0\,1 \end{bmatrix}, \quad c = \begin{bmatrix} 1\,1\,1\,2\,2 \\ 2\,3\,4\,4\,5 \end{bmatrix}, \quad s = \begin{bmatrix} 1\,0\,0\,2\,0 \\ 0\,3\,4\,0\,5 \end{bmatrix}, \quad \pi = (1, 3, 4, 2, 5)$$

We construct a $2 \times n$ matrix f, where the first row is $\neg X$ and the second is X. We obtain c by applying a prefix sum over each element of the first row followed by the second. Then take the component-wise mutiplication between f and c to obtain s. Observe that if one sums the two row vectors, we obtain a permutation $\pi = (1, 3, 4, 2, 5)$. This is precisely the inverse permutation that sorts X. The $\Pi_{\textsf{Partition}}$ protocol concludes by converting $[\![\pi^{-1}]\!]$ into $\langle\!\langle \pi^{-1} \rangle\!\rangle$ using $\mathcal{F}_{\textsf{A2C}}$. A second benefit of this approach is that the sorting permutation is stable, meaning that items with equal values maintain the same order after the sort.

We make the observation that many advanced protocols make use of similar techniques. I.e., they have a specialized protocol that allows each share to compute the index of the position to which it should be mapped, i.e. the inverse permutation.

This protocol can be extended to sort inputs with b bits, where $b = O(1)$, by defining f as having 2^b rows, where the ith column has a 1 in the position X_i. Overall, the time of this protocol is dominated by the cost of generating the permutation correlation, i.e. $O(n\kappa)$ time and bits of communication, assuming $\mathcal{F}_{\mathsf{Sowprf}}$ is computed in $O(\kappa)$ time [APRR24], otherwise $O(n\kappa^2)$ time.

Additionally, one can extend $\Pi_{\mathsf{Partition}}$ to implement our radix sort protocol Π_{Sort} for arbitrary bit length items. The idea is to call $\Pi_{\mathsf{Partition}}$ for each bit position and compose the resulting permutations. I.e., first sort the most significant bit, then the next most significant bit, and on down to the least significant bit. After each $\Pi_{\mathsf{Partition}}$, one can compose the permutations, apply the result to the next bit, and recurse. As with $\Pi_{\mathsf{Partition}}$, this radix sort returns a stable sorting permutation. Overall, the protocol requires $O(n\ell\kappa)$ time and communication, assuming $\mathcal{F}_{\mathsf{Sowprf}}$ is computed in $O(\kappa)$ time [APRR24], otherwise $O(n\ell\kappa^2)$ time.

We also present the quick-sort protocol $\Pi_{\mathsf{Quick\text{-}Sort}}$ using the so-called shuffle-and-reveal model [HKI+13]. The idea is that one first applies a random secret-shared permutation to the input X. Now, for most comparison-based sorting algorithms, we can show that the results of the comparisons are input-independent. Given this, it is secure to simply run a plaintext sorting algorithm, where each comparison is replaced with a protocol that compares the secret-shared inputs and reveals the result to the parties. Assuming one implements comparison using a parallel prefix adder circuit, the running time and communication of this protocol is $O(n \log n \log\log n + n\kappa)$, assuming $\mathcal{F}_{\mathsf{Sowprf}}$ is computed in $O(\kappa)$ time [APRR24], otherwise $O(n\kappa \log n \log\log n)$ time. The round complexity is $O(\log n \log\log n)$. One could also use a ripple adder to obtain $O(\log^2 n)$ rounds and less computation.

4.6 Batched Random Access

Finally, we present the novel $\Pi_{\mathsf{Batched\text{-}RAM\text{-}Read}}$ protocol, which can be viewed as a generalization of our permutation protocols and is defined formally in Section 10. In particular, the parties input a list X of length n and an additive secret sharing of an arbitrary function $\sigma : [m] \to [n]$. The output is a secret sharing of Y such that $Y_i := X_{\sigma(i)}$. This differs from our permutation functionality in that X_j could be mapped to *multiple* positions in Y. To support this functionality the protocol makes use of more sophisticated techniques such as sorting and aggregation trees. We will model these using the $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ and $\mathcal{F}_{\mathsf{Agg}}$ functionalities that we presented in Section 3. The protocol works by first calling stable sort $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ on $\sigma = (\sigma(1), ..., \sigma(n))$. If σ contains duplicates, they now each form a group. We mark the first element of each group and map the corresponding element of X to the marked position. All other positions are given a dummy value. The parties invoke $\mathcal{F}_{\mathsf{Agg}}$ on this vector with the markings as the aggregation tree control bits, resulting in the mapped elements duplicated across its group. The final result is obtained by permuting vector by the inverse of the sorting permutation for σ. The running time of this protocol is dominated by the cost of sorting. If implemented with the radix sort protocol $\Pi_{\mathsf{Radix\text{-}Sort}}$, the overhead is $O(n \log n\kappa)$ time and bits of communication.

Additionally, we present the novel $\Pi_{\mathsf{Batched\text{-}RAM\text{-}Write}}$ protocol, which uses similar techniques, but allows to perform write operations into a vector X. Each position of X can optionally be written to by multiple updated values. If more than one value is written to a single position, they are combined using a customizable associative operator. For example, maybe they are added together or the smallest value is taken.

4.7 Computational Overheads

Figure 5 summarizes the overheads of our protocols. We present two sets of asymptotic running times. The first is referred to as *Time (theory)* and corresponds to the running time when one can compute $\mathcal{F}_{\mathsf{Sowprf}}$ in $O(\kappa)$ time and communication. While not practical as of the time of writing this work, we note that [APRR24] combined with [BCG$^+$23] gives a theoretical construction that can achieve the desired overhead. However, it is more practical to assume $\mathcal{F}_{\mathsf{Sowprf}}$ requires $O(\kappa^2)$ time and $O(\kappa)$ communication. We denote this setting as *Time (practice)*. More generally, the overhead of most of our constructions is proportional to the overhead of invoking $\mathcal{F}_{\mathsf{Sowprf}}$ on an input size that is proportional to the input being permuted or otherwise manipulated. For radix sort, the primary overhead is a permutation for each bit of the key, while quick sort requires one permutation and $n \log n$ comparisons. Finally, the overhead of our batched RAM protocols is essentially proportional to sorting the selection vector and two permutations of the data.

5 Permutation Correlation Generators

We formally define the ideal functionality $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ in Figure 4. It is parameterized by public inputs $n, \ell, \mathbb{F}$, which are the permutation size, the correlation string length, and the group of the correlation, respectively. The functionality takes as input a permutation and outputs uniform $A, C, D \in \mathbb{F}^{n\times\ell}$ such that $\pi(A) = C + D$. We give two implementations of this functionality, $\Pi_{\mathsf{Prf\text{-}Perm}}$ and $\Pi_{\mathsf{Pcg\text{-}Perm}}$. The former is based on any weak PRF such as [APRR24]. The latter additionally makes use of syndrome decoding.

MPC-Friendly PRF-based Permutation Correlation Generator. We begin with the $\Pi_{\mathsf{Prf\text{-}Perm}}$ protocol and prove that is is secure in the semi-honest setting. We refer to Section 4.1 for the intuition of the protocol.

Theorem 1. *The $\Pi_{\mathsf{Prf\text{-}Perm}}$ protocol of Figure 6 realizes $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ functionality with semi-honest security in the $\mathcal{F}_{\mathsf{Sowprf}}$-hybrid model, assuming F is weak PRF.*

Proof. **Corrupt Sender:** The view of the sender consists of their shares of $[\![y]\!]$, which are uniformly distributed due to the output distribution of $\mathcal{F}_{\mathsf{Sowprf}}$, which evaluates the weak PRF F. Now consider the output distribution of the honest receiver. B is uniform subject to $\pi(A) = B + C$, as required.

Protocol	Time	Time	Comm.	Rounds
	(theory)	(practice)	(bits)	
$\Pi_{\mathsf{Prf\text{-}Perm}}$	$n\ell'$	$n\ell'\kappa$	$n\ell'$	2
$\Pi_{\mathsf{Pcg\text{-}Perm}}$	$n\kappa^2\log(\ell/\kappa^2)+n\ell$	$n\kappa^2\log(\ell/\kappa^2)+n\ell\log\ell$	$n\kappa^2\log(\ell/\kappa^2)+n\ell$	$\log(\ell/\kappa^2)$
$\Pi_{\mathsf{Derand\text{-}Perm}}$	n	n	$n\log n$	1
$\Pi_{\mathsf{Derand\text{-}Msg}}$	$n\ell$	$n\ell$	$n\ell$	1
$\Pi_{\mathsf{Derand\text{-}Inv\text{-}Msg}}$	$n\ell$	$n\ell$	$n\ell$	1
$\Pi_{\mathsf{Basic\text{-}Perm}}$	$n\ell'$	$n\ell'\kappa$	$n\ell'$	1
$\Pi_{\mathsf{Comp\text{-}Perm}}$	$n\ell'$	$n\ell'\kappa$	$n\ell'$	2
Π_{A2C}	$n\kappa$	$n\kappa^2$	$n\kappa$	1
Π_{C2A}	$n\kappa$	$n\kappa^2$	$n\kappa$	1
$\Pi_{\mathsf{Ext\text{-}Unord}}$	$n\kappa$	$n\kappa^2$	$n\kappa$	3
$\Pi_{\mathsf{Ext\text{-}Ord}}$	$n\kappa$	$n\kappa^2$	$n\kappa$	3
$\Pi_{\mathsf{Ext\text{-}Unord\text{-}Pad}}$	$n\kappa$	$n\kappa^2$	$n\kappa$	$3+\log n$
$\Pi_{\mathsf{Ext\text{-}Ord\text{-}Pad}}$	$n\kappa$	$n\kappa^2$	$n\kappa$	$3+\log n$
$\Pi_{\mathsf{Partition}}$	$n\kappa$	$n\kappa^2$	$n\kappa$	3
$\Pi_{\mathsf{Radix\text{-}Sort}}$	$n\ell\kappa$	$n\ell\kappa^2$	$n\kappa$	3ℓ
$\Pi_{\mathsf{Quick\text{-}Sort}}$	$n\ell^*\log n\cdot\log\ell^*+n\kappa$	$n\ell^*\kappa\log n\cdot\log\ell^*+n\kappa^2$	$n\ell^*\log n\cdot\log\ell^*+n\kappa$	$3+\log n\log\ell^*$
$\Pi_{\mathsf{Batched\text{-}RAM\text{-}Read}}$	$n\log n\log\log n+n\kappa$	$n\kappa\log n\log\log n+n\kappa^2$	$n\log n\log\log n+n\kappa$	$3+\log n$

Fig. 5. Performance metrics of our protocols. n is the input length, ℓ is the element bit length, κ is the security parameter, $\ell' := \lceil \ell/\kappa \rceil\kappa$ is the element bit length rounded up to κ, and $\ell^* := \ell + \log n$. For round complexity, we do not count any rounds that can be preprocessed. (theory) refers to the time required if [APRR24] is implemented in $O(\kappa)$ time and we make use of $O(1)$ time amortized bit OTs [BCG+23]. For the extraction protocols, we omit the cost of permuting the data, which can be done seperately via $\Pi_{\mathsf{Basic\text{-}Perm}}$.

To show that A is uniform, consider the weak PRF game of Definition 1. For sake of a contradiction, let us assume a distinguisher that can distinguish A from uniform. The simulator first queries the weak PRF challenger to obtain the instance $(x_1, ..., x_{nm}, y_1, ..., y_{nm})$ and programs the random oracle H to output x_{in+j} on input (t, i, j). Since t is picked at random, the probability that such a query has previously been made is negligible. Note that the output distribution of H remains uniformly random. Therefore, A is precisely the elements $y_1, ..., y_{nm}$, and therefore our distinguisher can also distinguish the weak PRF game.

Corrupt Receiver: The view of the receiver consists of the random nonce t and the output of $\mathcal{F}_{\mathsf{Sowprf}}$ which is uniformly random. Similarly, the output distribution of the honest sender is uniform subject to the desired correlation. □

PCG-based Permutation Correlation Generator. We now turn our attention to our permutation correlation generator based on the techniques underlying the recent developments on pseudo-random correlation generators [BCG+19b].

Protocol $\Pi_{\mathsf{Prf\text{-}Perm}}(\textsc{Sender} : \pi, \textsc{Receiver})$:

1. Let $F : \mathcal{K} \times \mathcal{X} \to \mathbb{F}^w$ be a weak PRF and $H : \{0,1\}^* \to \mathcal{X}$ be a random oracle.
2. The receiver samples $k \leftarrow \mathcal{K}$ and the sender samples $t \in \{0,1\}^\kappa$. The sender sends t to the receiver.
3. Let $x_{i,j} := H(t, i, j)$ for $i \in [n], j \in [m]$ where $m := \lceil \ell/w \rceil$.
4. The receiver computes $A_{i,j} := F_k(x_{i,j})$ for $i \in [n], j \in [m]$.
5. The parties invoke $\mathcal{F}_{\mathsf{Sowprf}}$ nm times with the receiver inputting k and sender inputting $\pi(x)$. The parties receive shares $[\![y_{i,j}]\!]$ where $y_{i,j} = F_k(x_{\pi(i),j})$ for $i \in [n], j \in [m]$.
6. The sender sets $C_i := [\![y_i]\!]_{\mathsf{s}}$ and the receiver sets $B_i := [\![y_i]\!]_{\mathsf{r}}$.
7. The receiver returns (A, B) and the sender returns C.

Fig. 6. The $\Pi_{\mathsf{Prf\text{-}Perm}}$ protocol that implements the $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ functionality.

Protocol $\Pi_{\mathsf{Pcg\text{-}Perm}}(\textsc{Sender} : \pi, \textsc{Receiver})$:

1. Let F be a weak PRF and H be a random oracle.
2. The receiver samples $k \leftarrow \{0,1\}^\kappa$ and the sender samples $t \in \{0,1\}^\kappa$. The sender sends t to the receiver.
3. Let $x_i := H(t, i)$ for $i \in [n]$.
4. The receiver defines $(p_{i,1}, ..., p_{i,t}) := F_k(x_i)$ where $p_{i,j} \in [\log_2(\ell/t) + 1)$.
5. Let $\vec{e}_{i,j} \in \{0,1\}^{2\ell/t}$ be the unit vector such that $e_{i,j,p_{i,j}} = 1$. Let $\vec{e}_i$ denote the concatenation of $\vec{e}_{i,1}, ..., \vec{e}_{i,t}$.
6. Let $G \in \{0,1\}^{\ell \times 2\ell}$ be a matrix such that syndrome decoding is hard with regular weight t noise.
7. The receiver defines $A_i := G\vec{e}_i$.
8. For each $i \in [n]$, the parties compute $[\![p'_{i,1}, ..., p'_{i,t}]\!] := F_k(H(\pi(i)))$ by invoking $\mathcal{F}_{\mathsf{Sowprf}}$, where $H(\pi(i))$ is input by the sender and k is input by the receiver.
9. For $i \in [n], j \in [t]$, the parties invoke the distributed point function key generation functionality $\mathcal{F}_{\mathsf{FSS\text{-}Gen}}$ on input $([\![p'_{i,j}]\!], [\![1]\!])$ to generate keys $K_{i,j,1}, K_{i,j,2}$, where the sender learns the former and the receiver learns the latter.
10. For $i \in [n], j \in [t]$, the parties each expand their key to get shares $[\![\vec{e}'_{i,j}]\!]$.
11. For $i \in [n]$, the parties compute $[\![\vec{e}'_i]\!] := ([\![\vec{e}'_{i,1}]\!] || ... || [\![\vec{e}'_{i,t}]\!])$.
12. For $i \in [n]$, the parties compute $[\![A'_i]\!] := G[\![\vec{e}'_i]\!]$. Define B, C as the shares of $[\![A']\!]$.
13. The receiver returns (A, B) and the sender returns C.

Fig. 7. The $\Pi_{\mathsf{Pcg\text{-}Perm}}$ protocol that implements the $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ functionality.

Theorem 2. *The $\Pi_{\textsf{Pcg-Perm}}$ protocol of Figure 7 realizes the $\mathcal{F}_{\textsf{Gen-Perm}}$ functionality with semi-honest security in the $\mathcal{F}_{\textsf{Sowprf}}$ and $\mathcal{F}_{\textsf{FSS-Gen}}$-hybrid model, assuming F is a weak PRF and* RSD *is hard.*

Proof. **Corrupt Sender:** The view of the corrupt sender is the output of $\mathcal{F}_{\textsf{Sowprf}}$, which is uniformly random. This is fed as input to $\mathcal{F}_{\textsf{FSS-Gen}}$, which returns $K_{i,j,1}$. By Definition 2, this key can be simulated.

Consider the hybrid model where the weak PRF is replaced by a random function. Following the same argument as in Theorem 1, the ability of the distinguisher to distinguish implies a distinguisher for the weak PRF game of Definition 1.

The output distribution of the honest receiver consists of A, B. Fixing A, then B is fully determined by π, C. For the sake of a contradiction, let us assume the distinguisher can distinguish A (and therefore B) from uniformly random, i.e. the simulator replaces A, B such that A is uniform. Given that each e_i was sampled as specified by Definition 3, this implies that the distinguisher can distinguish $G\vec{e_i}$ from uniformly random. However, this contradicts that RSD is hard.

Corrupt Receiver: The view of the corrupt receiver includes the random nonce t, the output of $\mathcal{F}_{\textsf{Sowprf}}$ and $\mathcal{F}_{\textsf{FSS-Gen}}$. These can all be simulated in a straightforward way. Finally, the output distribution of the honest sender is correct, i.e. C such that $C = B - \pi(A)$. □

6 Derandomization

We are now ready to present our derandomization protocols. The final goal of these protocols is to efficiently implement the $\mathcal{F}_{\textsf{Basic-Perm}}$ functionality in various settings. In particular, our goal is to enable the parties to call $\mathcal{F}_{\textsf{Gen-Perm}}$ either during a preprocessing phase or in the online phase, and then derandomize the correlation to generate a secret sharing of $\pi(X)$ where both $[\![X]\!]$ and π are chosen by the parties. $\mathcal{F}_{\textsf{Basic-Perm}}$ functionality is relatively straightforward. It takes as input a permutation of size n from the sender and a shared input list $X \in \mathbb{F}^{n\times\ell}$; the output is a secret sharing $[\![Y]\!]$ of the rows of X permuted by the permutation π, i.e. $Y = \pi(X)$.

Functionality $\mathcal{F}_{\textsf{Basic-Perm}}([\![X]\!], \textsc{Sender} : \pi)$:

Public parameters: Permutation of size n, group $\mathbb{F}$, and string length ℓ.
Input: The parties input a sharing of $X \in \mathbb{F}^{n\times\ell}$ and the sender party inputs a permutation $\pi : [n] \to [n]$.
Output: The functionality samples uniformly random sharing $[\![Y]\!]$ s.t. $Y = \pi(X)$.

Fig. 8. The $\mathcal{F}_{\textsf{Basic-Perm}}$ functionality.

We begin with our protocol $\Pi_{\textsf{Derand-Perm}}$ in Figure 9 for transforming a random permutation correlation (A, B, C, ρ) into a correlation (A', B, C, π), where π is chosen by the sender. Given that ρ is uniformly distributed in the view of the receiver, the resulting correlation (A', B, C, π) is indistinguishable from the one returned by $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ on input π.

Protocol $\Pi_{\mathsf{Derand\text{-}Perm}}(\textsc{Sender} : \pi, (\rho, C), \textsc{Receiver}(A, B))$:

1. The sender sends $\delta := \pi^{-1} \circ \rho$ to the receiver.
2. The receiver computes $A' := \delta(A)$.
3. The sender outputs (π, C) and the receiver outputs (A', B).

Fig. 9. The $\Pi_{\mathsf{Derand\text{-}Perm}}$ protocol that derandomizes a random permutation correlation to a chosen permutation correlation.

Theorem 3. *The composition of $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ for random ρ and $\Pi_{\mathsf{Derand\text{-}Perm}}$ for arbitrary π securely realizes $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ for input π in the semi-honest setting.*

Proof. Observe that the protocol is correct. Moreover, the only message sent is $\delta = \pi^{-1} \circ \rho$. Given that permutations under composition form a group and that ρ is uniformly distributed, it is straightforward to see the distribution of δ is uniform. □

Protocol $\Pi_{\mathsf{Derand\text{-}Msg}}$ in Figure 10 takes as input a list $X \in \mathbb{F}^{n\times\ell}$ and a permutation correlation (A, B, C, π). The protocol derandomizes the correlation to return secret shares of X permuted by π. Given that (A, B, C, π) is a uniformly distributed permutation correlation, the result is indistinguishable from the parties calling $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ on input π and X.

Protocol $\Pi_{\mathsf{Derand\text{-}Msg}}(\textsc{Sender} : (\pi, C), \textsc{Receiver} : X, (A, B))$:

1. The receiver sends $\Delta := X - A$ to the sender.
2. The sender computes $C' := \pi(\Delta) + C$.
3. The sender outputs C' and the receiver outputs B.

Fig. 10. The $\Pi_{\mathsf{Derand\text{-}Msg}}$ protocol that derandomizes a permutation correlation to generate shares of X permuted by π.

Theorem 4. *The composition of $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ for arbitrary π and $\Pi_{\mathsf{Derand\text{-}Msg}}$ for input $[\![X]\!]$ securely realizes $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ on input $[\![X]\!], \pi$ in the semi-honest setting.*

Proof. The only message sent is $\Delta = X - A$. Given that A is uniformly distributed, so is Δ. □

Protocol $\Pi_{\mathsf{Derand\text{-}Inv\text{-}Msg}}$ in Figure 11 takes as input a list $X \in \mathbb{F}^{n\times\ell}$ and a permutation correlation (A, B, C, π). The protocol derandomizes the correlation to return secret shares of X permuted by π^{-1}. Given that (A, B, C, π) is a uniformly distributed permutation correlation, the result is indistinguishable from the parties calling $\mathcal{F}_{\textsf{Basic-Perm}}$ on input π^{-1} and X.

Protocol $\Pi_{\textsf{Derand-Inv-Msg}}(\textsc{Sender} : (\pi, C), \textsc{Receiver} : X, (A, B))$:

1. The receiver sends $\Delta := X - B$ to the sender.
2. The sender computes $C' := \pi^{-1}(\Delta - C)$.
3. The sender outputs C' and the receiver outputs A.

Fig. 11. The $\Pi_{\textsf{Derand-Inv-Msg}}$ protocol that derandomizes a permutation correlation to generate shares of X permuted by π^{-1}.

Theorem 5. *The composition of $\mathcal{F}_{\textsf{Gen-Perm}}$ for arbitrary π and $\Pi_{\textsf{Derand-Inv-Msg}}$ for input $[\![X]\!]$ securely realizes $\mathcal{F}_{\textsf{Basic-Perm}}$ on input $[\![X]\!]$ and π^{-1} in the semi-honest setting.*

Proof. Similar to Theorem 4. □

Definition 4. *Let $\Pi_{\textsf{Basic-Perm}}(\pi, [\![X]\!])$ be defined as the composition of $\mathcal{F}_{\textsf{Gen-Perm}}(\pi)$ and $\Pi_{\textsf{Derand-Msg}}$.*

7 Secret-Shared Permutations

In functionalities $\mathcal{F}_{\textsf{Comp-Perm}}$ and $\mathcal{F}_{\textsf{Add-Perm}}$, we extend the core $\mathcal{F}_{\textsf{Basic-Perm}}$ permutations functionality to allow the permutation to be secret-shared. The former $\mathcal{F}_{\textsf{Comp-Perm}}$ allows the parties to input a composed secret shared permutation $\langle\!\langle \pi \rangle\!\rangle$ and a secret-shared list $[\![X]\!]$, and then receive secret shares of $\pi(X)$. This is achieved by each party holding a random plaintext permutation, which when composed together, equals π. In particular, we denote the two permutations $\langle\!\langle \pi \rangle\!\rangle_1, \langle\!\langle \pi \rangle\!\rangle_2$ (the first party holds $\langle\!\langle \pi \rangle\!\rangle_1$, the second $\langle\!\langle \pi \rangle\!\rangle_2$). The latter functionality $\mathcal{F}_{\textsf{Add-Perm}}$ achieves the same result, but allows the permutation π to be additively secret-shared, i.e. $\pi = [\![\pi]\!]_1 + [\![\pi]\!]_2$. To facilitate $\mathcal{F}_{\textsf{Add-Perm}}$, we give a conversion protocol $\mathcal{F}_{\textsf{A2C}}$ between $[\![\pi]\!]$ and $\langle\!\langle \pi \rangle\!\rangle$. This conversion is secure conditioned on π being a valid permutation.

We additionally present a relaxation to $\mathcal{F}_{\textsf{A2C}}$ which we call $\mathcal{F}_{\textsf{Par-A2C}}$ that allows $[\![\pi]\!]$ to be a secret sharing of a so called partial permutation. In particular, $\pi \in [n]^m$ is a partial permutation if when interpreted as a function $\pi : [m] \to [n]$ is injective and non-surjective, i.e. $m < n$ and the $\pi(i)$ values are all distinct. We implement $\mathcal{F}_{\textsf{Comp-Perm}}$ with the $\Pi_{\textsf{Comp-Perm}}$ protocol in the $\mathcal{F}_{\textsf{Basic-Perm}}$-hybrid model. We note that one can permute by π^{-1} simply by running the protocol in reverse.

Theorem 6. *Protocol $\Pi_{\textsf{Comp-Perm}}$ securely realizes the $\mathcal{F}_{\textsf{Comp-Perm}}$ functionality in the semi-honest setting.*

Functionality $\mathcal{F}_{\textsf{Comp-Perm}}$ $(\langle\!\langle \pi \rangle\!\rangle, [\![X]\!])$:

INPUT: Composed permutation $\langle\!\langle \pi \rangle\!\rangle$ and secret-shared list $[\![X]\!]$.
OUTPUT: Secret-shared list $[\![Y]\!]$ such that $Y := \pi(X)$.

Protocol $\Pi_{\textsf{Comp-Perm}}(\langle\!\langle \pi \rangle\!\rangle, [\![X]\!])$:

1. The parties invoke $\mathcal{F}_{\textsf{Basic-Perm}}$ on $\langle\!\langle \pi \rangle\!\rangle_1$ and $[\![X]\!]$, and receive $[\![Y]\!]$ as the result.
2. The parties invoke $\mathcal{F}_{\textsf{Basic-Perm}}$ on $\langle\!\langle \pi \rangle\!\rangle_2$ and $[\![Y]\!]$, and receive $[\![Z]\!]$ as the result.
3. The parties output $[\![Z]\!]$.

Fig. 12. The $\mathcal{F}_{\textsf{Comp-Perm}}$ functionality and $\Pi_{\textsf{Comp-Perm}}$ protocol that permute a secret shared list $[\![X]\!]$ by a composed permutation $\langle\!\langle \pi \rangle\!\rangle$.

Proof. The protocol is trivial to simulate given that only $\mathcal{F}_{\textsf{Basic-Perm}}$ is invoked and the parties do not directly send messages to each other. □

Functionality $\mathcal{F}_{\textsf{Add-Perm}}$ $([\![\pi]\!], [\![X]\!])$:

INPUT: Additively shared permutation $[\![\pi]\!]$ and secret-shared list $[\![X]\!]$.
OUTPUT: Secret-shared list $[\![Y]\!]$ such that $Y := \pi(X)$.

Fig. 13. The $\mathcal{F}_{\textsf{Add-Perm}}$ functionality that permutes a secret-shared list $[\![X]\!]$ by an additively shared permutation $[\![\pi]\!]$.

To implement $\mathcal{F}_{\textsf{Add-Perm}}$, we must first convert $[\![\pi]\!]$ into $\langle\!\langle \pi \rangle\!\rangle$. We achieve this with the $\mathcal{F}_{\textsf{A2C}}$ functionality, which is realized by the $\Pi_{\textsf{A2C}}$ protocol. Once the parties hold $\langle\!\langle \pi \rangle\!\rangle$, they can simply invoke $\mathcal{F}_{\textsf{Comp-Perm}}$ to realize the $\mathcal{F}_{\textsf{Add-Perm}}$ functionality. We also present the $\mathcal{F}_{\textsf{C2A}}$ functionality for converting $\langle\!\langle \pi \rangle\!\rangle$ into an additive sharing $[\![\pi]\!]$. $\mathcal{F}_{\textsf{C2A}}$ is realized by the $\Pi_{\textsf{C2A}}$ protocol.

Theorem 7. *Protocol $\Pi_{\textsf{A2C}}$ securely realizes the $\mathcal{F}_{\textsf{A2C}}$ functionality in the semi-honest setting.*

Proof. Observe that permutation composition $\gamma := \theta \circ \omega$ can be computed by considering the vector representation and computing $\gamma := \theta(\omega)$, where γ, ω are vectors. Therefore $\Pi_{\textsf{A2C}}$ computes shares of $\rho = \langle\!\langle \pi \rangle\!\rangle_2^{-1} \circ \pi$. If we multiply from the left by $\langle\!\langle \pi \rangle\!\rangle_2$ we obtain

$$\langle\!\langle \pi \rangle\!\rangle_2 \circ \langle\!\langle \pi \rangle\!\rangle_1 = \pi$$

as desired. Therefore, the protocol is correct. Privacy follows from a straightforward simulation of $\mathcal{F}_{\textsf{Basic-Perm}}$. □

Theorem 8. *Protocol $\Pi_{\textsf{C2A}}$ securely realizes the $\mathcal{F}_{\textsf{C2A}}$ functionality in the semi-honest setting.*

Functionality $\mathcal{F}_{\mathsf{A2C}}$ $(\llbracket \pi \rrbracket)$:

INPUT: Additively shared permutation $\llbracket \pi \rrbracket$.
OUTPUT: Composed permutation $\langle\!\langle \pi \rangle\!\rangle$.

Protocol $\Pi_{\mathsf{A2C}}(\llbracket \pi \rrbracket)$:

1. The second party samples a random permutation $\langle\!\langle \pi \rangle\!\rangle_2$.
2. The parties invoke $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ on permutation $\langle\!\langle \pi \rangle\!\rangle_2^{-1}$ and input $\llbracket \pi \rrbracket$. They receive $\llbracket \rho \rrbracket$ as the result.
3. The parties reveal $\llbracket \rho \rrbracket$ to the first party who computes $\langle\!\langle \pi \rangle\!\rangle_1 := \rho$.
4. The parties output $\langle\!\langle \pi \rangle\!\rangle$.

Fig. 14. The $\mathcal{F}_{\mathsf{A2C}}$ functionality and Π_{A2C} protocol that converts an additively secret-shared permutation $\llbracket \pi \rrbracket$ into a composed permutation $\langle\!\langle \pi \rangle\!\rangle$.

Functionality $\mathcal{F}_{\mathsf{C2A}}$ $(\langle\!\langle \pi \rangle\!\rangle)$:

INPUT: Composed permutation $\langle\!\langle \pi \rangle\!\rangle$.
OUTPUT: Additively shared permutation $\llbracket pi \rrbracket$.

Protocol $\Pi_{\mathsf{C2A}}(\langle\!\langle \pi \rangle\!\rangle)$:

1. The parties invoke $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ on permutation $\langle\!\langle \pi \rangle\!\rangle_2$ and for input a sharing of $\langle\!\langle \pi \rangle\!\rangle_1$[a]. They receive $\llbracket \pi \rrbracket$ as the result.
2. The parties output $\llbracket \pi \rrbracket$.

[a] $\langle\!\langle \pi \rangle\!\rangle_1$ can be considered an additive sharing by defining the first share as $\langle\!\langle \pi \rangle\!\rangle_1$ and the second share as 0. We also note that $\mathcal{F}_{\mathsf{Basic\text{-}Perm}}$ can trivially be extended to accept a plaintext input list.

Fig. 15. The $\mathcal{F}_{\mathsf{C2A}}$ functionality and Π_{C2A} protocol that converts composed permutation $\langle\!\langle \pi \rangle\!\rangle$ into an additively secret-shared permutation $\llbracket \pi \rrbracket$.

Proof. As detailed in the proof of Theorem 7, $[\![\pi]\!]$ is correctly computed due to $\pi = \langle\!\langle\pi\rangle\!\rangle_2 \circ \langle\!\langle\pi\rangle\!\rangle_1 = \langle\!\langle\pi\rangle\!\rangle_2(\langle\!\langle\pi\rangle\!\rangle_1)$. Therefore, the protocol is correct. Privacy follows from a straightforward simulation of $\mathcal{F}_{\textsf{Basic-Perm}}$. □

Corollary 1. *As implied in the proof of Theorem 7, the parties can compute composition of two permutations when the first is shared in the permutation group and the second is additively shared.*

Theorem 9. *The composition of $\mathcal{F}_{\textsf{A2C}}$ and $\mathcal{F}_{\textsf{Comp-Perm}}$ realizes the $\mathcal{F}_{\textsf{Add-Perm}}$ functionality in the semi-honest setting.*

Proof. Correctness follows by inspection. Similarly, privacy can be demonstrated via straightforward simulation. □

8 Extraction Protocols

We present our extraction protocols at a high level in Section 4.4. We now present them in formal detail:

8.1 Extract Unordered

$\mathcal{F}_{\textsf{Ext-Unord}}$ (Figure 16) receives as input $[\![X]\!]$ and a bitvector $[\![f]\!]$ such that $|X| = |f| = n$. f_i indicates if X_i is marked. Then it extracts and outputs all marked elements $[\![Y]\!] = \{[\![X_i]\!] | f_i = 1\}$ in a uniform order. Additionally, it outputs the permutation π and size c such that $Y = \pi(X)_{[c]}$. Therefore, one can unextract by simply computing $\pi^{-1}(Y||0^{n-c})$.

The protocol $\mathit{\Pi}_{\textsf{Ext-Unord}}$ first associates each X_i with its corresponding f_i and permutes them by a random π. (step 2). Given the output size $c = \sum_i f_i$, observe that the $f' = \pi(f)$ is a uniformly random weight c vector. We can thus safely reveal the permuted f'. Then we select all elements of the permuted $[\![X]\!]' = \pi([\![X]\!])$ where $f'_i = 1$ (step 4,5) and output them (step 6). The resulting permutation is also returned by computing the "selection" permutation ρ with π.

Theorem 10. *Protocol $\mathit{\Pi}_{\textsf{Ext-Unord}}$ realizes the $\mathcal{F}_{\textsf{Ext-Unord}}$ functionality in the semi-honest setting.*

Proof. Correctness can be verified by inspection. The simulation of $\mathcal{F}_{\textsf{Comp-Perm}}$ is straightforward. To simulate f', observe that the ideal output includes the weight c of f. Therefore, the simulator can simulate f' by sampling a uniformly random f' with weight c and then sampling a consistent π. This distribution is identical, and therefore f' reveals no information beyond the desired output. □

Functionality $\mathcal{F}_{\text{Ext-Unord}}([\![X]\!],[\![f]\!])$:

Input: Secret-shared list $[\![X]\!] \in \mathbb{F}^n$, flags $[\![f]\!] \in \{0,1\}^n$.
Output: $([\![Y]\!], \langle\!\langle \pi \rangle\!\rangle, c)$. Count $c := \sum_i f_i$, permutation $\langle\!\langle \pi \rangle\!\rangle$ where $\pi : [n] \to [n]$ is random subject to $\pi(f) = 1^*||0^*$, and $Y = \pi(X)$, i.e. the first c items of Y are $\{X_i \mid f_i = 1\}$ and are in random order.

Protocol $\Pi_{\text{Ext-Unord}}([\![X]\!],[\![f]\!])$:

```
1. Locally sample random $\langle\!\langle \pi \rangle\!\rangle$.
2. $[\![X']\!] \bowtie [\![f']\!] := \mathcal{F}_{\text{Comp-Perm}}(\langle\!\langle \pi \rangle\!\rangle, [\![X]\!] \bowtie [\![f]\!])$
3. $f' := \mathcal{F}_{\text{Open}}([\![f']\!]), c := \sum_i f'_i$
4. Sample an arbitrary permutation $\rho$ s.t. $\rho(f') = 1^c 0^{n-c}$
5. $[\![Y]\!] := \rho([\![X'_i]\!])$
6. return $([\![Y]\!], \rho \circ \langle\!\langle \pi \rangle\!\rangle, c)$
```

Fig. 16. $\Pi_{\text{Ext-Unord}}$ implements $\mathcal{F}_{\text{Ext-Unord}}$. It outputs the extracted elements in random order. $\Pi_{\text{Ext-Unord}}$ runs in $O(n)$ time & communication and $O(1)$ rounds.

8.2 Extract Ordered

$\mathcal{F}_{\text{Ext-Ord}}$ (Figure 17) is also similar to $\mathcal{F}_{\text{Ext-Unord}}$, but outputs the extracted X_i in the order they appear in X.

The protocol $\Pi_{\text{Ext-Ord}}$ uses $\Pi_{\text{Ext-Unord}}$ as a subprotocol. As the output of $\Pi_{\text{Ext-Unord}}$ is in random order, we mark X with additional information that will enable to place the extracted X_i in their original order in X. I.e., we compute $[\![\rho]\!]$, the number of marked elements before each X_i, via simple local additions (step 1), mark X with ρ and invoke $\Pi_{\text{Ext-Unord}}$ (step 2), open the extracted ρ' to both parties (step 3), and permute the extracted X' according to ρ' (step 4). We then output the resulting list (step 5) along with the shared permutation and output size c.

Functionality $\mathcal{F}_{\text{Ext-Ord}}$ $([\![X]\!],[\![f]\!])$:

Input: Secret-shared list $[\![X]\!] \in \mathbb{F}^n, [\![f]\!] \in \{0,1\}^n$ of size n.
Output: $([\![Y]\!], \langle\!\langle \pi \rangle\!\rangle, c)$. Count $c := \sum_i f_i$, permutation $\langle\!\langle \pi \rangle\!\rangle$ where $\pi : [n] \to [n]$ is subject to $\pi(f) = 1^c||0^*$ and $\pi(i) < \pi(i')$ for $0 \le i < i' < c$, and $Y = \pi(X)$, i.e. the first c items of Y are $\{X_i \mid f_i = 1\}$ and are in order.

Protocol $\Pi_{\text{Ext-Ord}}([\![X]\!],[\![f]\!])$:

```
1. $[\![\rho_i]\!] := \Sigma_{j \le i} [\![f_j]\!]$
2. $([\![X']\!] \bowtie [\![\rho']\!], \langle\!\langle \pi' \rangle\!\rangle) := \mathcal{F}_{\text{Ext-Unord}}([\![X]\!] \bowtie [\![\rho]\!], [\![f]\!], c)$
3. $\rho' := \mathcal{F}_{\text{Open}}([\![\rho']\!]_{[c]})||[c,n]$
4. $[\![Y]\!] := \rho'^{-1}([\![X']\!])$
5. return $([\![Y]\!], \rho'^{-1} \circ \langle\!\langle \pi' \rangle\!\rangle, c)$
```

Fig. 17. $\Pi_{\text{Ext-Ord}}$ implements $\mathcal{F}_{\text{Ext-Ord}}$. It outputs the extracted elements in the original order they appear in the input list X. $\Pi_{\text{Ext-Unord}}$ runs in $O(n)$ time & communication and $O(1)$ rounds.

Theorem 11. *Protocol $\Pi_{\textsf{Ext-Ord}}$ realizes the $\mathcal{F}_{\textsf{Ext-Ord}}$ functionality in the semi-honest setting.*

Proof. Correctness can be verified by inspection. Simulating $\mathcal{F}_{\textsf{Ext-Unord}}$ is straightforward. For simulating ρ', observe that ρ contains $[c]$ in monotonically increasing order. Moreover, for i with $f_i = 1$, ρ_i is one larger than its predecessor. Therefore, $\mathcal{F}_{\textsf{Ext-Unord}}$ will return the ρ_i for $f_i = 1$ in a random order and these ρ_i will all be unique. This can be simulated simply by returning a random permutation of $[c]$. □

8.3 Extract Unordered Padded

$\mathcal{F}_{\textsf{Ext-Unord-Pad}}$ (Figure 18) is similar to $\mathcal{F}_{\textsf{Ext-Unord}}$ but pads the output to fixed length t. If fewer than t flags are one, then the output will include padding elements from a second input $[\![P]\!]$. Let $c := \Sigma_{i<n} f_i$ denote the number of flagged inputs. The protocol $\Pi_{\textsf{Ext-Unord-Pad}}$ first invokes $\Pi_{\textsf{IndexToOneHot}}$ (step 1), which returns a one-hot vector $[\![h]\!]$ of size t. If $c \leq t$, then $h_c = 1$ and otherwise is zero. In step 2, we extend f with the prefix sum of h. Note that after this step $|\{i \mid f_i = 1\}| = n + t$. In step 3, we extend $[\![X]\!]$ with the padding elements and then invoke $\mathcal{F}_{\textsf{Ext-Unord}}$ to extract $X_i | f_i = 1$ (step 4) along with any padding elements.

Now that we explained $\Pi_{\textsf{Ext-Unord-Pad}}$, we look more closely at $\Pi_{\textsf{IndexToOneHot}}$ invoked in step 1. $\Pi_{\textsf{IndexToOneHot}}$ constructs a complete binary tree of depth $w := \lceil \log_2(c) \rceil$ (step 1). Our invariant is that the node values at each level represent a one-hot vector. At the root we have a size 1 one-hot vector. At the next levels i, we have size 2^i one-hot vectors. This results in the last level having c leaves, all zeros but at one point. Our technique carefully arranges that the nonzero value is at point I. The values at the leaves represent the output of $\Pi_{\textsf{IndexToOneHot}}$. We denote each node as $t_{i,j}$, where i is the current level of the tree and j is the node index at that level (from the left). We bit decompose I into $I_{w-1}, \ldots, I_0$ (I_{w-1} is the MSB), which represents the path to $t_{w-1,I} = 1$ (step 2).

To maintain our invariant, we set the root of the tree $t_{0,0} := 1$ (step 3) and next proceed with setting the remaining $t_{i,j}$. We iterate over each level i and over each node j in that level (step 4). In step a we compute the position (i, j) of the current parent p and the position of its children c_0, c_1.

In steps b-c, we set the node values of the children t_{c_0} and t_{c_1}. We set them such that the one-hot position at the children's level is in exactly one of the children of the one-hot position in the parent's level. We use I_{w-1-i} to determine if the left child is 1 ($I_{w-1-i} = 0$) or the right child ($I_{w-1-i} = 1$). This ensures that we hold 1 on the path to position I in the leaf level. More specifically, we look at all the *right* children of a given level. It will be 1 if and only if the parent $t_p = 1$ AND $I_{w-1-i} = 1$ (step b). Then we look at all the *left* children of a given level. It will be 1 if and only if the parent $t_p = 1$ AND $I_{w-1-i} = 0$. We can optimize this logic. We know that $t_{c_0} \oplus t_{c_1} \oplus t_p = 0$. I.e., if the parent $t_p = 0$, then both children $t_{c_0} = t_{c_1} = 0$. If the parent $t_p = 1$, then at most one child t_{c_0} or t_{c_1} is 1. Hence, we can simply define the right child as $t_{c_0} \oplus t_p$ (step c). Now that we computed all $t_{i,j}$, we output the leaves t_{w-1} (step 5).

Functionality $\mathcal{F}_{\textsf{Ext-Unord-Pad}}([\![f]\!], [\![X]\!], [\![P]\!])$:

Input: Secret-shared flags $[\![f]\!] \in \{0,1\}^n$. Optionally, shared list $[\![X]\!] \in \mathbb{F}^n$, padding list $[\![P]\!] \in \mathbb{F}^t$.
Output: $([\![Y]\!], \langle\!\langle\pi\rangle\!\rangle, c)$. Count $c := \mathsf{max}(f^*, t)$ where $f^* = \sum_i f_i$. Shared permutation $\langle\!\langle\pi\rangle\!\rangle$ where $\pi : [n+t] \to [n+t]$ is random subject to $\pi(f||1^r||0^{t-r}) = 1^*||0^*$ where $r := c - f^*$. Additionally, if the optional $[\![X]\!], [\![P]\!]$ are provided, output a sharing of $Y = \pi(X||P)$; the first c positions of Y contain $\{X_i \mid f_i = 1\} \cup \{P_i \mid i \in [r]\}$ in a random order.

Protocol $\Pi_{\textsf{Ext-Unord-Pad}}([\![f]\!], [\![X]\!], [\![P]\!])$:

```
1. $[\![h]\!] := \Pi_{\textsf{IndexToOneHot}}(\Sigma_{i<n}[\![f_i]\!], t)$
2. $[\![f_{n+i}]\!] := \bigoplus_{j<i} [\![h_j]\!]$ for $i \in [t]$.
3. $[\![X]\!] := [\![X]\!]||[\![P]\!]$
4. return $\mathcal{F}_{\textsf{Ext-Unord}}([\![X]\!], [\![f]\!])$
```

Protocol $\Pi_{\textsf{IndexToOneHot}}([\![I]\!], t)$:

```
1. $w := \lceil \log_2(t) \rceil$
2. $[\![I_{w-1}]\!], \ldots, [\![I_0]\!] := \mathcal{F}_{\textsf{Bit-Decomp}}([\![I]\!], w)$.
3. $[\![t_{0,0}]\!] := 1$
4. for $i = [w-1]$ and parallel for $j \in [2^i]$:
   a. $p := (i, j), c_0 := (i+1, 2j), c_1 := (i+1, 2j+1)$
   b. $[\![t_{c_0}]\!] := [\![t_p]\!] \wedge [\![I_{w-1-i}]\!]$
   c. $[\![t_{c_1}]\!] := [\![t_{c_0}]\!] \oplus [\![I_{w-1-i}]\!]$
5. return $[\![t_{w-1}]\!]$
```

Fig. 18. $\Pi_{\textsf{Ext-Unord-Pad}}$ implements $\mathcal{F}_{\textsf{Ext-Unord-Pad}}$. Like $\Pi_{\textsf{Ext-Unord}}$, it extracts elements in random order. Additionally, the output list is padded to size c so that the number of non-dummies $c' \leq c$ stays private. The dummies are interspersed with the output. It runs in $O(n)$ time & communication and $O(1)$ rounds.

Theorem 12. *Protocol $\Pi_{\textsf{Ext-Unord-Pad}}$ realizes the $\mathcal{F}_{\textsf{Ext-Unord-Pad}}$ functionality in the semi-honest setting.*

Proof. Correctness can be verified by inspection. Privacy can be demonstrated by invoking the simulator for ideal functionalities and that of a generic MPC protocol. □

8.4 Extract Ordered Padded

$\mathcal{F}_{\textsf{Ext-Ord-Pad}}$ (Figure 19) is the equivalent of $\mathcal{F}_{\textsf{Ext-Unord-Pad}}$, but outputs extracted elements $\{X_i \mid f_i = 1\}$ in their original order (followed by dummies to pad to size t). The protocol $\Pi_{\textsf{Ext-Ord-Pad}}$ combines the ideas from $\Pi_{\textsf{Ext-Ord}}$ and $\Pi_{\textsf{Ext-Unord-Pad}}$. Like $\Pi_{\textsf{Ext-Ord}}$, $\Pi_{\textsf{Ext-Ord-Pad}}$ computes $[\![t]\!]$ in step 1 to remember the original order in X. But since our protocol also pads (similarly to $\Pi_{\textsf{Ext-Unord-Pad}}$), it additionally computes $[\![t]\!]$ for the c appended elements so that after extraction they are placed at the end of the list (step 4). Steps 2-3 are equivalent to those in $\Pi_{\mathsf{Ext\text{-}Unord\text{-}Pad}}$, and output a bitvector of size c that is non-zero only for the number of elements that should be padded. This ensures that the extracted output is of size c. This bitvector is then used in step 4 to extend t for the padded elements. Step 5 sets all values of X in the c appended elements to zeros (to mark them as dummies). Now X contains c elements with $f_i = 1$ and their associated order t. Hence, we are ready to extract $[\![X']\!] \bowtie [\![t']\!]$ (step 6). The remaining steps 7-9 are now identical to $\Pi_{\mathsf{Ext\text{-}Unord\text{-}Pad}}$. We open t, permute $[\![X']\!]$ based on t, and output.

Functionality $\mathcal{F}_{\mathsf{Ext\text{-}Ord\text{-}Pad}}([\![f]\!], [\![X]\!], [\![P]\!])$:

Input: Secret-shared flags $[\![f]\!] \in \{0,1\}^n$. Optionally, shared list $[\![X]\!] \in \mathbb{F}^n$, padding list $[\![P]\!] \in \mathbb{F}^t$.
Output: $([\![Y]\!], \langle\!\langle \pi \rangle\!\rangle, c)$. Count $c := \mathsf{max}(f^*, t)$ where $f^* = \sum_i f_i$. Shared permutation $\langle\!\langle \pi \rangle\!\rangle$ where $\pi : [n+t] \to [n+t]$ is random subject to $\pi(f||1^r||0^{t-r}) = 1^*||0^*$ where $r := c - f^*$. Additionally, if the optional $[\![X]\!], [\![P]\!]$ are provided, output a sharing of $Y = \pi(X||P)$; the first c positions of Y contain $\{X_i \mid f_i = 1\} \cup \{P_i \mid i \in [r]\}$ in their original order.

Protocol $\Pi_{\mathsf{Ext\text{-}Ord\text{-}Pad}}([\![f]\!], [\![X]\!], [\![P]\!])$:

```
1. $[\![d_i]\!] := \Sigma_{j<i}[\![f_j]\!]$ for $i \in [n]$
2. $[\![h]\!] := \Pi_{\mathsf{IndexToOneHot}}(\Sigma_{i<n}[\![f_i]\!], t)$
3. $[\![f_{n+i}]\!] := \bigoplus_{j \le i} [\![h_j]\!]$ for $i \in [c]$
4. $[\![d_{n+i}]\!] := [\![f_{n+i}]\!] \cdot i$ for $i \in [t]$
5. $[\![X]\!] := [\![X]\!]||[\![P]\!]$
6. $([\![X']\!] \bowtie [\![d']\!], [\![\pi]\!], c) := \mathcal{F}_{\mathsf{Ext\text{-}Unord}}([\![X]\!] \bowtie [\![d]\!], [\![f]\!])$
7. $\rho := \mathcal{F}_{\mathsf{Open}}([\![d']\!]_{[c]})||[c, n+t]$
8. $[\![Y]\!] := \rho([\![X'_i]\!])$
9. return $([\![Y]\!], \rho \circ \langle\!\langle \pi \rangle\!\rangle, c)$.
```

Fig. 19. $\Pi_{\mathsf{Ext\text{-}Ord\text{-}Pad}}$ implements $\mathcal{F}_{\mathsf{Ext\text{-}Ord\text{-}Pad}}$. Like $\Pi_{\mathsf{Ext\text{-}Ord}}$, it extracts elements in the order they appear in X. Additionally, the output list is padded to size c so that the number of non-dummies $c' \le c$ stays private. The dummies are (obliviously) placed at the end of the output list. It runs in $O(n)$ time & communication and $O(1)$ rounds.

Theorem 13. *Protocol $\Pi_{\mathsf{Ext\text{-}Ord\text{-}Pad}}$ realizes the $\mathcal{F}_{\mathsf{Ext\text{-}Ord\text{-}Pad}}$ functionality in the semi-honest setting.*

Proof. Correctness can be verified by inspection. The simulation of $\Pi_{\mathsf{IndexToOneHot}}$ and $\mathcal{F}_{\mathsf{Ext\text{-}Unord}}$ directly follows from their simulators. For simulating ρ, observe that ρ contains $[c]$ in monotonically increasing order. Moreover, for i with $f_i = 1$, ρ_i is one larger than its predecessor. Any missing ρ_i values up to t are then manually included in step 4. Therefore, $\mathcal{F}_{\mathsf{Ext\text{-}Unord}}$ will return the ρ_i for $f_i = 1$ in a random order and these ρ_i will all be unique. This can be simulated simply by returning a random permutation of $[c]$. □

9 Sorting

We demonstrate that several prior works [CHI$^+$19,AHI$^+$22,HKI$^+$13] can be efficiently implemented in our framework. The ideal functionality $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ is presented in Figure 20. $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ is stable, and hence ensures that equal values maintain their order. It outputs a permutation that sorts the inputs.

Functionality $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$:

$\mathcal{F}_{\mathsf{Stable\text{-}Sort}}(\llbracket X \rrbracket)$:
Upon input $\llbracket X \rrbracket$, compute stable sorting permutation π of X. Return $\langle\!\langle \pi \rangle\!\rangle$.

Fig. 20. The random and stable sorting functionality $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$. We note that one could consider outputting π in additive format $\llbracket \pi \rrbracket$.

9.1 Partition

Most prior works were presented in the three-party honest-majority setting due to the existence of efficient permutations. [CHI$^+$19,AHI$^+$22] implement radix sort and generate a sorting permutation of a single bit using a circuit. In particular, they consider a subprotocol, which we denote as $\Pi_{\mathsf{Partition}}$. It takes as input a bit vector X and returns the stable sorting permutation. We refer to 4.5 for the intuition of this protocol. We note that compared to [CHI$^+$19], we optimize this protocol with the use of $\Pi_{\mathsf{IndexToOneHot}}$ which requires significantly less communication.

Protocol $\Pi_{\mathsf{Partition}}$:

```
Π_Partition-Add(⟦X⟧) :

 1. $i \in [n] : (\llbracket f_{1,i} \rrbracket, ..., \llbracket f_{2^\ell,i} \rrbracket) := \Pi_{\mathsf{IndexToOneHot}}(\llbracket X_i \rrbracket, 2^\ell)$
 2. $j \in [2^\ell], i \in [n] : \llbracket s_{j,i} \rrbracket = \sum_{i',j' \text{ s.t. } j'<j \vee (j'=j \wedge i' \leq i)} \llbracket f_{i',j'} \rrbracket$
 3. $i \in [n] : \llbracket \pi_i \rrbracket = \sum_{j \in [2^\ell]} \llbracket s_{i,j} \rrbracket$
 4. return $\llbracket \pi \rrbracket$

Π_Partition(⟦X⟧) :

 1. return $\mathcal{F}_{\mathsf{A2C}}(\Pi_{\mathsf{Partition\text{-}Add}}(\llbracket X \rrbracket))$
```

Fig. 21. Protocol $\Pi_{\mathsf{Partition}}$ that implements the $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ functionality. Let n be the length of X and ℓ be the bit-length of the elements of X. It makes use of the subprotocol $\Pi_{\mathsf{Partition\text{-}Add}}$, which implements $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ with additive secret-shared output.

Theorem 14. *Protocol $\Pi_{\mathsf{Partition}}$ realizes the $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ functionality in the semi-honest setting.*

Proof. Correctness can be verified by inspection. Simulation follows from only making use of generic computation for circuits and the simulator for $\mathcal{F}_{\mathsf{A2C}}$. □

9.2 Radix Sort

To sort multiple bits, one can invoke $\Pi_{\mathsf{Partition}}$ multiple times, starting with the most significant bit and then compose the resulting permutations to get the overall sorting permutation. Given access to $O(n)$ time permutation gates, the radix sort protocol $\Pi_{\mathsf{Radix\text{-}Sort}}$ of Figure 22 requires $O(n\ell \log \ell)$ communication and $O(\ell)$ rounds to stable sort n items of length ℓ.

Protocol $\Pi_{\mathsf{Radix\text{-}Sort}}$:

Parameter: Let $t \in \mathbb{N}$ be a hyperparameter with default value 2.

```
Π_Radix-Sort([[X]]) :

1. Let $[\![X'_i]\!] := [\![X_{i,[\ell-t,\ell]}]\!]$ be the $t$ most significant bits of $[\![X_i]\!]$ for $i \in [n]$
2. $\langle\!\langle \pi \rangle\!\rangle := \Pi_{\mathsf{Partition}}([\![X']\!])$
3. For $j \in [1, \lceil \ell/t \rceil]$ :
   (a) $\ell' := \ell - jt$
   (b) $[\![X'_i]\!] := [\![X_{i,[\ell'-t,\ell']}]\!]$ for $i \in [n]$
   (c) $[\![\rho]\!] := \Pi_{\mathsf{Partition\text{-}Add}}([\![X']\!])$
   (d) $[\![\pi]\!] := \mathcal{F}_{\mathsf{Comp\text{-}Perm}}(\langle\!\langle \pi \rangle\!\rangle, [\![\rho]\!])$
   (e) $\langle\!\langle \pi \rangle\!\rangle := \mathcal{F}_{\mathsf{A2C}}([\![\pi]\!])$
4. return $\langle\!\langle \pi \rangle\!\rangle$
```

Fig. 22. Protocol $\Pi_{\mathsf{Radix\text{-}Sort}}$ implements the $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ functionality.

Theorem 15. *Protocol $\Pi_{\mathsf{Radix\text{-}Sort}}$ realizes the $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ functionality in the semi-honest setting.*

Proof. Correctness can be verified by inspection. Simulation follows from the simulators for $\Pi_{\mathsf{Partition}}, \mathcal{F}_{\mathsf{Comp\text{-}Perm}}, \mathcal{F}_{\mathsf{A2C}}$. □

9.3 Quick Sort

The latter, [HKI+13], take a different approach in the so-called shuffle-reveal model. The idea is that if the input lists are first shuffled, one can run most insecure comparison-based sorting algorithms, e.g. quick sort, where each comparison is replaced by a protocol that only reveals the result of the comparison. This protocol makes use of $O(n \log n)$ comparisons over $O(\log n)$ steps. The running time of this protocol is $O(n\ell \log n \log \ell)$ and $O(\log n\ell \log \ell)$ rounds assuming a comparison requires $O(\ell \log \ell)$ time and $O(\ell)$ rounds.

However, one shortcoming of a direct implementation of this paradigm is that the inputs must be totally ordered, i.e. with *no duplicate values*. Let us consider quick sort and consider the view of the parties if all elements are the same or all are different. If they are the same, the result of the comparisons will all be the same, while different values implies that the comparisons will be uniformly random. [HKI+13] proposed a simple solution to make any input list totally ordered by appending the index of the item as its least significant bit, e.g. $X'_i := X_i \cdot n + i$. Then it is possible to invoke the shuffle-reveal compiler and get secure sort. One benefit of this approach is that the resulting sort is stable while quick sort is usually unstable. We present this protocol as $\Pi_{\mathsf{Quick\text{-}Sort}}$ in Figure 23. The overall running time is $O(n\ell' \log n \log \ell')$ and $O(\log n + \log \ell')$ rounds where $\ell' := \ell + \log n$.

Protocol $\Pi_{\mathsf{Quick\text{-}Sort}}$:

$\Pi_{\mathsf{Quick\text{-}Sort\text{-}Impl}}([\![X]\!])$:

```
1. The parties jointly sample $i \leftarrow [n]$
2. $[\![p]\!] := [\![X_i]\!]$ and $c_i := 1$
3. for $j \in [n] \setminus \{i\}$ : $c_j := \mathcal{F}_{\mathsf{Open}}(([\![X_j]\!] \cdot 2 + [\![t]\!]) < ([\![p]\!] \cdot 2 + [\![s]\!])) \cdot 2$ where $[\![t]\!]$ is sampled
   as a uniform bit and $[\![s]\!] := 1 - [\![t]\!] = 1 \oplus [\![t]\!]$.
4. Let $\rho$ be the sorting permutation for $c$
5. $[\![X']\!] := \rho([\![X]\!])$
6. $i' := \rho^{-1}(i)$
7. $[\![Z]\!] := [\![X'_{[i'-1]}]\!]$
8. $[\![Y]\!] := [\![X'_{[i'+1,n]}]\!]$
9. $(\theta, \gamma) := (\Pi_{\mathsf{Quick\text{-}Sort\text{-}Impl}}([\![Z]\!]), \Pi_{\mathsf{Quick\text{-}Sort\text{-}Impl}}([\![Y]\!]))$
10. return $(\theta || i' || \gamma + i') \circ \rho$
```

$\Pi_{\mathsf{Quick\text{-}Sort}}([\![X]\!])$:

```
1. The parties sample $\langle\!\langle \pi \rangle\!\rangle$ uniformly at random
2. $[\![X']\!] := \mathcal{F}_{\mathsf{Comp\text{-}Perm}}(\langle\!\langle \pi \rangle\!\rangle, [\![X]\!])$
3. return $\Pi_{\mathsf{Quick\text{-}Sort\text{-}Impl}}([\![X']\!], [n]) \circ \langle\!\langle \pi \rangle\!\rangle$
```

Fig. 23. Protocol $\Pi_{\mathsf{Quick\text{-}Sort}}$ that implements the $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ functionality.

Theorem 16. *Protocol $\Pi_{\mathsf{Quick\text{-}Sort}}$ realizes the $\mathcal{F}_{\mathsf{Stable\text{-}Sort}}$ functionality in the semi-honest setting.*

Proof. Correctness can be verified by inspection. The simulator works by sampling a random permutation $\theta : [n] \to [n]$. For the first invocation of $\Pi_{\mathsf{Quick\text{-}Sort\text{-}Impl}}$, the c_j values are computed as

$$c_j := \theta^{-1}(i) < \theta^{-1}(j).$$

Subsequent recursions of $\Pi_{\mathsf{Quick\text{-}Sort\text{-}Impl}}$ are computed the same way with the indices appropriately updated based on the subrange in question.

Observe that this perfectly simulates the protocol. The c_j collectively specify θ exactly. In the real protocol, the overall sorting permutation is $\rho = \theta \circ \pi$ where π is uniformly distributed. Therefore, $\theta := \rho \circ \pi^{-1}$ is also uniformly distributed. $\square$

10 Batched Random Access Memory

We now present a generalization of our permutation functionalities to allow the parties to input an arbitrary selection vector σ and list X. The protocol results in the parties performing batched random access into X based on σ. I.e. it returns the sharing $[\![Y]\!] := ([\![X_{\sigma(1)}]\!], ..., [\![X_{\sigma(m)}]\!])$. This protocol makes use of the aggregation tree technique first presented by [BDG+22] in the three-party honest majority setting and summarized in Section 3.4.

The protocol begins by applying a stable sort to the set of indices $[n]$ extended with the selection vector σ to obtain the sorting permutation π. Intuitively, we will use π to place each X_i before the "output" positions that want to access X_i. More specifically, π is used to permute both the values X concatenated with m dummy elements to obtain X'. Additionally, the parties compute flags $b = \pi(0^n || 1^m)$ to denote which items are dummies, i.e. X'_i is a dummy if $b_i = 1$. The critical property of X' is that each non-dummy X'_i is followed by t dummies where t is the number of times it is being selected.

The parties then invoke the aggregation tree functionality $\mathcal{F}_{\mathsf{Agg}}$ with the flag bits b and the permuted list X'. $\mathcal{F}_{\mathsf{Agg}}$ will duplicate each non-dummy X'_i into the next set of contiguous dummy positions [BDG+22]. We refer to Section 3.4 or [BDG+22] for more details on how this is achieved but overall it requires $O(n\ell)$ time and $O(\log n)$ rounds. Let Z denote the result of $\mathcal{F}_{\mathsf{Agg}}$. The parties can then unpermute the aggregated vector Z to Y. The first n positions of Z will be X while the last m positions will be the desired output Y.

Functionality $\mathcal{F}_{\mathsf{Batched\text{-}RAM\text{-}Read}}([\![\sigma]\!], [\![X]\!])$:

Input: A shared selection $[\![\sigma]\!]$ where $\sigma \in [n]^m$ and shared list $[\![X]\!]$ where $X \in \mathbb{F}^{n \times \ell}$
Output: A shared vector $[\![Y]\!]$ such that $Y_i = X_{\sigma(i)}$

Protocol $\Pi_{\mathsf{Batched\text{-}RAM\text{-}Read}}([\![\sigma]\!], [\![X]\!])$:

```
1. $\langle\!\langle \pi \rangle\!\rangle := \mathcal{F}_{\mathsf{Stable\text{-}Sort}}([n] || [\![\sigma]\!])$
2. $[\![b]\!] := \mathcal{F}_{\mathsf{Comp\text{-}Perm}}(\langle\!\langle \pi \rangle\!\rangle, 0^n || 1^m)$
3. $[\![X']\!] := \mathcal{F}_{\mathsf{Comp\text{-}Perm}}(\langle\!\langle \pi \rangle\!\rangle, [\![X]\!] || 0^m)$
4. $[\![Z]\!] := \mathcal{F}_{\mathsf{Agg}}([\![b]\!], [\![X']\!], \mathsf{dup})$ where $\mathsf{dup}(x, y) := x$.
5. $[\![Y]\!] := \mathcal{F}_{\mathsf{Comp\text{-}Perm}}(\langle\!\langle \pi \rangle\!\rangle^{-1}, [\![Y]\!])$
6. return $[\![Y]\!]_{[n, n+m]}$
```

Fig. 24. Protocol $\Pi_{\mathsf{Batched\text{-}RAM\text{-}Read}}$ that implements the $\mathcal{F}_{\mathsf{Batched\text{-}RAM\text{-}Read}}$ functionality.

Theorem 17. *The protocol $\Pi_{\text{Batched-RAM-Read}}$ realizes the $\mathcal{F}_{\text{Batched-RAM-Read}}$ functionality in the semi-honest setting.*

Proof. Correctness can be verified by inspection. Simulation follows from a simple composition argument. □

We note that the inverse write operation of $\Pi_{\text{Batched-RAM-Read}}$ has the following challenges. First, we need to define what happens when multiple inputs are written to the same output position. We consider several options, the output could take the first or the last value to be written. More generally, the caller can define their own associative operator $\star(x, y)$ and the values mapped to an output position j are computed as $\bigstar_{i\in\{i|\sigma_j=i\}} X_i$.

Lastly, we need to define the value of output position j for which σ has no mapping, i.e. $|\{i \mid \sigma_j = i\}| = 0$. The most natural option is to provide a default value D_j. Overall, the jth output position will be the summation of the jth default value and the input values written to it, i.e. $Y_j := D_j \star \bigstar_{i\in\{i|\sigma_j=i\}} X_i$. We note that simple modifications of the protocol can select the default only if no X_i are mapped to it. The full protocol $\Pi_{\text{Batched-RAM-Write}}$ is presented in Figure 25. It follows a similar logic as $\Pi_{\text{Batched-RAM-Read}}$.

Functionality $\mathcal{F}_{\text{Batched-RAM-Write}}(\llbracket\sigma\rrbracket, \llbracket X\rrbracket, \llbracket D\rrbracket, \star)$:

INPUT: A shared selection $\llbracket\sigma\rrbracket$ where $\sigma \in [n]^m$ and shared list $\llbracket X\rrbracket, \llbracket D\rrbracket$ where $X \in \mathbb{F}^{n\times\ell}, D \in \mathbb{F}^{m\times\ell}$ and associative operator $\star : \mathbb{F}^\ell \times \mathbb{F}^\ell \to \mathbb{F}^\ell$.
OUTPUT: A shared vector $\llbracket Y\rrbracket$ such that $Y_j = D_j \star \bigstar_{i\in\{i|\sigma_j=i\}} X_i$ for $j \in [m]$.

Protocol $\Pi_{\text{Batched-RAM-Write}}(\llbracket\sigma\rrbracket, \llbracket X\rrbracket, \llbracket D\rrbracket, \star)$:

```
1. $\langle\!\langle\pi\rangle\!\rangle := \mathcal{F}_{\text{Stable-Sort}}([n]||\llbracket\sigma\rrbracket)$
2. $\llbracket b\rrbracket := \mathcal{F}_{\text{Comp-Perm}}(\langle\!\langle\pi\rangle\!\rangle, 1^n||0^m)$
3. $\llbracket X'\rrbracket := \mathcal{F}_{\text{Comp-Perm}}(\langle\!\langle\pi\rangle\!\rangle, \llbracket D\rrbracket||\llbracket X\rrbracket)$
4. $\llbracket Z\rrbracket := \mathcal{F}_{\text{Agg}}(\llbracket b\rrbracket, \llbracket X'\rrbracket, \star)$.
5. $\llbracket Y\rrbracket := \mathcal{F}_{\text{Comp-Perm}}(\langle\!\langle\pi\rangle\!\rangle^{-1}, \llbracket Y\rrbracket)$
6. return $\llbracket Y\rrbracket_{[n]}$
```

Fig. 25. Protocol $\Pi_{\text{Batched-RAM-Write}}$ that implements the $\mathcal{F}_{\text{Batched-RAM-Write}}$ functionality.

Theorem 18. *The $\Pi_{\text{Batched-RAM-Write}}$ protocol realizes the $\mathcal{F}_{\text{Batched-RAM-Write}}$ functionality in the semi-honest setting.*

Proof. Correctness is by inspection. Simulation follows from composition. □

11 Evaluation

We implement many of our protocols and report on their performance. The implementation uses libOTe [PR] and primarily focuses on binary secret sharing. We intend to open source the implementation. Where more efficient, e.g.

Protocol \ n	Time (ms)			Comm. (MB)		
	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
$\Pi_{\mathsf{Basic\text{-}Perm}}$, Gazelle* [JVC18]	2,135	19,846	303,219	0.8	13.4	215
$\Pi_{\mathsf{Basic\text{-}Perm}}$, Paillier	143,030	572,120	2,288,480	4.2	67	1,075
$\Pi_{\mathsf{Basic\text{-}Perm}}$, Chase et al.[CGP20]	89	2,003	44,168	5	168	4,044
$\Pi_{\mathsf{Basic\text{-}Perm}}$, $\Pi_{\mathsf{Prf\text{-}Perm}}$ w/ [ARS+15]	970	13,498	250,689	3.8	61	975
$\Pi_{\mathsf{Basic\text{-}Perm}}$, $\Pi_{\mathsf{Prf\text{-}Perm}}$ w/ [APRR24]	**86**	**552**	**7,263**	**0.7**	**11.4**	**182**
$\Pi_{\mathsf{Partition}}$	86	986	16,012	8.0	20	324
$\Pi_{\mathsf{Radix\text{-}Sort}}$ $\ell = 32$	1,041	13,601	189,148	16.9	270	4,333
$\Pi_{\mathsf{Batched\text{-}RAM\text{-}Read}}$	521	9,881	177,421	7.9	163	3,183

Fig. 26. Performance metrics for running our protocols on lists of size n. The string length is $\ell = 128$ bits (except for sort). Time is measured in milliseconds and communication in MB. $\star$ Gazelle numbers are taken from [JVC18].

$\Pi_{\mathsf{Partition}}$, the protocols switch to arithmetic secret sharing and then back to binary. We employ the binary GMW [GMW87] protocol to evaluate circuits with correlated randomness generated using Silent OT [BCG+19b] with [RRT23]'s optimization. Each AND gate requires 4 bits of communication. We consider two implementations of a weak PRF. The first is LowMC PRP [ARS+15] and the second is alternating moduli-based PRF [APRR24]. For the latter, we make use of their implementation.

All performance numbers for our protocols, Chase et al. [CGP20] and Paillier were obtained by running the protocol on a laptop computer with modern i7 CPU and 16GB of RAM. All costs include the preprocessing cost. Communication is performed by copying values between memory buffers within a single process, i.e. network latency is near zero and the parties have greater than a 10Gbps connection. More realistic network latency will not significantly increase the running time due to our protocols mostly being constant round. Similarly, given that our protocols send the least amount of data, decreasing the bandwidth will make our protocols comparatively better.

We compare our protocols implementing $\mathcal{F}_{\mathsf{Gen\text{-}Perm}}$ to the lattice AHE-based Gazelle protocol [JVC18], the folklore Paillier AHE protocol, and the optimized Benes network of Chase et al. [CGP20]. Both AHE protocols follow the same outline. The receiver holds an input vector A and the sender holds a permutation π. The receiver encrypts the components of A using AHE, i.e. $[\![A_i]\!] = \mathsf{AHE.enc}_k(A_i)$, sends $[\![A]\!]$ to the sender who sends $[\![C]\!] = \pi([\![A]\!]) - B$ back where B is a random vector. The receiver can decrypt the result as C such that $B + C = \pi(A)$. Unlike our binary correlation, Gazelle uses a 20-bit integer modulus and Paillier requires a larger modulus, e.g. a 2000-bit integer. Although we do not report this overhead, one can convert such correlations to binary with additional overhead, e.g. $b \log b$ OTs and $\log b$ rounds for a b-bit modulus.

The Gazelle protocol presents various optimizations such as lattice-based AHE packing/SIMD where the multiplied plaintexts are packed into a single ciphertext. This results in better communication and encryption times, but adds additional complexity due to the need to permute values within a single ciphertext. We refer to [JVC18] for details, but note that ciphertexts must be decomposed, permuted, masked and then recombined. We *do not* include the time to mask or recombine. Moreover, to prevent leaking information about π via the distribution of LWE ciphertext noise, one must use an LWE/AHE scheme with circuit privacy [Klu22]. However, Gazelle does not consider this, and therefore the overhead of a fully secure scheme will be noticeably higher, e.g. due to the need to perform noise flooding. In particular, in Figure 26 we only report the times provided by [JVC18] to encrypt, decrypt, decompose and permute values within single ciphertext without circuit privacy.

The optimized Benes network of Chase et al. [CGP20] follows a different approach. They first implement a specialized permutation for some small number of items T, e.g. $T = 256$ that we use. Using the idea of a Benes permutation circuit, one can then compile many small permutations into a large permutation. To permute n items, their approach requires $n \log n/T \log T$ sub-permutations of size T. Their specialized sub-permutation protocol is based on punctured point functions. To implement this, we use the state-of-the-art implementation of [PR]. We note that the time reported only accounts for the generation of the sub-permutations and not the time required to actually permute the data. Their overall communication complexity is $O(\kappa n \log n + \ell n \log n / \log T)$. We chose $T = 256$ as this gives a good running time/communication trade off (larger T results in less communication but exponentially more time).

Our protocol $\Pi_{\mathsf{Basic\text{-}Perm}} + \Pi_{\mathsf{Prf\text{-}Perm}}$ with the alternating moduli PRF of [APRR24] outperforms the alternative of using Paillier additive homomorphic encryption by $300\times$ in terms of running time and $5\times$ for communication. We also report the performance metrics of our permutation protocol with the LowMC PRP [ARS+15] and observe that is is about two order of magnitude slower than [APRR24] and requires $7\times$ more rounds. The lattice based AHE scheme of Gazelle [JVC18] requires only slightly more communication[3] but noticable more computation time, e.g. $40\times$ more for $n = 2^{20}, \ell = 128$. Lastly, we compare with Chase et al. [CGP20] and observe that their protocol is the second fastest, requiring 44 seconds compared to our 7.2 seconds for $n = 2^{20}$, but consumes the most bandwidth, 4GB compared to our 0.18GB.

Six seconds of our running time is consumed by preprocessing the correlated randomness used by [APRR24], e.g. Beaver triples. When we parallelize this preprocessing using 4 threads, the overall running times falls to just 2.8 seconds.

We additionally implement the $\Pi_{\mathsf{Partition}}$ protocol and observe that the main overhead is generating the permutation correlation, which we implement using $\Pi_{\mathsf{Prf\text{-}Perm}}$ with [APRR24]. We report the performance of our radix sort protocol $\Pi_{\mathsf{Radix\text{-}Sort}}$ for 32-bit strings. We observe that the majority of the overhead comes from the generation of the 16 permutation correlations that the protocol requires. For these protocols, the main overhead is the generation of the permutation,

[3] Although, the Gazelle parameters do not target ciphertext privacy, and therefore may have some leakage on π.

and therefore, we expect a proportional slowdown if $\Pi_{\textsf{Prf-Perm}}$ with [APRR24] is replaced by an alternative.

Lastly, we implement our batched RAM protocol and report the performance of reading n memory locations out of a list of size n. The read elements are of size $\ell = 128$. The vast majority of the overhead associated with this protocol is the invocation of $\mathcal{F}_{\textsf{Stable-Sort}}$ on n elements of size $\log(n)$ bits. We implement $\mathcal{F}_{\textsf{Stable-Sort}}$ using $\Pi_{\textsf{Radix-Sort}}$.

References

AFL+16. Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. CCS '16, page 805–817, New York, NY, USA, 2016. Association for Computing Machinery.

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

AKK+23. Pranav Shriram A, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Ruffle: Rapid 3-party shuffle protocols. In *PoPETS 2023*, pages 24–42, 2023.

APRR24. Navid Alamati, Guru Vamsi Policharla, Srinivasan Raghuraman, and Peter Rindal. Improved alternating moduli prfs and post-quantum signatures. In *Advances in Cryptology – CRYPTO*, 2024.

ARS+15. Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 430–454. Springer, Heidelberg, April 2015.

BCG+19a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 291–308, New York, NY, USA, 2019. Association for Computing Machinery.

BCG+19b. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 489–518, Cham, 2019. Springer International Publishing.

BCG+22. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 603–633, Cham, 2022. Springer Nature Switzerland.

BCG+23. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Oblivious transfer with constant computational overhead. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 271–302, Cham, 2023. Springer Nature Switzerland.

BCP16. Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram and applications. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography*, pages 175–204, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

BDG$^+$22. Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. Secret-shared joins with multiplicity from aggregation trees. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 209–222, New York, NY, USA, 2022. Association for Computing Machinery.

Ben64. V. E. Benes. Optimal rearrangeable multistage connecting networks. *The Bell System Technical Journal*, 43(4):1641–1656, 1964.

BGI15. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 337–367, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

BIP$^+$18. Dan Boneh, Yuval Ishai, Alain Passelègue, Amit Sahai, and David J. Wu. Exploring crypto dark matter: New simple PRF candidates and their applications. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018, Part II*, volume 11240 of *LNCS*, pages 699–729. Springer, Heidelberg, November 2018.

CFL$^+$24. Xinle Cao, Weiqi Feng, Jian Liu, Jinjin Zhou, Wenjing Fang, Lei Wang, Quanqing Xu, Chuanhui Yang, and Kui Ren. Towards practical oblivious map. Cryptology ePrint Archive, Paper 2024/1650, 2024.

CGP20. Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *Advances in Cryptology – ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III*, page 342–372, Berlin, Heidelberg, 2020. Springer-Verlag.

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, Paper 2019/695, 2019. https://eprint.iacr.org/2019/695.

CSSW24. Gowri R. Chandran, Thomas Schneider, Maximilian Stillger, and Christian Weinert. Concretely efficient private set union via circuit-based PSI. *IACR Cryptol. ePrint Arch.*, page 1494, 2024.

DEs16. Jack Doerner, David Evans, and abhi shelat. Secure stable matching at scale. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1602–1613. ACM Press, October 2016.

DGH$^+$21. Itai Dinur, Steven Goldfeder, Tzipora Halevi, Yuval Ishai, Mahimna Kelkar, Vivek Sharma, and Greg Zaverucha. MPC-friendly symmetric cryptography from alternating moduli: Candidates, protocols, and applications. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 517–547, Virtual Event, August 2021. Springer, Heidelberg.

DS17. Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Com-*

munications Security, CCS '17, page 523–535, New York, NY, USA, 2017. Association for Computing Machinery.

DWA$^+$21. F. Betül Durak, Chenkai Weng, Erik Anderson, Kim Laine, and Melissa Chase. Precio: Private aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Paper 2021/1490, 2021.

FO20. Brett Hemenway Falk and Rafail Ostrovsky. Secure merge with $O(n \log \log n)$ secure operation. Cryptology ePrint Archive, Report 2020/807, 2020. https://eprint.iacr.org/2020/807.

GI14. Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to prove all NP-statements in zero-knowledge, and a methodology of cryptographic protocol design. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 171–185. Springer, Heidelberg, August 1987.

GRR24. Gayathri Garimella, Srinivasan Raghuramam, and Peter Rindal. Distributional secure merge. Cryptology ePrint Archive, Paper 2024/1048, 2024.

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, *Information Security and Cryptology – ICISC 2012*, pages 202–216, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

HKO23. David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. Tri-state circuits - A circuit model that captures RAM. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 128–160. Springer, 2023.

HZF$^+$22. Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiang-Yang Li. Scape: Scalable collaborative analytics system on private database with malicious security. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 1740–1753. IEEE, 2022.

JVC18. Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1651–1669. USENIX Association, 2018.

KLS24. Jiseung Kim, Hyung Tae Lee, and Yongha Son. Revisiting shuffle-based private set unions with reduced communication. Cryptology ePrint Archive, Paper 2024/1560, 2024.

Klu22. Kamil Kluczniak. Circuit privacy for fhew/tfhe-style fully homomorphic encryption in practice. Cryptology ePrint Archive, Paper 2022/1459, 2022. https://eprint.iacr.org/2022/1459.

KMPP24. Banashri Karmakar, Shyam Murthy, Arpita Patra, and Protik Paul. Quickpool: Privacy-preserving ride-sharing service. *IACR Cryptol. ePrint Arch.*, page 1109, 2024.

Lin16. Yehuda Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Paper 2016/046, 2016. https://eprint.iacr.org/2016/046.

LO13. Steve Lu and Rafail Ostrovsky. How to garble ram programs? In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 719–734, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

LO17. Steve Lu and Rafail Ostrovsky. Black-box parallel garbled ram. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 66–92, Cham, 2017. Springer International Publishing.

LWDY24. Qiyao Luo, Yilei Wang, Wei Dong, and Ke Yi. Secure query processing with linear complexity. *CoRR*, abs/2403.13492, 2024.

MAB$^+$10. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

MR18. Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 35–52, New York, NY, USA, 2018. Association for Computing Machinery.

MRR20. Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and PSI for secret shared data. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1271–1287. ACM Press, November 2020.

NWI$^+$15. Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*, pages 377–394. IEEE Computer Society Press, May 2015.

Ost24. Benjamin Ostrovsky. Privacy-preserving dijkstra. Cryptology ePrint Archive, Paper 2024/988, 2024.

PLS23. Andrew Park, Wei-Kai Lin, and Elaine Shi. Nanogram: Garbled ram with o(log n) overhead. page 456–486, Berlin, Heidelberg, 2023. Springer-Verlag.

PR. Lance Roy Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`.

RRT23. Srinivasan Raghuraman, Peter Rindal, and Titouan Tanguy. Expand-convolute codes for pseudorandom correlation generators from lpn. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 602–632, Cham, 2023. Springer Nature Switzerland.

SYB$^+$23. Xiangfu Song, Dong Yin, Jianli Bai, Changyu Dong, and Ee-Chien Chang. Secret-shared shuffle with malicious security. Cryptology ePrint Archive, Paper 2023/1794, 2023.

TN21. Hikaru Tsuchida and Takashi Nishide. Private decision tree evaluation with constant rounds via (only) fair SS-4PC. In Joonsang Baek and Sushmita Ruj, editors, *ACISP 21*, volume 13083 of *LNCS*, pages 309–329. Springer, Heidelberg, December 2021.

WCS15. Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 850–861, New York, NY, USA, 2015. Association for Computing Machinery.

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

Disclaimer