Private Set Operations from Oblivious Switching[*]

Gayathri Garimella[1], Payman Mohassel[2], Mike Rosulek[1], Saeed Sadeghian[3], and Jaspal Singh[1]

[1]Oregon State University
[2]Facebook
[3]Security Compass

March 1, 2021

Abstract

Private set intersection reveals the intersection of two private sets, but many real-world applications require the parties to learn only *partial information* about the intersection. In this paper we introduce a new approach for computing arbitrary functions of the intersection, provided that it is safe to also reveal the cardinality of the intersection. In the most general case, our new protocol provides the participants with secret shares of the intersection, which can be fed into any generic 2PC protocol. Certain computations on the intersection can also be done even more directly and efficiently, avoiding this secret-sharing step. These cases include computing *only* the cardinality of intersection, or the "cardinality-sum" application proposed in Ion *et al.* (ePrint 2017). Compared to the state-of-the-art protocol for computing on intersection (Pinkas et al., Eurocrypt 2019), our protocol has about $2.5-3\times$ less communication, and has faster running time on slower (50Mbps) networks.

Our new techniques can also be used to privately compute the *union* of two sets as easily as computing the intersection. Our protocol concretely improves the leading private set union protocol (Kolesnikov et al., Asiacrypt 2020) by a factor of $2-2.5\times$, depending on the network speed. We then show how private set union can be used in a simple way to realize the "Private-ID" functionality suggested by Buddhavarapu et al. (ePrint 2020). Our protocol is significantly faster than the prior Private-ID protocol, especially on fast networks.

All of our protocols are in the two-party setting and are secure against semi-honest adversaries.

1 Introduction

In 2-party private set intersection (PSI), Alice's input is a set of items X, Bob's input is a set Y, and the output (given to one or both of them) is the **entire contents of the intersection** $X \cap Y$. PSI protocols have become incredibly efficient over the last decade.

The fastest PSI protocols generally follow the rough approach of Pinkas et al. [PSZ14], which was the first special-purpose PSI protocols to be based on efficient OT (oblivious transfer) extension. Since then, the techniques have been considerably refined and improved for both semi-honest [PSSZ15, KKRT16, PRTY19, CM20] and malicious [DCW13, RR17a, RR17b, PRTY20] security. An entirely different approach to PSI requires public-key operations (*e.g.*, key agreement or partially homomorphic encryption) linear in the size of the sets [Mea86, HFH99, FNP04, CT10, CT12, FHNP16]. Our focus in this work is on faster OT-extension-based PSI techniques.

Computing on the Intersection. Many real-world applications are closely related to PSI but in fact require only **partial/aggregate information about the intersection** to be revealed. In a notable real-world deployment of secure computation, Google is known to compute the cardinality of the intersection and

[*]authors from Oregon State University - Partially supported by NSF award 1617197 and a Facebook research award

the sum of values in the intersection [IKN+19, MPR+20]. More generally, we consider **private computing on set intersection (PCSI)**: the problem of securely computing $g(X \cap Y)$ for a (mostly) generic choice of function g.

There are several techniques for computing set intersections within generic 2PC, so that the intersection can be easily fed into another function. Huang, Katz & Evans [HEK12] gave an efficient sort-compare-shuffle circuit for use in either GMW or Yao's protocol. Further combinatorial improvements to intersection circuits were proposed in [PSSZ15, PSWW18]. The current state of the art for PCSI is due to [PSTY19], using a special-purpose preprocessing phase before using general-purpose 2PC to perform the necessary comparisons.

Why the Performance Gap? Plain PSI and PCSI are clearly closely related problems, and yet the state-of-the-art protocols for these problems have significantly different efficiency. Semi-honest PCSI – even in the simplest possible cases, like cardinality of intersection – is concretely about $20\times$ slower and requires over $30\times$ more communication than semi-honest PSI. Why is this the case?

All PSI and PCSI protocols use various combinatorial techniques to reduce the problem to a series of private equality tests. A private equality test (PEqT) takes a private string from each party and reveals (only) whether the strings are identical.

In the case of PSI, each party is allowed to learn whether each of their input items is in the intersection or not. This fact leads PSI protocols to use efficient, special-purpose PEqT subprotocols, which reveal the output of the equality test directly to at least one of the parties. This approach doesn't immediately work for PCSI, since in that case the participants should not learn whether a particular item is in the intersection or not. Instead, the outcome of the PEqTs should remain "inside the secure computation," prompting PCSI protocols to implement PEqTs simply as circuits within a general-purpose 2PC protocol.

These divergent choices of PEqTs lead to the differences in performance between PSI and PCSI. A general-purpose PEqT on ℓ-bit strings is a boolean circuit with ℓ non-free gates, leading to $O(\ell)$ cryptographic operations and $O(\ell\kappa)$ bits of communication. The state-of-the-art for special-purpose PEqTs [KKRT16] has cost that is *independent* of ℓ: only $O(\kappa)$ bits of communication and $O(1)$ symmetric-key cryptographic operations per equality test.

One exception to this general rule is due to Ciampi & Orlandi [CO18]. They provide a special-purpose PEqT (actually a generalization where one party has m items and the other has 1) that produces outputs in "encrypted form" that can be subsequently fed into a generic 2PC. However, their approach still requires $\Theta(\kappa\ell)$ bits of communication per comparison. While their concrete constants are smaller than a circuit-based comparison, their approach is not an asymptotic improvement.

Other Related Work. Another body of work studies the special case of computing the *cardinality* of intersection [HFH99, VC05, CZ09, CGT12, EFG+15, BA12, KS05, DD15]. It is not clear how to extend such results for computing more general functions of the intersection. The work of [BA12, EFG+15, MRR19] is in the multi-party setting ($n \geq 3$ parties) with an honest majority based on secret-sharing. As a result, no cryptographic operations are needed but the techniques are not applicable to the two-party setting.

1.1 Our contribution

We describe a new approach for semi-honest PCSI, which leaks the cardinality $|X \cap Y|$. Hence, our protocol works to compute $g(X \cap Y)$ for any g that leaks the cardinality $|X \cap Y|$.This class of g includes many applications of interest, discussed below.

The main idea is to obliviously permute all of the strings that will be used in the PEqTs, so that one party does not know which items are tested in which PEqT instance. We can then use the more efficient special-purpose PEqTs, giving output directly to the party who is oblivious to the permutation. This reveals only the cardinality of the intersection (*i.e.*, how many PEqTs give output TRUE).

Obliviously permuting n items incurs a $\log n$ overhead. However, in return for this extra cost we are able to replace general-purpose PEqTs with special-purpose PEqTs, saving a factor of ℓ (for strings of length ℓ). In almost all situations, $\log n \ll \ell$ and the tradeoff is an asymptotic as well as concrete improvement over the state of the art.

Extensions and Applications Our protocol supports any symmetric function $g(X \cap Y)$ that leaks $|X \cap Y|$. Useful such functions include:

- Computing the intersection; *i.e.*, PSI (although our protocol is not competitive with the most efficient PSI-only protocols).
- Computing *only* the cardinality of the intersection.
- Computing secret shares of the items in the intersection.
- The "intersection-sum" functionality proposed in [IKN+19], in which Alice has a set of keys $\{x_1, \ldots, x_n\}$ and Bob has a set of key-value pairs $\{(y_1, v_1), \ldots, (y_n, v_n)\}$. Both parties learn the cardinality of $\{x_1, \ldots, x_n\} \cap \{y_1, \ldots, y_n\}$ as well as the sum of values $\sum_{i : y_i \in \{x_1, \ldots, x_n\}} v_i$. Although not strictly an instance of PCSI as we have defined it, our protocol is easily modified to realize this functionality.

For all of these cases except the plain-PSI case, our protocol gives the most concretely efficient solution to date.

We also show how to use our main techniques to also securely compute the **union** of the input sets. Our private set union protocol is concretely more efficient than the state-of-the-art protocol of [KRTW19].

Finally, we show how our techniques can be used to realize the "private ID" functionality proposed in [BKM+20]. In this functionality, both parties learn pseudorandom universal identifiers for the values in the union of their sets, as well as the identifiers corresponding to their own items. This functionality allows parties to locally sort their data sets according to these universal identifiers, and feed them into any general-purpose 2PC protocol for simplified processing. Our construction is the first instantiation of Private ID using OT-based techniques that are dominated by symmetric-key crypto operations.

We have implemented our protocols and give a full comparison to existing protocols.

2 Preliminaries

Security Model. We use the standard notion of security in the presence of semi-honest adversaries. Let π be a protocol for computing the function $f(x_1, x_2)$, where party P_i has input x_i. We define security in the following way.

For each party P, let $\text{VIEW}_P(1^\kappa, x_1, x_2)$ denote the view of party P during an honest execution of π on inputs x_1 and x_2. The view consists of P's input, random tape, and all messages exchanged as part of the π protocol.

Definition 1. *2-party protocol π securely realizes f in the presence of semi-honest adversaries if there exists a simulator Sim such that, for all inputs x_1, x_2 and all $i \in \{1, 2\}$:*

$$\mathsf{Sim}(1^\kappa, i, x_i, f(x_1, x_2)) \cong_\kappa \text{VIEW}_{P_i}(1^\kappa, x_1, x_2)$$

where $\cong_\kappa$ denotes computational indistinguishability with respect to security parameter κ.

Essentially, a protocol is secure if the view of a party leaks no more information than $f(x_1, x_2)$.

3 Protocol Building Blocks

3.1 Oblivious Transfer

Oblivious Transfer (OT) is a fundamental cryptographic protocol widely used in secure computation, and initially introduced in [Rab05]. It allows a sender with two inputs m_0, m_1 and a receiver with a bit b to engage in a protocol where the receiver learns m_b, and neither party learns any additional information. A single OT requires public-key operations and hence is expensive. But a powerful technique called OT extension [IKNP03, KK13, ALSZ13] allows one to perform n OTs by only performing $O(\kappa)$ public-key operations (where κ is a computational security parameter) and $O(n)$ fast symmetric-key operations, allowing for faster and more scalable implementation when invoking many OTs. In Figure 1 we formally define the ideal functionality for OT that provides n parallel instances of OT.

3.2 Oblivious Switching Network

An oblivious switching network works as follows. One party chooses a permutation π on n items, and the other party chooses a vector $\vec{x}$. The parties learn additive secret shares of $\pi(\vec{x})$ (*i.e.*, $\vec{x}$ permuted according to π). The formal description of the functionality is given in Figure 2.

Mohassel & Sadeghian [MS13] introduced oblivious switching and described a semi-honest oblivious switching protocol that is based on oblivious transfers. Briefly, the protocol works by considering a *universal switching network* (*i.e.*, Waksman or Beneš network), which consists of $O(n \log n)$ 2-input, 2-output switches. The receiver chooses programming of the switches (whether to swap the order of the inputs or not) based on their permutation π. The sender chooses a random one-time pad for each wire of the network, and the invariant is that the receiver will learn the value on each wire but masked with the one-time pad of that wire. The parties use oblivious transfer to allow the receiver to select whether to learn the XOR of masks of input b and output b, or to learn the XOR of masks of input b and output $1-b$. These XOR values suffice to preserve the invariant across the switches. At the output layer of the switching network, the sender holds a vector of one-time pads, and the receiver holds the permuted values masked by these one-time pads. We give more details in Appendix A.

The total cost of the switching network is $O(n \log n)$ oblivious transfers, one for every switch in the switching network. Each OT is on a pair of 2ℓ-bit strings (two masks).

We described the ideal functionality to allow the input vector $\vec{x}$ to be longer than the output (secret-shared) vectors, which leads to π being an *injective function* rather than a permutation. This can be accomplished by simply permuting the input vector so that the desired items are "in the front", and then both parties truncating their vector of shares by the appropriate amount. In Appendix B we describe an optimization for injective functions that slightly improves over permuting-then-discarding.

3.3 Batch Oblivious PRF

Kolesnikov et al. [KKRT16] describe an efficient protocol for **batched oblivious PRF** (OPRF) based on OT extension. The protocol provides a batch of oblivious PRF instances in the following way. In the ith instance, the receiver has an input x_i; the sender learns a PRF seed k_i and the receiver learns $\mathsf{PRF}(k_i, x_i)$. Note that the receiver learns the output of the PRF on only one value per key, and the sender does not learn which output the receiver learned. The batch OPRF functionality is described formally in Figure 3.

The KKRT batch OPRF protocol is based on OT extension and extremely fast. Each OPRF instance requires roughly only 4.5κ total bits of communication between the parties, and a few calls to a hash function. On a fast network, a million OPRF instances can be generated in just a few seconds.

Technically speaking, the KKRT protocol realizes OPRF instances where the keys k_i are related in some sense. However, the PRF that it instantiates has all the expected security properties, even in the presence of such related keys. For the sake of simplicity, we ignore this issue in our notation. For more details, see [KKRT16].

3.4 Private Equality Tests

A private equality test (PEqT) allows two parties to determine whether their two input strings are equal (while leaking nothing else about the inputs).

An oblivious PRF can be used to realize a secure equality test in a simple way. Suppose Alice has input x and Bob has input y, and they would like to learn whether $x = y$. Alice acts as OPRF receiver with input x and learns $\mathsf{PRF}(k, x)$. Bob learns PRF seed k and sends the value $\mathsf{PRF}(k, y)$. If $x \neq y$ then the PRF property ensures that Bob's message looks random to Alice; otherwise the message is the PRF output that Alice already knows.

Using the batch OPRF protocol of [KKRT16], the parties can realize a large batch of equality tests in a natural way. The functionality $\mathcal{F}_{\mathsf{bEQ}}$ of Figure 4 formalizes this batch equality testing. We take advantage of the fact that its output can be given to just one party.

Parameters: number of OTs n; payload length ℓ.

On input $(m_{1,0}, m_{1,1}), \ldots, (m_{n,0}, m_{n,1})$ from the sender, where each $m_{i,b} \in \{0,1\}^\ell$, and input $\vec{b} \in \{0,1\}^n$ from the receiver:

1. Give output $(m_{1,b_1}, m_{2,b_2}, \ldots, m_{n,b_n})$ to the receiver.

Figure 1: Ideal functionality $\mathcal{F}_{\mathsf{ot}}$ for n oblivious transfers.

Parameters: input length n_{in}; output length $n_{\mathsf{out}} \le n_{\mathsf{in}}$; item length ℓ.

On input an injective function $\pi : [n_{\mathsf{out}}] \to [n_{\mathsf{in}}]$ from the receiver, and vector $\vec{x} \in (\{0,1\}^\ell)^{n_{\mathsf{in}}}$ from the sender:

1. Choose uniform vector $\vec{a} \leftarrow (\{0,1\}^\ell)^{n_{\mathsf{out}}}$.
2. Define the vector $\vec{b} \in (\{0,1\}^\ell)^{n_{\mathsf{out}}}$ via $b_i = a_i \oplus x_{\pi(i)}$.
3. Give output $\vec{a}$ to the sender and $\vec{b}$ to the receiver.

Figure 2: Ideal functionality $\mathcal{F}_{\mathsf{osn}}$ for oblivious switching network.

Parameters: batch size n; suitable PRF PRF.

On input vector $\vec{x} \in (\{0,1\}^*)^n$ from the receiver:

1. For each $i \in [n]$, choose uniform PRF key k_i.
2. For each $i \in [n]$, define $f_i = \mathsf{PRF}(k_i, x_i)$.
3. Give vector $\vec{k}$ to the sender and vector $\vec{f}$ to the receiver.

Figure 3: Ideal functionality $\mathcal{F}_{\mathsf{bOPRF}}$ for batch oblivious PRF.

Parameters: Number n of string-pairs to compare.

On input $\vec{x} \in (\{0,1\}^*)^n$ from the sender, and $\vec{y} \in (\{0,1\}^*)^n$ from the receiver:

1. Define the vector $\vec{e} \in \{0,1\}^n$, where $e_i = 1$ if $x_i = y_i$ and $e_i = 0$ otherwise.
2. Give $\vec{e}$ to the receiver.

Figure 4: Ideal functionality $\mathcal{F}_{\mathsf{bEQ}}$ for batch string equality testing.

3.5 Reducing PSI to $O(n)$ Comparisons

The leading protocol for PCSI is due to Pinkas et al. [PSTY19]. One of their main contributions is to show how to *interactively* reduce a PSI computation to $O(n)$ comparisons, using only a linear amount of communication.

The main idea behind the PSTY19 preprocessing is for Alice to use hash functions h_1, h_2, h_3 to assign her items to m bins via Cuckoo hashing, so that each bin has at most one item. Bob assigns each of his items y to all of the bins $h_1(y), h_2(y), h_3(y)$. The parties use the batch OPRF functionality $\mathcal{F}_{\mathsf{bOPRF}}$, with Alice acting as receiver. If she has placed item x in bin j, then she will receive output $\mathsf{PRF}(k_j, x)$, while Bob learns each k_j.

Now, Bob chooses a random value s_j for each bin j. The goal is to arrange that if Alice and Bob have a matching item in the jth bin, then Alice will somehow learn that bin's s_j value. Suppose for example that one of Bob's items in bin #1 is y^*. Then Bob needs to somehow communicate to Alice "if you have y^* in bin #1, then XOR your PRF output with $\mathsf{PRF}(k_1, y^*) \oplus s_1$". But he needs to do so without revealing y^* and the rest of his input items. He can do this by interpolating a polynomial P with the following property: if Bob has item y in bin j, then $P(y\|j) = \mathsf{PRF}(k_j, y) \oplus s_j$. Using the pseudorandomness of PRF and the randomness of the s_j values, it is possible to show that P is indistinguishable from a uniformly random polynomial, and hence it hides Bob's y-values.

Alice therefore can take her $\mathsf{PRF}(k_j, x)$ values and XOR with $P(y\|j)$. In the case that Bob also had this item x, then he would have assigned it to bin j (and to other bins as well), so Alice's result is s_j. If Bob did not have this x, then it is possible to show that Alice's result matches s_j with negligible probability (assuming the polynomial is over a sufficiently large field).

Overall, Alice obtains a vector of values (call them $t_1, \ldots, t_m$) where $t_j = s_j$ if and only if Alice's item in the jth bin is in the intersection. Hence we have reduced the problem of intersection to the problem of $m = O(n)$ string equality tests. These pairs of strings must be compared privately, since comparing them in the clear leaks information to both parties.

More details. We write Cuckoo hashing with the following notation:

$$\mathcal{C} \leftarrow \mathsf{Cuckoo}^m_{h_1,h_2,h_3}(X)$$

This expression means to hash the items of X into m bins using Cuckoo hashing on hash functions $h_1, h_2, h_3 : \{0,1\}^* \to [m]$. The output is $\mathcal{C} = (C_1, \ldots, C_m)$, where for each $x \in X$ there is some $i \in \{1,2,3\}$ such that $C_{h_i(x)} = x\|i$.[1] Some positions of $\mathcal{C}$ will not matter, corresponding to empty bins.

Using this notation, the PSTY19 preprocessing is as follows:

1. Alice does $\mathcal{A} \leftarrow \mathsf{Cuckoo}^m_{h_1,h_2,h_3}(X)$.
2. The parties call $\mathcal{F}_{\mathsf{bOPRF}}$, where Alice is receiver with input $\mathcal{A}$ and Bob is sender. Bob receives output $(k_1, \ldots, k_m)$ and Alice receives output $(f_1, \ldots, f_m)$. For each $x \in X$ assigned to bin j by hash function i, we have $f_j = \mathsf{PRF}(k_j, x\|i)$.
3. For each $j \in [m]$, Bob choose a random s_j. He then interpolates a polynomial P of degree $< 3n$ such that for every $y \in Y$ and $i \in \{1,2,3\}$:

   $$P(y\|i) = s_{h_i(y)} \oplus \mathsf{PRF}(k_{h_i(y)}, y\|i)$$

   He sends P to Alice.
4. Alice computes a vector $(t_1, \ldots, t_m)$ where $t_j = P(\mathcal{A}_j) \oplus f_j$.

Mega-Bin Optimization. The PSTY19 approach requires parties to interpolate and evaluate a polynomial of degree $3n$, where n can be very large (*e.g.*, $n = 2^{20}$). The fastest algorithms for interpolating such a polynomial (and evaluating it on n points) runs in $O(n \log^2 n)$ time. The cost of such polynomial operations can be prohibitive, so the authors of PSTY19 propose an alternative way to encode the same information.

Call a mapping "$y\|i \mapsto s_{h_i(y)} \oplus \mathsf{PRF}(k_{h_i(y)}, y\|i)$" a **hint**. Bob must convey $3n$ such hints to Alice in the protocol. One way to do this is to make $n' = n/\log n$ so-called *mega-bins* and assign each hint into a mega-bin using a hash function — *i.e.*, assign the hint for $y\|i$ to the mega-bin indexed $H(y\|i)$ for a public random function $H : \{0,1\}^* \to [n']$. With these parameters, all mega-bins hold fewer than $O(\log n)$ items, with overwhelming probability. Bob adds dummy hints to each mega-bin so that all mega-bins contain the worst-case $O(\log n)$ number of hints (since the number of "real" hints per mega-bin leaks information about his input set). In each mega-bin, Bob interpolates a polynomial over the hints in that bin, and sends all the polynomials to Alice. For each $x\|i$ held by Alice, she can find the corresponding hint (if it exists) in the polynomial for the corresponding mega-bin.

The total communication cost is a degree-$O(\log n)$ polynomial for each of $n/\log n$ mega-bins; in other words, a constant-factor increase over sending a single degree-$3n$ polynomial. However, the total computation cost is an interpolation of a degree-$O(\log n)$ polynomial in each mega-bin, a total cost of $O\Big((n/\log n)(\log n)(\log\log n)^2\Big) =$

[1]Appending the index of the hash function is helpful for dealing with edge cases like $h_1(x) = h_2(x)$, which happen with non-negligible probability.

$O(n(\log\log n)^2)$. In practice, the mega-bins are small enough that the asymptotically inferior quadratic polynomial interpolation algorithm is preferable, but this still leads to $O(n \log n)$ computational cost overall.

For simplicity, we describe our protocol in terms of the simpler single-polynomial solution, while our implementations use the mega-bins optimization.

4 Protocol Overviews and Details

In this section we give the details of our protocols for PCSI and related problems.

4.1 Our Protocol Core: Permuted Characteristic

All of our protocols build on the same core, which roughly consists of: (1) the PSTY19 preprocessing, reducing the intersection computation to $O(n)$ string equality tests; (2) an oblivious shuffle; (3) special-purpose equality tests.

We formalize this "protocol core" in terms of a **permuted characteristic** functionality $\mathcal{F}_{\mathsf{pc}}$ defined in Figure 5. Roughly speaking, the sender Alice learns a permutation π of her items, and the receiver Bob learns a vector $\vec{e}$, where $e_i = 1$ if Alice's $\pi(i)$'th item is in Bob's set. In other words, $\vec{e}$ is the characteristic vector of Alice's (permuted) set with respect to the intersection.

Our protocol for permuted characteristic is given formally in Figure 6.

Lemma 2. *The protocol in Figure 6 securely realizes $\mathcal{F}_{\mathsf{pc}}$ against semi-honest adversaries.*

Proof. Alice's view consists of her input, private randomness $\widetilde{\pi}$, outputs from $\mathcal{F}_{\mathsf{bOPRF}}$ and $\mathcal{F}_{\mathsf{osn}}$, and protocol message P from Bob. The simulator for a corrupt Alice runs the protocol honestly with the following changes:

- In step 2, it simulates uniform outputs f_j from $\mathcal{F}_{\mathsf{bOPRF}}$.
- In step 4, it simulates a uniform polynomial P from Bob.
- In step 6, it chooses $\widetilde{\pi}$ so that $x_{\pi(i)} = \mathcal{A}_{\widetilde{\pi}(i)}$, where π is the ideal output from $\mathcal{F}_{\mathsf{pc}}$.

We show that this simulation is correct via the sequence of hybrids:

- *Hybrid 0.* The real interaction, in which Bob runs honestly with his input set Y.
- *Hybrid 1* The only change is that all terms of the form $\mathsf{PRF}(k_j, \cdot)$ are replaced with uniform values, including Alice's outputs from the $\mathcal{F}_{\mathsf{bOPRF}}$ functionality in step 2. This change is indistinguishable by the pseudorandomness of PRF.
- *Hybrid 2* The only change is that in step 4 the polynomial P is chosen uniformly at random. Previously, P was interpolated through points of the form $s_{h_i(y)} \oplus \mathsf{PRF}(k_{h_i(y)}, y\|i)$. If Alice didn't have item y or didn't place item y according to hash function i, then the PRF-output term has been replaced by a random term that is independent of her view, so this output of P is uniform. For all other outputs of P (corresponding to Alice's placement of intersection items), the corresponding s_j values are uniform, making those P-outputs uniform as well. Overall, P is being interpolated to give only uniform outputs; hence P itself is distributed uniformly among polynomials of degree $< 3n$. Hence this change in hybrids has no effect on Alice's view.
- *Hybrid 3* In the previous hybrid, Alice first chooses injective function $\tilde{\pi}$ and then uses it to compute permutation π. This induces a uniform distribution on π, so the same distribution can be obtained by first choosing uniform π and then computing the corresponding $\tilde{\pi}$.

The final hybrid corresponds to the simulator as described above.

Bob's view consists of his input, private randomness $\{s_j\}_j$, outputs from $\mathcal{F}_{\mathsf{bOPRF}}$, $\mathcal{F}_{\mathsf{osn}}$, $\mathcal{F}_{\mathsf{bEQ}}$. Clearly the outputs k_i from $\mathcal{F}_{\mathsf{bOPRF}}$ are distributed independently of the honest party's inputs. By definition, the output $\vec{b}$ from $\mathcal{F}_{\mathsf{osn}}$ is uniformly distributed, as a secret-share. This leaves only the output $\vec{e}$ of $\mathcal{F}_{\mathsf{bEQ}}$. It is a simple matter to check that $\vec{e}$ is distributed exactly as the ideal output of $\mathcal{F}_{\mathsf{pc}}$. Namely, it is a uniform bit-vector with exactly $|X \cap Y|$ ones. Hence, all of Bob's view can be trivially simulated given the ideal output $\vec{e}$ from $\mathcal{F}_{\mathsf{pc}}$. □

Parameters: Number of items n for the sender and receiver.

On input $X = \{x_1, \ldots, x_n\} \subseteq \{0,1\}^*$ from the sender and $Y = \{y_1, \ldots, y_n\} \subseteq \{0,1\}^*$ from the receiver, with $|X| = |Y| = n$:

1. Choose a random permutation π over $[n]$.
2. Define the vector $\vec{e} \in \{0,1\}^n$, where $e_i = 1$ if $x_{\pi(i)} \in Y$ and $e_i = 0$ otherwise.
3. Give the vector π to the sender and give $\vec{e}$ to the receiver.

Figure 5: Permuted Characteristic Functionality $\mathcal{F}_{\mathsf{pc}}$.

Parameters: Size n of input sets. Cuckoo hashing parameters: hash functions h_1, h_2, h_3 and number of bins m.

Inputs: $X = \{x_1, \ldots, x_n\} \subseteq \{0,1\}^*$ from Alice; $Y = \{y_1, \ldots, y_n\} \subseteq \{0,1\}^*$ from Bob.

Protocol:

1. **(PSTY19 preprocessing)** Alice does $\mathcal{A} \leftarrow \mathsf{Cuckoo}^m_{h_1,h_2,h_3}(X)$.
2. The parties call $\mathcal{F}_{\mathsf{bOPRF}}$, where Alice is receiver with input $\mathcal{A}$ and Bob is sender. Bob receives output $(k_1, \ldots, k_m)$ and Alice receives output $(f_1, \ldots, f_m)$. Alice's output is such that, for each $x \in X$ assigned to bin j by hash function i, we have $f_j = \mathsf{PRF}(k_j, x\|i)$.
3. For each $j \in [m]$, Bob choose a random s_j.
4. Bob interpolates a polynomial P of degree $< 3n$ such that for every $y \in Y$ and $i \in \{1,2,3\}$, we have
$$P(y\|i) = s_{h_i(y)} \oplus \mathsf{PRF}(k_{h_i(y)}, y\|i)$$
He sends P to Alice.
5. Alice computes a vector $(t_1, \ldots, t_m)$ where $t_j = P(\mathcal{A}_j) \oplus f_j$.
6. **(Oblivious shuffle)** Recall that Alice places her n items into m bins, with each item placed exactly once. Alice chooses a random injective function $\widetilde{\pi} : [n] \to [m]$ such that $\widetilde{\pi}(1), \ldots, \widetilde{\pi}(n)$ are the non-empty bins of $\mathcal{A}$.
7. The parties invoke $\mathcal{F}_{\mathsf{osn}}$, where Alice acts as receiver with input $\widetilde{\pi}$ and Bob acts as sender with input $\vec{s}$. Alice receives output $\vec{a}$ and Bob receives output $\vec{b}$, where $a_i \oplus b_i = s_{\widetilde{\pi}(i)}$.
8. Alice locally computes vector $\vec{a}'$ as $a'_i = a_i \oplus t_{\widetilde{\pi}(i)}$, so that a'_i and b_i are secret shares of $s_{\widetilde{\pi}(i)} \oplus t_{\widetilde{\pi}(i)}$. i.e; , $a'_i = b_i$ whenever $s_{\widetilde{\pi}(i)} = t_{\widetilde{\pi}(i)}$.
9. **(Equality tests)** The parties invoke $\mathcal{F}_{\mathsf{bEQ}}$, where Alice is sender with input $\vec{a}'$ and Bob is receiver with input $\vec{b}$. Bob receives output $\vec{e}$.
10. Recall that each Cuckoo bin $\mathcal{A}_{\widetilde{\pi}(i)}$ holds some item $x_j \in X$. Define the permutation π on $[n]$ so that $x_{\pi(i)} = \mathcal{A}_{\widetilde{\pi}(i)}$. Alice gives output π and Bob gives output $\vec{e}$.

Figure 6: Permuted characteristic protocol.

4.2 Intersection and Union

Our protocol core (permuted characteristic) $\mathcal{F}_{\mathsf{pc}}$ can be used to realize plain **private set intersection (PSI)** and **private set union (PSU)** in a simple way. After $\mathcal{F}_{\mathsf{pc}}$, say Alice holds a permutation of her input set, and Bob holds the characteristic vector $\vec{e}$. If the characteristic vector is 0 in position i, this means that Alice's ith item is in $X \setminus Y$. If the characteristic vector is 1 in position i, then Alice's ith item is in

$X \cap Y$.

For PSI, the parties can use $n = |X|$ oblivious transfers to allow Bob to learn the items in $X \cap Y$. If $e_i = 1$, Bob will choose to learn Alice's ith item; otherwise he will choose to learn nothing.

Observe that PSU is equivalent to letting Bob learn $X \setminus Y$: Given the ideal PSU output $X \cup Y$ and Bob's input Y, he can indeed compute $X \setminus Y = (X \cup Y) \setminus Y$. Conversely, given $X \setminus Y$ and Bob's input Y, he can compute the PSU output $X \cup Y = (X \setminus Y) \cup Y$. With that in mind, Bob can easily compute $X \setminus Y$ by simply inverting his logic in the previous paragraph. If $e_i = 0$, Bob will choose to learn (via OT) Alice's ith item; otherwise he will choose to learn nothing.

The formal details of these PSI/PSU protocols are given in Figure 7. We remark that this approach for PSI is not competitive with the state-of-the-art special-purpose protocols for PSI. In particular, an oblivious shuffle is unnecessary for PSI. We include this PSI protocol merely for illustrative purposes. However, as we shall see, our approach for PSU is indeed competitive with the state of the art, and is useful as a stepping stone to another interesting application.

Parameters: Size of sets n.

On input $X = \{x_1, \ldots, x_n\} \subseteq \{0,1\}^*$ from the sender and $Y = \{y_1, \ldots, y_n\} \subseteq \{0,1\}^*$ from the receiver:

1. **[for intersection:]** give $X \cap Y$ to the receiver
2. **[for union:]** give $X \cup Y$ to the receiver

Figure 7: Ideal functionalites for intersection/union ($\mathcal{F}_{\mathsf{psi}}/\mathcal{F}_{\mathsf{psu}}$).

Parameters: Size of sets n.

Inputs: $X = \{x_1, \ldots, x_n\}$ for the sender; $Y = \{y_1, \ldots, y_n\}$ for the receiver.

Protocol:

1. Parties invoke $\mathcal{F}_{\mathsf{pc}}$ with inputs X, Y Sender obtains a permutation π. Receiver obtains characteristic vector $\vec{e}$.
2. Parties invoke n instances of OT via $\mathcal{F}_{\mathsf{ot}}$. The receiver uses $\vec{e}$ as the choice bits.
3. **[For intersection:]**
   (a) The sender uses input $(\bot, x_{\pi(i)})$ as input to the ith OT.
   (b) The receiver learns $\{x_{\pi(i)} \mid e_i = 1\} = X \cap Y$, which he outputs.
4. **[For union:]**
   (a) The sender uses input $(x_{\pi(i)}, \bot)$ as input to the ith OT.
   (b) The receiver learns $\{x_{\pi(i)} \mid e_i = 0\} = X \setminus Y$, he outputs $\{X \setminus Y\} \cup Y$.

Figure 8: Protocols for intersection and union.

Lemma 3. *The PSI and PSU protocols of Figure 8 securely realize $\mathcal{F}_{\mathsf{psi}}$ and $\mathcal{F}_{\mathsf{psu}}$, respectively, (Figure 7) against semi-honest adversaries.*

Proof sketch. We focus on the security proof for PSI, as the proof for PSU is analagous. Security against a corrupt sender is trivial, since their view consists of only the output π from $\mathcal{F}_{\mathsf{pc}}$. For a corrupt receiver, their view consists of the vector $\vec{e}$ and OT outputs. If $x_{\pi(i)} \in Y$, then $e_i = 1$ and the ith OT output is $x_{\pi(i)}$. Otherwise, $e_i = 0$ and the ith OT outputs is $\bot$. Furthermore, π is uniform, and therefore this distribution can be simulated given only ideal output $X \cap Y$: Sample a uniform binary vector $\vec{e}$ containing $|X \cap Y|$ 1s. Then choose a uniform assignment of elements of $X \cap Y$ to OT instances i for which $e_i = 1$. $\square$

Our protocols give output only to one party (the receiver). In the semi-honest setting, the receiver can simply report the output to the sender in order to provide output to both parties.

4.3 PCSI: Computing on the Intersection

We now discuss PCSI: computing a function of the intersection. Our approach inherently leaks the cardinality, and we formalize this in the ideal functionality $\mathcal{F}_{\mathsf{pcsi+card}}$ of Figure 9, which outputs the cardinality of the intersection along with a function g of the intersection.

Parameters: Size of sets n. Function g.

On input $X \subseteq \{0,1\}^*$ from the sender and $Y \subseteq \{0,1\}^*$ from the receiver:

1. Give $\Big(|X \cap Y|,\ g(X \cap Y)\Big)$ to the receiver.

Figure 9: Ideal functionality for computing cardinality and an arbitrary function of the intersection $\mathcal{F}^{g}_{\mathsf{pcsi+card}}$.

Perhaps the most common instance of PCSI is to compute *only* the cardinality (*i.e.*, g is empty). This special case can be obtained trivially by our $\mathcal{F}_{\mathsf{pc}}$ protocol core:

Proposition 1. *If the parties run $\mathcal{F}_{\mathsf{pc}}$ on their inputs and the receiver outputs the hamming weight of $\vec{e}$, then the resulting protocol securely realizes $\mathcal{F}^{g}_{\mathsf{pcsi+card}}$ for $g = \bot$, against semi-honest adversaries.*

Proof sketch. Security against corrupt sender is trivial since the sender's view consists only of a uniformly distributed permutation (*i.e.*, independent of anyone's inputs). Regarding a corrupt receiver: since π is uniformly chosen among permutations, the vector $\vec{e}$ is distributed as a uniform vector of length n with exactly $|X \cap Y|$ ones. This distribution can therefore be simulated given only the ideal output $|X \cap Y|$. □

Note also that if the sizes of X and Y are public, then computing $|X \cap Y|$ is equivalent to computing $|X \cup Y|$, via the standard inclusion-exclusion formula.

Cardinality-sum If the function g is simple enough, then $\mathcal{F}^{g}_{\mathsf{pcsi+card}}$ can be realized in a very simple way from $\mathcal{F}_{\mathsf{pc}}$. We illustrate with an example, which does not exactly fit into the definition of $\mathcal{F}_{\mathsf{pcsi+card}}$ since one party has a set of key-value pairs. Our example involves the **cardinality-sum** functionality proposed by Ion *et al.* [IKN+19]. The functionality is described formally in Figure 10. It reveals the intersection of the cardinality as well as the sum of all *values* whose *keys* are in the intersection.

In Figure 11 we describe a simple protocol realizing the cardinality-sum functionality. Similar to how we achieve PSI & PSU from $\mathcal{F}_{\mathsf{pc}}$, this protocol uses oblivious transfers to let the receiver learn things, based on the characteristic vector. In this case, instead of learning the sender's items in the clear, the receiver learns either an additive secret share of 0 or a secret share of that item's associated value. Then the receiver can compute the sum by locally adding the shares.

Lemma 4. *The protocol of Figure 11 securely realizes ideal functionality $\mathcal{F}_{\mathsf{card+sum}}$ (Figure 10), against semi-honest adversaries.*

Proof sketch. Security against a corrupt sender is immediate. Relative to the cardinality protocol, the only addition to a corrupt receiver's view are the outputs of the OTs. View these outputs as the vector $\vec{r} + \vec{q}$, where $\vec{r}$ is uniform subject to having sum 0; and $q_i = v_i$ if $x_i \in Y$ and $q_i = 0$ otherwise. Since the r_i's are a perfect additive secret share of 0, the distribution of $\vec{r} + \vec{q}$ depends only on $\sum_i q_i$, which is the ideal output s. □

General case. More generally, suppose the sender has a set of key-value pairs (x_i, v_i), and the receiver has a set of keys Y. The parties can use parallel oblivious transfers to secret share a vector $\vec{q}$, where:

$$q_i = \begin{cases} v_i & x_i \in Y \\ \tilde{v} & x_i \notin Y \end{cases}$$

Parameters: Size of sets n. Group $\mathbb{G}$.

On input $X = \{(x_1, v_1), \ldots, (x_n, v_n)\}$ from the sender, where each $v_i \in \mathbb{G}$, and input $Y \subseteq \{0,1\}^*$ from the receiver:

1. Give output (c, s) to the receiver, where:

$$c = \Big|\{x_1, \ldots, x_n\} \cap Y\Big| \qquad\qquad s = \sum_{x_i \in Y} v_i$$

Figure 10: Ideal functionality $\mathcal{F}_{\mathsf{card+sum}}$ for cardinality-sum.

Parameters: Size of sets n. Group $\mathbb{G}$.

Inputs: $X = \{(x_1, v_1), \ldots, (x_n, v_n)\}$ for the sender, where $v_i \in \mathbb{G}$; $Y \subseteq \{0,1\}^*$ for the receiver.

Protocol:

1. Parties invoke $\mathcal{F}_{\mathsf{pc}}$ with inputs $\{x_1, \ldots, x_n\}$ and Y. The Sender obtains a permutation π. Receiver obtains characteristic vector $\vec{e}$.
2. The sender chooses a random vector $(r_1, \ldots, r_n) \leftarrow \mathbb{G}^n$ such that $\sum_i r_i = 0$.
3. Parties invoke n instances of OT via $\mathcal{F}_{\mathsf{ot}}$. The receiver uses $\vec{e}$ as their choice bits. The sender uses input $(r_i, r_i + v_{\pi(i)})$ as input to the ith OT. The receiver gets output $\hat{v}_i$ from the ith OT.
4. The receiver outputs $\left(\sum_i e_i,\ \sum_i \hat{v}_i\right)$.

Figure 11: Protocol for cardinality-sum.

where $\tilde{v}$ is some dummy/default value. In the case of cardinality-sum, $\tilde{v} = 0$.

With secret shares of such a vector, the parties can compute a function g that takes in a vector of inputs and ignores the dummy/default values in the input. In the case of cardinality-sum, g was simple addition and no interaction was required to compute it.

4.4 Secret-Shared Intersection

In some settings, it is more convenient for the parties to obtain secret shares of the items of the intersection, so that it can be fed into a generic 2PC.

To illustrate the challenges here, let's first consider a very natural approach that *doesn't* work. The parties run $\mathcal{F}_{\mathsf{pc}}$, so that Bob learns the indices of Alice's intersection items, permuted according to the secret permutation π. Whereas with PSI/PSU, Bob used OT to selectively learn the items of the intersection (or set-difference), we might be tempted to have Bob now learn secret-shares of the items in the intersection.

To see why this isn't so straight forward, imagine that each party has 1 million items, and there are 10 in the intersection. Bob could indeed use OT to learn secret shares of those 10 items. But now it is time to run the 2PC to compute g on those 10 items. Alice prepared 1M additive shares, and she doesn't know which 10 of them should be given to g! Bob knows which ones are the right ones, but he can't tell Alice because she knows the secret permutation π — this would reveal the entire contents of the intersection to Alice!

We address this challenge by simply doing another oblivious switching network. Alice holds a secret permutation of her items. Bob knows which indices in this permutation correspond to items in the intersection. He chooses an injective function ρ whose range covers exactly those intersection items. They use an oblivious switching network, so that both parties learn additive shares of only those items referenced by ρ.

Details of this protocol are given in Figure 13. Bear in mind that the input to g is necessarily given as an ordered vector. Most applications of PCSI will involve a function g that is symmetric, meaning that

Parameters: Size of sets n.

On input $X \subseteq \{0,1\}^*$ from Alice and $Y \subseteq \{0,1\}^*$ from Bob:

1. Let $z_1, \ldots, z_k$ be a random permutation of $X \cap Y$
2. Choose $\vec{a}$ uniformly at random. Define $\vec{b}$ via $b_i = a_i \oplus z_i$. *I.e.*, $\vec{a}$ and $\vec{b}$ are secret shares of $(z_1, \ldots, z_k)$.
3. Give $\vec{a}$ to Alice and $\vec{b}$ to Bob.

Figure 12: Ideal functionality for computing secret shares of the intersection $\mathcal{F}_{\mathsf{ss\text{-}int}}$.

Parameters: Size of sets n.

Inputs: $X, Y \subseteq \{0,1\}^*$ for the sender and receiver, respectively.

Protocol:

1. Parties invoke $\mathcal{F}_{\mathsf{pc}}$ with inputs $\{x_1, \ldots, x_n\}$ and Y Sender obtains a permutation π. Receiver obtains characteristic vector $\vec{e}$.
2. Let c be the Hamming weight of $\vec{e}$. The receiver announces c to the sender.
3. The receiver chooses a random injective function $\rho : [c] \to [n]$ such that $e_{\rho(i)} = 1$ for all i.
4. The parties invoke $\mathcal{F}_{\mathsf{osn}}$ where the sender provides input $\{x_{\pi(1)}, \ldots, x_{\pi(n)}\}$ and the receiver provides input ρ. The result is that the parties hold secret shares of $(x_{\pi(\rho(1))}, \ldots, x_{\pi(\rho(c))})$.

Figure 13: Protocol for secret-shared intersection.

g is insensitive to the order of its inputs. However, note that the values that are fed into g are randomly permuted, from both parties' perspective (Bob didn't know π and Alice didn't know ρ). Hence, our protocol is meaningful even if g is sensitive to the order of its input items. In that case, we still achieve the most natural security, where the items of the intersection are randomly shuffled before being given as input to g.

Lemma 5. *The protocol of Figure 13 securely realizes $\mathcal{F}_{\mathsf{ss\text{-}int}}$ (Figure 12), against semi-honest adversaries.*

Proof. Beyond the output of $\mathcal{F}_{\mathsf{pc}}$, the only thing added to parties' views in Figure 13 is the cardinality c and the secret shares output by $\mathcal{F}_{\mathsf{osn}}$. The former can be inferred by the ideal output of $\mathcal{F}_{\mathsf{ss\text{-}int}}$, and the latter coincides with the ideal output itself. □

4.5 Private ID

Buddhavarapu *et al.* [BKM+20] proposed a useful functionality that they called **private-ID**. In this functionality, both parties provide a set of items. The functionality assigns to each item a truly random identifier (where identical items receive the same identifier). It then reveals to each party the identifiers corresponding to their own items, and also the entire set of *all* identifiers (*i.e.*, the identifiers of the union of their input sets).

The advantage of Private ID is that both parties can sort their private data relative to the global set of identifiers. They can then proceed item-by-item, doing any desired private computation, being assured that identical items are aligned.

Our approach. Our approach for private-ID builds on oblivious PRF and private set union. Roughly speaking, suppose the parties run an oblivious PRF twice: first, so that Alice learns k_A and Bob learns $\mathsf{PRF}(k_A, y_i)$ for each of his items y_i; and second so that Bob learns k_B and Alice learns $\mathsf{PRF}(k_B, x_i)$ for each of her items x_i. We will define the random identifier of an item x as

$$R(x) \stackrel{\text{def}}{=} \mathsf{PRF}(k_A, x) \oplus \mathsf{PRF}(k_B, x).$$

Parameters: Number of items n for the sender and receiver; length of identifiers ℓ.

On input $X = \{x_1, \dots, x_n\} \subseteq \{0,1\}^*$ from Alice and $Y = \{y_1, \dots, y_n\} \subseteq \{0,1\}^*$ from Bob, with $|X| = |Y| = n$:

1. For every $z \in X \cup Y$, choose a random identifier $R(z) \leftarrow \{0,1\}^\ell$.
2. Define $R^* = \{R(z) \mid z \in X \cup Y\}$.
3. Give output $(R^*, R(x_1), \dots, R(x_n))$ to Alice.
4. Give output $(R^*, R(y_1), \dots, R(y_n))$ to Bob.

Figure 14: Private ID functionality $\mathcal{F}_{\textsf{priv-ID}}$.

Note that after running the relevant OPRF protocols, both parties can compute $R(x)$ for their own items. To complete the private-ID protocol, they must simply perform a private set *union* on their sets $R(X)$ and $R(Y)$.

This approach indeed leads to a fine private-ID protocol.In the full-version of our paper we present and prove secure an optimization we observe that a full-fledged OPRF is not needed and a so-called "**sloppy** OPRF" would suffice.

In particular, if Bob has an item y^* that is *not* held by Alice, then it doesn't matter whether Bob learns the "correct" value $\textsf{PRF}(k_A, y^*)$. Suppose that Bob instead learns some other value z^* instead. Then Bob will consider $z^* \oplus \textsf{PRF}(k_B, y^*)$ to be the identifier of this item. Since Alice doesn't know k_B, this identifier looks random to Alice, which is the only property we need from private-ID for an item that is held by Bob and not Alice.

Hence we instantiate this general OPRF-based approach, but with a more efficient "**sloppy** OPRF" protocol. In a sloppy OPRF, Alice provides a set X; Bob provides a set Y; Alice learns k_A and Bob learns a list of output values $z_1, \dots, z_n$. For every $y_i \in Y$, if $y_i \in X$, then $z_i = \textsf{PRF}(k_A, y_i)$, but for other z_i values there is no correctness guarantee.

We achieve a sloppy OPRF using the OPPRF idea that is also used in the PSTY19 pre-processing. Namely, Bob hashes his items into bins with Cuckoo hashing. They perform a batch-OPRF, where Bob will learn $\textsf{PRF}(k_{h_i(y)}, y\|i)$ if he placed item y according to hash function h_i. Alice chooses a random seed s for a different PRF $\textsf{PRF}'$ and sends a polynomial P that satisfies $P(x\|i) = \textsf{PRF}'(s, x) \oplus \textsf{PRF}(k_{h_i(y)}, y\|i)$ for all $x \in X$ and all $i \in \{1, 2, 3\}$. Bob will compute his final output as $P(y\|i) \oplus \textsf{PRF}(k_{h_i(y)}, y\|i)$, which will equal $\textsf{PRF}'(s, y)$ in the case that Alice held the item y.

Lemma 6. *The protocol in Figure 15 securely realizes the $\mathcal{F}_{\textsf{priv-ID}}$ functionality Figure 14 in the presence of semi-honest adversaries.*

Proof. The protocol is symmetric with respect to the parties' roles, so we focus on the case of a corrupt Alice.

Claim 7. *In step 8, when Bob computes R^B, it satisfies the property that if $y \in X \cap Y$ then $R^B(y) = \textsf{PRF}'(s^A, y) \oplus \textsf{PRF}'(s^B, y)$.*

Proof. Suppose Bob placed item y into bin $h_i(y)$ according to hash function i. Then Bob computed $R^B(y)$ as $R^B(y) = P^A(y\|i) \oplus \textsf{PRF}(k^B_{h_i(y)}, y\|i) \oplus \textsf{PRF}'(s^B, y)$. Since $y \in X$ also, the polynomial P^A satisfies $P^A(y\|i) = \textsf{PRF}(k^B_{h_i(y)}, y\|i) \oplus \textsf{PRF}'(s^A, y)$. Substituting, we see that indeed $R^B(y) = \textsf{PRF}'(s^A, y) \oplus \textsf{PRF}'(s^B, x)$. This implies in particular that $R^A(y) = R^B(y)$ for $y \in X \cap Y$. □

The simulator for corrupt Alice receives ideal output $(R^*, R(x_1), \dots, R(x_n))$ and simulates Alice's view as follows:

Parameters: Size n of input sets. Cuckoo hashing parameters: hash functions h_1, h_2, h_3 and number of bins m. An auxiliary PRF PRF'.

Inputs: $X = \{x_1, \ldots, x_n\} \subseteq \{0,1\}^*$ from Alice; $Y = \{y_1, \ldots, y_n\} \subseteq \{0,1\}^*$ from Bob.

Protocol:

1. **(Sloppy PRF Bob $\rightarrow$ Alice)** Alice does $\mathcal{A} \leftarrow \mathsf{Cuckoo}^m_{h_1,h_2,h_3}(X)$.
2. The parties call $\mathcal{F}_{\mathsf{bOPRF}}$, where Alice is receiver with input $\mathcal{A}$ and Bob is sender. Bob receives output $(k^B_1, \ldots, k^B_m)$ and Alice receives output $(f^A_1, \ldots, f^A_m)$. Alice's output is such that, for each $x \in X$ assigned to bin j by hash function i, we have $f_j = \mathsf{PRF}(k^B_j, x\|i)$.
3. Bob chooses a random PRF seed s^B. He interpolates a polynomial P^A of degree $< 3n$ such that for every $y \in Y$ and $i \in \{1,2,3\}$, we have
$$P^B(y\|i) = \mathsf{PRF}'(s^B, y) \oplus \mathsf{PRF}(k^B_{h_i(y)}, y\|i)$$
He sends P^B to Alice.
4. For each item x that Alice assigned to a bin with hash function i, Alice defines
$$R^A(x) = P^B(x\|i) \oplus f^A_{h_i(x)} \oplus \mathsf{PRF}'(s^A, x)$$
5. **(Sloppy PRF Alice $\rightarrow$ Bob)** Bob does $\mathcal{B} \leftarrow \mathsf{Cuckoo}^m_{h_1,h_2,h_3}(Y)$.
6. The parties call $\mathcal{F}_{\mathsf{bOPRF}}$, where Bob is receiver with input $\mathcal{B}$ and Alice is sender. Alice receives output $(k^A_1, \ldots, k^A_m)$ and Bob receives output $(f^B_1, \ldots, f^B_m)$. Bob's output is such that, for each $y \in Y$ assigned to bin j by hash function i, we have $f^B_j = \mathsf{PRF}(k^A_j, x\|i)$.
7. Alice chooses a random PRF seed s^A. She interpolates a polynomial P^A of degree $< 3n$ such that for every $x \in X$ and $i \in \{1,2,3\}$, we have
$$P^A(x\|i) = \mathsf{PRF}'(s^A, x) \oplus \mathsf{PRF}(k^A_{h_i(x)}, x\|i)$$
She sends P^A to Bob.
8. For each item y that Bob assigned to a bin with hash function i, Bob defines
$$R^B(y) = P^A(y\|i) \oplus f^B_{h_i(y)} \oplus \mathsf{PRF}'(s^B, y)$$
9. **(Union)** The parties invoke $\mathcal{F}_{\mathsf{psu}}$, with inputs $\{R^A(x) \mid x \in X\}$ for Alice and $\{R^B(y) \mid y \in Y\}$ for Bob. They obtain output U and output the following:
$$(U, \langle R^A(x_i) \mid i \in [n]\rangle) \text{ (Alice)}$$
$$(U, \langle R^B(y_i) \mid i \in [n]\rangle) \text{ (Bob)}$$

Figure 15: Private-ID protocol.

- in step 2, uniform output f^A_j from $\mathcal{F}_{\mathsf{bOPRF}}$.
- in step 4, a polynomial P^B satisfying $P^B(x\|i) = f^A_{h_i(x)} \oplus R(x) \oplus \mathsf{PRF}'(s^A, x)$ for every item $x \in X$ placed according to hash function i, and uniform otherwise.
- in step 6, uniform keys k^A_j from $\mathcal{F}_{\mathsf{bOPRF}}$.
- in step 9, output $U = R^*$ from $\mathcal{F}_{\mathsf{psu}}$.

We show the correctness of this simulation via a sequence of hybrids:

- *Hybrid 0:* The real protocol interaction.
- *Hybrid 1:* Replace all terms of the form $\mathsf{PRF}'(s^B, y)$ with random; this change is indistinguishable from the pseudorandomness property.
- *Hybrid 2:* Replace all terms of the form $\mathsf{PRF}(k_j, x\|i)$ with random (including outputs f_j^A given to Alice); this change is indistinguishable from the security of $\mathcal{F}_{\mathsf{bOPRF}}$ and the pseudorandomness of PRF.

  Previously P^B was interpolated as $P^B(y\|i) = \mathsf{PRF}'(s^B, y) \oplus \mathsf{PRF}(k^B_{h_i(y)}, y\|i)$. Now, if Alice did not have item y and placed it according to hash function i, then the $\mathsf{PRF}(k^B_{h_i(y)}, y\|i)$ term is now uniform and independent of her view, making this output of P^B random. For $y\|i$ corresponding to Alice's item placement, the y's are distinct, and the $\mathsf{PRF}'(s^B, y)$ in those terms are now uniform, making this output of P^B random. In short, P^B is now a uniform polynomial.

  Note also that $R^B(y)$ is uniform for $y \in Y \setminus X$, because of the fresh random $\mathsf{PRF}'(s^B, y)$ term in its definition.
- *Hybrid 3:* Instead of computing $R^A(x)$ as in step 4, where one of the terms $P^B(x\|i)$ is a uniform value, we instead compute $R^A(x)$ randomly and then interpolate P^B to go through the correct value (and be otherwise uniform), *i.e.*,
$$P^B(x\|i) = R^A(x) \oplus f^A_{h_i(x)} \oplus \mathsf{PRF}'(s^A, x)$$
  This change has no effect on Alice's view distribution. Note that in this hybrid, every $R^A(x)$ is random, and every $R^B(y)$ is random subject to $R^B(y) = R^A(y)$ in the case that $y \in X \cap Y$.

This final hybrid corresponds to the final simulation, after some slight rearranging. First, a random $R(z)$ is chosen for every $z \in X \cap Y$. Then the polynomial P^B is interpolated according to $\{R(x) \mid x \in X\}$, via the expression in the simulator description. Finally, the output of $\mathcal{F}_{\mathsf{psu}}$ is $\{R(z) \mid z \in X \cap Y\}$. □

5 Comparing Communication Costs

In this section we compare our new approach to existing protocols. The focus in this section is on quantitative differences and communication complexity. In Section 6 we report on the running time of the implemented protocols.

5.1 PSU

The state of the art PSU protocol is due to Kolesnikov *et al.* [KRTW19]. In that protocol, each party's n items are hashed into $m = O(n/\log n)$ bins. The expected number of items per bin is n/m, but the worst-case load among the bins is larger by a constant factor. In order to hide the true number of items per bin, each party must add dummy items up to this worst-case maximum.

Within each bin, the parties perform a subprotocol with linear number of OPRFs, linear number of OTs, and quadratic communication. Specifically, the additional communication for β items in a bin is $\beta^2\sigma$, where $\sigma = \lambda + 2\log n$ and λ is the statistical security parameter.

Let c be the constant factor expansion within a bin to accommodate the dummy items (*i.e.*, n/m expected items in a bin, padded to cn/m including dummies). For usual set sizes, the constant is 3.2–3.6. Then the total communication cost for the protocol is:

$$cn \cdot \mathsf{bOPRF} + cn \cdot \mathsf{OT} + (c^2 n \log n)\sigma$$

Here bOPRF and OT refer to the communication costs for a single bOPRF and OT, respectively.

Our protocol requires the following: $1.27n$ OPRFs, sending one degree-$3n$ polynomial (for the PSTY19 preprocessing), roughly $1.27n\log n$ OTs (for the switching newtork), and then n additional OTs (to selectively transfer the union). Note the constant bounding the size of the Beneš network is indeed 1. The total communication cost is therefore:

$$1.27n \cdot \mathsf{bOPRF} + 3n\sigma + (1.27n \log n + n) \cdot \mathsf{OT}$$

In comparings the protocols, the dominant term is the one containing $O(n \log n)$. Our protocol is superior if $1.27\mathsf{OT} < c^2\sigma$. Indeed, the cost of an OT is $\kappa + 2\ell$ (where ℓ is the length of the item being transferred), which in our implementation is $128 + 2.60 = 248$. Hence $1.27\mathsf{OT} \approx 315$. In [KRTW19], $c^2\sigma$ is at least $10 \cdot 80 = 800$.

These pen-and-paper calculations match what we find empirically in Table 2 where our communication cost is half that of Kolesnikov *et al.* [KRTW19]. Our protocol is a significant constant factor better.

5.2 PCSI

For general-purpose PCSI, the leading protocol is due to Pinkas *et al.* [PSTY19] (PSTY19). Recall that our protocol builds on the first several steps of their protocol, which we call the PSTY19 preprocessing. We focus on the difference between the two approaches, after performing the common preprocessing. In [PSTY19], the authors report that the cost of preprocessing is roughly 4% of the total protocol cost; hence the differences we discuss in this section are reflective of the overall cost difference in the protocols.

In [PSTY19], the pre-processing is followed up with $1.27n$ private equality tests, which are performed inside generic MPC (*e.g.*, garbled circuits). To compare ℓ-bit items, the cost of such a private equality test is $2\ell\kappa$ using the state-of-the-art garbled circuit construction [ZRE15]. Hence the total communication cost is $2.54\ell\kappa n$.

In our protocol, the pre-processing is followed up by an oblivious switching network of roughly $1.27n \log n$ nodes, each requiring OT on strings of length 2ℓ. The cost of each OT is $\kappa + 4\ell$ bits, and our total communication cost is $1.27(n \log n)(\kappa + 4\ell)$.

Focusing on the asymptotically dominant term, our implementation is superior if the costs per items satisfy $1.27(\log n)(\kappa + 4\ell) < 2.54\ell\kappa$. In our implementations, $\ell = 60$ and $\kappa = 128$. Hence our cost per item is $1.27 \cdot 368 \cdot \log n = 467 \log n$ and theirs is $2.54 \cdot 60 \cdot 128 \approx 19500$. We can see that for all reasonable values of n, our cost will be significantly less than their cost (the break-even point for these particular parameters is an unrealistic $n = 2^{41}$).

5.3 Cardinality-Sum, Private ID

For cardinality-sum, private-ID, and secret-shared intersection, our approach is the first based on efficient symmetric-key operations. The prior protocols of [IKN+19, MPR+20, BKM+20] are all based on public-key techniques (Diffie-Hellman and partially homomorphic encryption). As such, their protocols will have superior communication cost but significantly higher computation costs, due to their use of public-key operations linear in the size of the input sets.

6 Performance

In this section we discuss details of our implementation and report our performance in computing the following set operations: (1) **card**: cardinality of the intersection (permuted characteristic) ; (2) **psu**: union of the sets / **psi**: intersection of the sets; (3) **priv-ID**: computing a universal identifier for every item in the union; (4) **card-sum** sum of the associated values for every item in the intersection. We compare our work with the current fastest known protocol implementation for each functionality. To the best of our knowledge, there is no known implementation to compare our **card-sum** protocol and we leave it out of our comparison. Our run times for **card-sum** is almost equal to that of **psu**.

6.1 Experimental Setup

We ran all our protocols on a single Intel Xeon processor at 2.30 GHz with 256 GB RAM. We execute the protocol on a single thread and emulate the two network connections using Linux `tc` command. For the LAN setting, we set the network latency to 0.02 ms and bandwidth of 10 Gbps and for the WAN setting the latency is set to 80 ms and bandwidth 50 Mbps. We also use a tc sub-command to compute the communication complexity for all the protocols evaluated in the performance section. We stress that we used the same methodology and environment to compute all the reported costs in this section.

6.2 Implementation details

For concrete analysis we set the computational security parameter $\kappa = 128$ and the statistical security parameter $\sigma = 40$. Our protocols are written in C++ and we use the following libraries in our implementation.

- *PSTY19 pre-processing phase.* We re-use the implementation by the authors of the paper [PSTY19]. Found: `https://github.com/encryptogroup/OPPRF-PSI.git`

- *Private equality tests.* We use the batch-OPRF construction of [KKRT16] implemented in libOTe library to compute the string equality tests. Found: `https://github.com/osu-crypto/libOTe.git`

- *Oblivious transfers and switching.* We generate many instances of oblivious transfer using the implementation of IKNP OT extension [IKNP03] from libOTe. Found: `https://github.com/osu-crypto/libOTe.git`
  Recent advances in OT extension [BCG+19b, BCG+19a] provide better asymptotic performance, but we found the existing implementations to improve over IKNP only in the multi-threaded case, while we measure only single-threaded performance. We developed our own implementation of Beneš network programming/evaluation. We used the code base in `https://github.com/elf11/benes_network_implementation` as a starting point. We emphasize that we made many corrections, implemented the functions to evaluate the network, augment it to an oblivious switching network. Further, we implemented the generalized OSN that can process any choice of input size n as opposed input sizes that are powers of 2.

- Additionally, we rely use the cryptoTools library as the general framework to compute hash functions, PRNG calls, creating channels, sending 128-bit blocks and so on. Found: `https://github.com/ladnir/cryptoTools.git`

In Table 1 we present a breakdown run time of each step in our permuted characteristic protocol. Unsurprisingly, the oblivious switching network is the most expensive step in the WAN setting, as its communication scales as $O(n \log n)$, while all other steps are linear.

	LAN (s)			**WAN (s)**		
	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
Protocol steps						
PSTY19	0.70	2.97	43.47	1.03	6.27	67.53
OSN	0.39	2.39	32.44	2.72	12.19	186.68
PEqT	0.49	1.00	8.50	3.36	6.38	28.68
Protocol core	1.58	6.36	84.41	7.11	24.84	282.89

Table 1: Run time (in seconds) of our protocol core to compute the permuted characteristic (with breakdown for each step) for input set sizes $n = \{2^{12}, 2^{16}, 2^{20}\}$ executed over a single thread for the LAN and WAN configurations.

6.3 Comparison running times

Now, we compare the run time of our protocol with the state-of-the-art for each of the functionalities. We analyse how our work compares to the previous best protocol and highlight the settings in which we beat their performance. For a fair comparison, we compiled and ran the comparison protocols and our protocol in the same hardware environment. We report the numbers for 3 input sizes $n = \{2^{12}, 2^{16}, 2^{20}\}$ all executed over a single thread. We choose our LAN setting to have latency set to 0.02 ms and a bandwidth of 10 Gbps and our WAN setting to have latency set to 80 ms and bandwidth of 50 Mbps. For our protocol, we report the average run time over 5 iterations.

Private set union. From Table 2, we can see that the empirical communication cost of our protocol is roughly half the cost of [KRTW19]. This is consistent with our back-of-the-envelope estimates from section 5. We highlight that our improvement over [KRTW19] increases with the size of the input set. This is because the run time is dominated by $O(n \log n)$ term and this becomes more significant with increased input sizes.

PSU	**LAN (s)**			**WAN (s)**			**Comm (MB)**		
	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
[KRTW19]	1.42	12.77	243.03	4.76	46.56	823.01	7.74	131.4	2476
Our protocol	1.87	8.54	114.42	9.56	28.80	319.87	3.85	67.38	1155

Table 2: Communication (in MB) and run time (in seconds) of private set union protocol for input set sizes $n = \{2^{12}, 2^{16}, 2^{20}\}$ executed over a single thread for LAN and WAN configurations.

Cardinality of intersection. From Table 3 we can observe that the communication cost of our protocol is roughly a third of the cost of [PSTY19]. This contributes to our improved run time in the WAN setting. In the LAN setting, our cardinality protocol is comparable but does not beat the numbers of [PSTY19]. This can be attributed to the time-intensive programming of the switching network in the OSN step of our protocol.

Card	**LAN (s)**			**WAN (s)**			**Comm (MB)**		
	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
[PSTY19]	1.230	5.07	65.12	7.90	38.79	530.15	10.53	166.18	2656
Our protocol	1.60	6.56	84.882	8.40	24.57	284.62	2.93	55.49	1030

Table 3: Communication (in MB) and run time (in seconds) of cardinality of intersection protocol for input set sizes $n = \{2^{12}, 2^{16}, 2^{20}\}$ executed over a single thread for LAN and WAN configurations.

Private-ID The implementation in Table 4 relies on techniques from public-key cryptography which explains their significantly lower communication costs. In comparison, our OT-based implementation that largely relies on symmetric-key operations has better performance. This is more noticeable with larger input sets, where the number of public-key operations increases linearly for [BKM+20]. It's consistent with this reasoning to see that our improvement in run times in more noticeable in the LAN setting. Unlike our Private-ID protocol, the run time of the protocol in [BKM+20] is a function of the intersection size. We sampled inputs where roughly half the elements were present in the intersection, for our experiments with both protocols. [BKM+20] implemented their protocol in Rust programming language with specific libraries that are tailored to be more efficient with elliptic curve operations speeding up their run time despite using public-key operations.

priv-ID	**LAN (s)**			**WAN (s)**			**Comm (MB)**		
	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
[BKM+20]	2.76	34.70	394.60	6.63	40.49	426.11	0.99	14.85	224.26
Our protocol	2.75	9.70	118.14	12.74	34.09	346.32	4.43	76.57	1293

Table 4: Communication (in MB) and run time (in seconds) of the private-ID protocol for input set sizes $n = \{2^{12}, 2^{16}, 2^{20}\}$ executed over a single thread for LAN and WAN configurations.

References

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 40–41. ACM Press, May 2012.

[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 Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

[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, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[Ben64] Václav E. Beneš. Optimal rearrangeable multistage connecting networks. *Bell system technical journal*, 43(4):1641–1656, 1964.

[BKM$^+$20] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Report 2020/599, 2020. https://eprint.iacr.org/2020/599.

[CGT12] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In *Cryptology and Network Security, 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings*, pages 218–231, 2012.

[CM97] Chihming Chang and Rami Melhem. Arbitrary size Benes networks. *Parallel Processing Letters*, 7(03):279–284, 1997.

[CM20] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 34–63. Springer, Heidelberg, August 2020.

[CO18] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. In Dario Catalano and Roberto De Prisco, editors, *SCN 18*, volume 11035 of *LNCS*, pages 464–482. Springer, Heidelberg, September 2018.

[CT10] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers*, pages 143–159, 2010.

[CT12] Emiliano De Cristofaro and Gene Tsudik. Experimenting with fast private set intersection. In *Trust and Trustworthy Computing - 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings*, pages 55–73, 2012.

[CZ09] Jan Camenisch and Gregory M. Zaverucha. Private intersection of certified sets. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 108–127. Springer, Heidelberg, February 2009.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 789–800. ACM Press, November 2013.

[DD15] Sumit Kumar Debnath and Ratna Dutta. Secure and efficient private set intersection cardinality using bloom filter. In *International Information Security Conference*, pages 209–226. Springer, 2015.

[EFG$^+$15] Rolf Egert, Marc Fischlin, David Gens, Sven Jacob, Matthias Senker, and Jörn Tillmanns. Privately computing set-union and set-intersection cardinality via bloom filters. In Ernest Foo and Douglas Stebila, editors, *ACISP 15*, volume 9144 of *LNCS*, pages 413–430. Springer, Heidelberg, June / July 2015.

[FHNP16] Michael J. Freedman, Carmit Hazay, Kobbi Nissim, and Benny Pinkas. Efficient set intersection with simulation-based security. *J. Cryptology*, 29(1):115–155, 2016.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 1–19, 2004.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[HFH99] Bernardo A. Huberman, Matthew K. Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *EC*, pages 78–86, 1999.

[IKN$^+$19] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. Cryptology ePrint Archive, Report 2019/723, 2019. https://eprint.iacr.org/2019/723.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 54–70. Springer, Heidelberg, August 2013.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.

[Knu08] Donald E. Knuth. The art of computer programming, vol. 4, pre-fascicle 1a, 2008.

[KRTW19] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 636–666. Springer, Heidelberg, December 2019.

[KS05] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 241–257. Springer, Heidelberg, August 2005.

[Mea86] Catherine A. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 7-9, 1986*, pages 134–137, 1986.

[MPR$^+$20] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-sided malicious security for private intersection-sum with cardinality. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2020.

[MRR19] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins for secret shared data. Cryptology ePrint Archive, Report 2019/518, 2019. https://eprint.iacr.org/2019/518.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 557–574. Springer, Heidelberg, May 2013.

[PRTY19] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. SpOT-light: Lightweight private set intersection from sparse OT extension. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 401–431. Springer, Heidelberg, August 2019.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 739–767. Springer, Heidelberg, May 2020.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium, USENIX Security 15*, pages 515–530, 2015.

[PSTY19] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 122–153. Springer, Heidelberg, May 2019.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 125–157. Springer, Heidelberg, April / May 2018.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *23rd USENIX Security Symposium, USENIX Security 14*, pages 797–812, 2014.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. http://eprint.iacr.org/2005/187.

[RR17a] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 235–259. Springer, Heidelberg, April / May 2017.

[RR17b] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1229–1242. ACM Press, October / November 2017.

[VC05] Jaideep Vaidya and Chris Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, 13(4):593–622, 2005.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

A Switching Networks

A.1 Definitions

A **switching network** is a circuit (dag) with the following kinds of gates. Each gate has *primary* inputs/outputs as well as a *programming* input.

- A **multiplexer** takes k primary inputs and selects one of them to transfer to its single primary output. The choice of input is determined by the programming input (an element of $[k]$).
- A **permute switch** maps its two primary inputs to its primary outputs using a permutation selected by the programming input (a single bit).[2]

If $\mathbb{S}$ is a switching network, then we write $\mathbb{S}^p(a_1, \ldots, a_{n_{\mathsf{in}}}) = (b_1, \ldots, b_{n_{\mathsf{out}}})$ to denote that on programming inputs p, and primary inputs $a_1, \ldots, a_{n_{\mathsf{in}}}$, the network outputs $b_1, \ldots, b_{n_{\mathsf{out}}}$.

A.2 Universal Switching Networks

A switching network $\mathbb{S}$ is **universal** for a class of functions $\pi : [n_{\mathsf{out}}] \to [n_{\mathsf{in}}]$ if for every such function π in the class, there is a way to set the programming inputs of $\mathbb{S}$ so that it realizes the mapping $(a_1, \ldots, a_{n_{\mathsf{in}}}) \mapsto (a_{\pi(1)}, \ldots, a_{\pi(n_{\mathsf{out}})})$.

When discussing a universal switching network $\mathbb{S}$, we often abuse notation and identify the function $\pi : [n_{\mathsf{out}}] \to [n_{\mathsf{in}}]$ with the *programming string* for $\mathbb{S}$ that realizes it. Hence, we write $\mathbb{S}^\pi(a_1, \ldots, a_{n_{\mathsf{in}}}) = (a_{\pi(1)}, \ldots, a_{\pi(n_{\mathsf{out}})})$. We sometimes denote the output vector by $\vec{a}^{(\pi)}$.

Permutations. There is a well-known recursive construction of universal switching networks, for the class of all **permutations on** $[n]$, due to Beneš [Ben64] that uses at most $n \log_2 n$ permute switches when n is a power of two. A generalizion to arbitrary n is found in Chang & Melhem [CM97]. We review this construction in Appendix B.1.

Injective Functions. A permutation network selects an ordering of its inputs. Consider a switching network with n_{in} inputs and $n_{\mathsf{out}} \le n_{\mathsf{in}}$ outputs, that selects an ordered **subset** of its inputs. In our terminology, such a switching network corresponds to an injective function $\pi : [n_{\mathsf{out}}] \to [n_{\mathsf{in}}]$.

In Appendix B.2 we give a new generalization of the Beneš construction that is universal for this class of functions, and uses at most $n_{\mathsf{in}} \log_2 n_{\mathsf{out}}$ permute switches and n_{out} multiplexers.

A.3 Oblivious Switching Network

An **oblivious switching network (OSN)** protocol is a protocol that takes as input π from a receiver, and an input $\vec{x}$ from a sender, and gives as output additive secret shares of $\mathbb{S}^\pi(\vec{x})$, where $\mathbb{S}$ is a public switching network. More formally, the ideal functionality for an oblivious switching network is given in Figure 2.

Mohassel & Sadeghian [MS13] describe an efficient OSN protocol based on oblivious transfer extension. The main idea is that the sender chooses a random mask M for every wire of the network; then the invariant is that for each wire, the receiver will learn $M+v$ where v is the logical value on that wire of the network. The sender's masks on the output wires, along with the receiver's masked values on the output wires, correspond to an additive sharing of the switching network's logical output. For the input wires, the sender simply sends its input blinded by the appropriate input-wire masks.

It suffices to show how to obliviously evaluate each gate. Consider a multiplexer whose input wires have masks $A_1, \ldots, A_k$ and whose output wire has mask B. The parties can perform a 1-out-of-k oblivious transfer where the sender's inputs are $(A_1 \oplus B, \ldots, A_k \oplus B)$, the receiver's input is its programming input $p \in [k]$, and the receiver learns $A_p \oplus B$. Inductively, the receiver already knows $A_p \oplus v$ for the value v on wire p. He can therefore compute $(A_p \oplus B) \oplus (A_p \oplus v) = (B \oplus v)$ which is the correct masked value for the output of this multiplexer gate.

For permutation switches, observe that these can be written as two 1-out-of-2 multiplexers. However, since the two multiplexers are always programmed in opposite fashion, the OSN protocol can use just a single oblivious transfer for these switches. In particular, consider a permutation switch with input wire masks A_1, A_2 and output wire masks B_1, B_2. Parties can perform a 1-out-of-2 oblivious transfer where the sender's input is $((A_1 \oplus B_1)||(A_2 \oplus B_2), (A_1 \oplus B_2)||(A_2 \oplus B_2))$, the receiver's input is the programming bit $p \in \{0, 1\}$ and receiver learns $(A_1 \oplus B_{1+p})||(A_2 \oplus B_{2-p})$. Receiver already knows $A_1 \oplus v_1$ and $A_2 \oplus v_2$, and

[2]Formally speaking, a permute switch can be realized from two multiplexers, but conceptually it is convenient to separate this important case.

Parameters: A switching network $\mathbb{S}$ with n_{in} inputs, n_{out} outputs.
Sender's input: A vector $\vec{x} \in (\{0,1\}^\ell)^{n_{\mathsf{in}}}$.
Receiver's input: A programming sequence π for $\mathbb{S}$.

1. For every wire i in $\mathbb{S}$, the sender chooses random mask $M_i \leftarrow \{0,1\}^\ell$
2. For every gate g in $\mathbb{S}$, the parties perform an oblivious transfer, where the receiver's input is the programming input π_g for that gate.
   (a) If g is a k-multiplexer with input wires $i_1, \ldots, i_k$ and output wire j, then the OT is a 1-out-of-k OT and the sender's input is $(M_{i_1} \oplus M_j, \ldots, M_{i_k} \oplus M_j)$.
   (b) If g is a permute switch with input wires i_1, i_2 and output wires j_1, j_2, then the OT is a 1-out-of-2 OT and the sender's input is $\big((M_{i_1} \oplus M_{j_1}) \| (M_{i_2} \oplus M_{j_2}), (M_{i_2} \oplus M_{j_1}) \| (M_{i_1} \oplus M_{j_2})\big)$.
3. For every input wire i, the sender sends $x_i \oplus M_i$
4. For every output wire w, the receiver identifies the path from output wire to the corresponding input wire $\pi(w)$. Say that path contains wires $i_1, \ldots, i_k$. Then from step 2a the receiver has $M_{i_j} \oplus M_{i_{j+1}}$ for every j, and from step 3 the receiver has $M_{i_1} \oplus x_{\pi(w)}$. XORing all of these together, the receiver obtains $x_{\pi(w)} \oplus M_w$ where M_w is the mask on the output wire w.
5. The sender outputs the collection of output wire masks. The receiver outputs the collection of $x_{\pi(w)} \oplus M_w$ values computed in the previous step.

Figure 16: Oblivious Switching Network protocol of [MS13].

therefore can compute $v_1 \oplus B_{1+p}$ and $v_2 \oplus B_{2-p}$ which are the correct masked values given the permute bit p.

A.3.1 Details, Costs, Optimizations

The protocol is described in more detail in Figure 16. As mentioned, it requires an oblivious transfer for each permutation switch or multiplexer in the switching network. Note that all of the OTs can be done in parallel. The sender's OT inputs are derived from random masks that are chosen upfront for each wire in the network. The receiver's OT inputs are just the programming string of the switching network.

Let ℓ be the length of the items on the wires of the switching network. For a multiplexer, the parties perform a 1-out-of-k OT where each payload is of length ℓ. For a permute-switch, we perform a 1-out-of-2 OT where each payload is of length 2ℓ. Using modern OT extension protocols, the marginal communication cost for an OT is simply the sum of its payload lengths.

Random OT optimizations. In these OT extension protocols, no online communication is needed for a payload that is chosen randomly [ALSZ13]. This is because the OT extension paradigm natively gives OT of random payloads (i.e., the "protocol itself" chooses random OT payload). When the sender lets the OT extension mechanism choose a random payloads, there is no need to send a further "correction" value. We can apply this optimization to the OSN protocol, since our OT payloads are XORs of randomly chosen wire masks. Instead of first choosing random wire masks for this gate's output wire(s) and using those to determine the OT payloads, we let the OT extension dictate the first OT payload randomly, and use that to solve for the appropriate wire mask. Applying this optimization saves only ℓ out of $k\ell$ bits for a multiplexer, but reduces the communication cost of a permute switch by half.

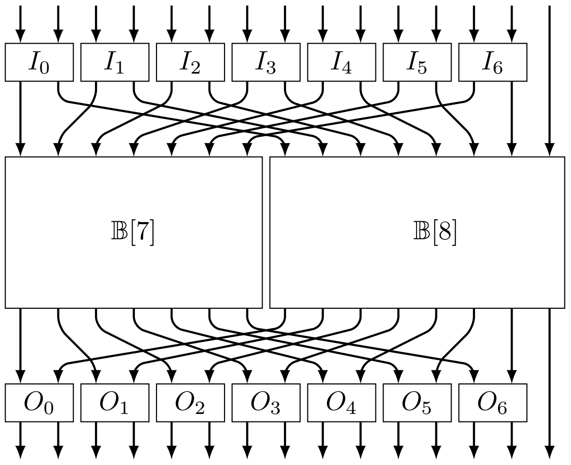

Figure 17: Beneš network $\mathbb{B}[15]$.

B Beneš Networks

B.1 Traditional Beneš Network

In this section we describe Beneš networks for permutations. The original construction of Beneš was restricted to powers of two [Ben64]. We present a generalization due to Chang & Melhem [CM97]. We follow the analysis from Knuth [Knu08]. In this section we use notation $[N]$ to mean $\{0, \dots, N-1\}$.

Definitions. Let $\mathbb{B}[N]$ denote a Beneš network with N inputs and N outputs. $\mathbb{B}[N]$ is defined as follows:

- $\mathbb{B}[1]$ is a circuit with a single wire and no switches.
- $\mathbb{B}[2]$ is a circuit with a single permutation switch.

For $n \geq 3$, let $R = \lceil N/2 \rceil$ and $L = \lfloor N/2 \rfloor$. Then $\mathbb{B}[N]$ is constructed recursively as follows. The network consists of:

- "Input-layer" permutation switches $I_0, \dots, I_{L-1}$
- Recursive instances: a left instance $\mathbb{B}[L]$ and an right instance $\mathbb{B}[R]$.
- "Output-layer" permutation switches $O_0, \dots, O_{L-1}$.

Let us identify the inputs/outputs of $\mathbb{B}[N]$ as elements of $[N]$. Then the jth input to $\mathbb{B}[N]$ is connected to $I_{\lfloor j/2 \rfloor}$. The jth output of $\mathbb{B}[N]$ is connected to $O_{\lfloor j/2 \rfloor}$. Each switch I_k sends one of its outputs to the kth input of $\mathbb{B}[L]$ and the other to the kth input of $\mathbb{B}[R]$. Likewise, each switch O_k receives one input from the kth output of $\mathbb{B}[L]$ and the other from the kth output of $\mathbb{B}[R]$. When N is odd, there is one input/output leftover, and these are connected directly (*i.e.*, without input/output-layer switches) to $\mathbb{B}[R]$ which has an unassigned input/output.

The number of switches in the Beneš network is defined by the recurrence:

$$\begin{aligned} S(1) &= 0 \\ S(2) &= 1 \\ S(N) &= 2\lfloor N/2 \rfloor + S(\lfloor N/2 \rfloor) + S(\lceil N/2 \rceil) \end{aligned}$$

It is possible to show that $S(N) \leq N \log_2 N - \frac{N}{2}$.

Lemma 8. *For any permutation $\pi : [N] \to [N]$, the switches in $\mathbb{B}[N]$ can be programmed to realize the mapping $(x_0, \dots, x_{N-1}) \mapsto (x_{\pi(0)}, \dots, x_{\pi(N-1)})$.*

Proof. Define an undirected graph with vertex set:

$$\begin{aligned} V = \{(\mathsf{in}, \{2i, 2i+1\}) \mid i \in [L]\} \\ \cup \{(\mathsf{out}, \{\pi(2i), \pi(2i+1)\}) \mid i \in [L]\} \end{aligned}$$

If N is odd, then we also include vertices $(\mathsf{in}, \{N-1\})$ and $(\mathsf{out}, \{N-1\})$. Note that for every $i \in [N]$ there is a unique vertex (in, S) with $i \in S$ and a unique vertex (out, S) with $i \in S$.

The labeled edges of the graph are defined as follows. For every i, connect the unique vertices (in, S), (out, S') having $i \in S \cap S'$ with an edge labeled by i. Note that a vertex $(\cdot, S)$ has degree exactly $|S|$.

By construction, the graph is bipartite, so every cycle has even length. This implies that the edges of the graph can be colored with two colors $\{\mathsf{left}, \mathsf{right}\}$ so that incident edges have opposite colors. Note that each connected component has two equally valid edge-colorings (obtained by swapping colors). We swap the colors in a component so that the edges incident to degree-1 vertices have color right. Note that vertices of degree 1 happen only when N is odd, and in that case there are two such vertices, in opposite sides of the bipartition. It follows that any path between these two vertices has odd length, so the first/last edges of this path will have the same color.

We claim that this edge-coloring corresponds to a programming of the switches. Namely, the edge with label i has color left if and only if item i is routed through the left recursive switching network. To see why this is a valid, observe:

- Every degree-2 vertex $(\mathsf{in}, \{a, b\})$ corresponds to an input-layer switch that is receiving input items a and b. The two edges incident to this vertex are labeled a and b, and are colored opposite colors from $\{\mathsf{left}, \mathsf{right}\}$. This corresponds to the two possible settings of the switch (*i.e.*, sending a left and b right, or vice-versa).

  The possible degree-1 vertex $(\mathsf{in}, \{a\})$ corresponds to the input wire for item $N-1$. By construction, the color of its incident edge is right. This corresponds to item $N-1$ being sent directly into the right recursive instance, which indeed agrees with the topology of the network.

- Similarly, every degree-2 vertex $(\mathsf{out}, \{a, b\})$ corresponds to an output-layer switch that wants to receive items a and b. Its two edges are colored opposite colors, corresponding to a valid setting for the switch (*i.e.*, receiving a from the left and b from the right, or vice-versa).

  The possible degree-1 vertex $(\mathsf{out}, \{a\})$ corresponds to the last output wire of the circuit, which wants to receive item $\pi(N-1)$. As above, its incident edge is colored right, indicating that $\pi(N-1)$ will arrive from the right recursive instance.

- For every item i, there is a single edge with that label. Hence both the endpoints (in, S) and (out, S') with $i \in S \cap S'$ agree on whether item i is being routed through the left or right recursive instance. Hence, each recursive instance has the same set of inputs as outputs. In other words, each recursive instance is indeed being asked to realize a permutation. Hence, the programming of switches can be completed recursively.

□

B.2 Generalized Beneš Construction

In this section we generalize the above construction to any mapping $(x_0, \ldots, x_{N-1}) \mapsto (x_{\pi(0)}, \ldots, x_{\pi(T-1)})$ where $T \leq N$ and $\pi : [T] \to [N]$ is any injective function.

Let $\mathbb{B}[N, T]$ denote such a network, which we construct recursively:

- $\mathbb{B}[N, 1]$ is the base case, which we interpret as a 1-out-of-N multiplexer.
- $\mathbb{B}[N, T]$ for $T \geq 2$ consists of $\lfloor N/2 \rfloor$ input switches, $\lfloor T/2 \rfloor$ output switches, and recursive copies of $\mathbb{B}[\lfloor N/2 \rfloor, \lfloor T/2 \rfloor]$ and $\mathbb{B}[\lceil N/2 \rceil, \lceil T/2 \rceil]$.

The connectivity of the switches is analogous to the standard Beneš network. That is, each switch has one connection to/from each of the two recursive copies. When N or T is odd, the "leftover" input/output wire is connected directly to the (larger) right recursive instance.

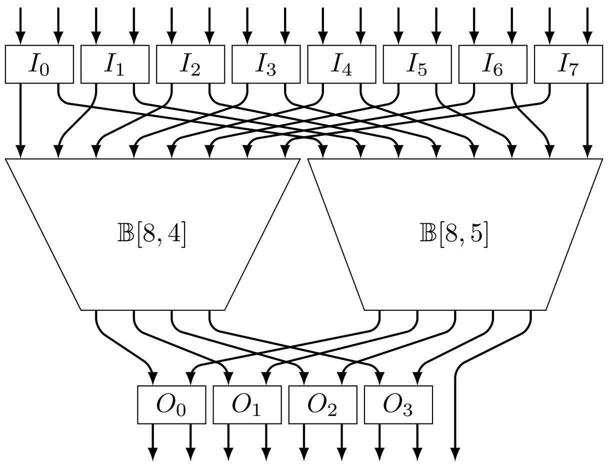

Figure 18: Generalized Beneš network $\mathbb{B}[16, 9]$.

Lemma 9. *$\mathbb{B}[N, T]$ can be programmed to realize any mapping $(x_0, \dots, x_{N-1}) \mapsto (x_{\pi(0)}, \dots, x_{\pi(T-1)})$ where $\pi : [T] \to [N]$ is an injective function.*

Proof. We have written the proof of the standard Beneš network in such a way that it applies almost verbatim to this case. As before, we define a graph with vertices corresponding to every input/output switch as well as any input/output wires attached directly to the recursive subnetworks.

Unlike before, there may not be an edge corresponding to every item. If item i is not needed as an output, there is no edge labeled i. Hence some vertices will be deficient in their degree (*i.e.*, a vertex $(\cdot, S)$ may have fewer than $|S|$ incident edges), and we may even have isolated vertices. Still, the graph is bipartite and can be edge-colored so that no vertex has two incident edges of the same color. As before, we arrange so that any edge incident to a "singleton" vertex $(\cdot, \{a\})$ is colored right.

Any vertex that is not isolated corresponds either to a input/output-layer switch or a directly-connected input/output wire. The color of the incident vertices uniquely determines the programming of those switches. By the same arguments as above, the switches are self-consistent — every item that is requested as an output of one of the recursive components is indeed being given as an input to that same component. This implies that the recursive components are each being asked to perform a valid task, and they can be programmed recursively.

Note that some input-layer switches may not have been determined from this reasoning. In particular, we may have input-layer switches whose corresponding vertices are isolated in the graph. These can be set arbitrarily; they cannot affect the above consistency property. ☐

Note that this Beneš network consist of permutation switches as well as multiplexers. The number of such components required is:

$$\begin{aligned}
S_{\mathsf{per}}(N,1) &= 0 \\
S_{\mathsf{per}}(N,T) &= \lfloor N/2 \rfloor + \lfloor T/2 \rfloor + S_{\mathsf{per}}(\lfloor N/2 \rfloor, \lfloor T/2 \rfloor) \\
&\qquad + S_{\mathsf{per}}(\lceil N/2 \rceil, \lceil T/2 \rceil) \\
S_{\mathsf{mul}}(1,1) &= 0 \\
S_{\mathsf{mul}}(N,1) &= 1 \\
S_{\mathsf{mul}}(N,T) &= S_{\mathsf{mul}}(\lfloor N/2 \rfloor, \lfloor T/2 \rfloor) + S_{\mathsf{mul}}(\lceil N/2 \rceil, \lceil T/2 \rceil)
\end{aligned}$$

In particular, when $N = 2^n$ and $T = 2^t$ are powers of two, we have:

$$\begin{aligned}
S_{\mathsf{per}}(N,T) &= t\frac{N+T}{2} \\
S_{\mathsf{mul}}(N,T) &= T
\end{aligned}$$

Furthermore, the multiplexer required is a 1-out-of-2^{n-t} multiplexer. In general, $S_{\mathsf{per}}(N,T) \leq N \log_2 T$, and $S_{\mathsf{mul}}(N,T) = T$.

Asymptotically, the construction requires $\Theta(N \log T)$ permutation switches and T of 1-out-of-$\Theta(N/T)$ multiplexers.