Two-party Private Set Intersection with an Untrusted Third Party

Phi Hung Le*[1], Samuel Ranellucci†[2], and S. Dov Gordon‡[1]

[1]Department of Computer Science, George Mason University
[2]Unbound Tech, Petach Tikva, Israel

November 20, 2019

Abstract

We construct new protocols for two parties to securely compute on the items in their intersection. Our protocols make use of an untrusted third party that has no input. The use of this party allows us to construct highly efficient protocols that are secure against a single malicious corruption.

1 Introduction

Secure multi-party computation protocols enable multiple distrusting data holders, to jointly compute on their collected input, while revealing nothing to any party other than the output. Many of the foundational questions about secure computation were resolved in the last century; in the last decade, a long line of research has focused on concrete complexity, in an attempt to close the gap between the cost of computing on private data, and that of computing in the clear. Today, a handful of companies across the world have begun selling secure computation for a variety of applications[1], and Google claims to be using it in-house to perform set intersection, helping advertisers to determine the efficacy of their ads [24].

Protocols for secure computation can be categorized according to various modeling parameters. Generally speaking, protocols that are secure when a majority of parties are honest have much lower bandwidth requirements than those that tolerate a malicious majority. This makes the three-party setting for secure computation especially appealing in real deployments: users can outsource their private computation by secret sharing their input among three servers, with a guarantee of security so long as at most one party becomes malicious. Consequently, there has been a lot of recent advancement in making three-party [6, 27, 14, 2, 16, 26] and four-party [21] secure computation highly efficient in the face of a single corruption. In the three-party setting, Furukawa et al. [16] require just 10 bits of communication per party for each Boolean gate, Araki et al. [1] require just 7, and in the four-party setting, Gordon et al. [21] require just 1.5 bits of data. As a point of contrast, using garbled circuits requires sending $O(\kappa)$ bits per gate in the two-party setting.

Most of the literature exploring the advantages of computing in the three-party or four-party model have focused on generic computation. (One important exception is the work of Kamara et al. [27], which we discuss below.) However, certain problems are important enough to warrant optimized protocols that out-perform the generic solutions. Perhaps no single application has received as much attention as private set intersection (PSI). In part, undoubtedly, this is because PSI is such an important application. But it is also a natural, simply stated problem that makes for the perfect academic challenge: PSI protocols have been constructed through polynomial interpolation [15], oblivious PRFs [29], public key encryption [9], Bloom filters [12, 37], and cuckoo hashing [35], to name just some of the results on PSI. At one point, it was demonstrated that garbled circuits out-performed many of these constructions in certain domains [23], but the research continued, and the fastest PSI protocols are again custom constructions [29, 38].

While this focus on PSI is understandable, arguably it goes too far in its *lack* of generality. While intersection is undoubtedly an important computation, one can easily imagine many scenarios where it is only the first step in a broader computation. (Pinkas et al. [35] provide a nice list of such examples.) A more general application, which

*ple13@gmu.edu
†samuel.ranellucci@unboundtech.com
‡gordon@gmu.edu
[1]Unbound Tech, Sharemind, and Partisia, to name a few.

we call $f(PSI)$, has received somewhat less attention. In this application, the users wish to compute some arbitrary function f over the intersection, revealing $f(PSI)$, but not the items in the intersection themselves. Motivated by the discussion described above, we look at this problem while enlisting the help of an untrusted third party: we assume that two parties each have large input sets, and that a third party is available to help them compute. We allow for the malicious corruption of at most one party. Taking two further relaxations, in nearly all of our protocols we allow the third party to learn the size of the intersection, and, although we assume an honest majority among the three parties, we do not guarantee *fairness*: one of the parties might receive output and then choose to abort the protocol before the others learn anything.

1.1 Contributions

We develop several new protocols related to private set intersection and private set union, in the three-party model. Our protocols are of two varieties: one group of protocols relies on polynomial interpolation, and the other relies on a mix of generic techniques, both existing and new, for 3-party computation. Protocols using interpolation require less communication, while those from circuits require less local computation; we also construct a hybrid protocol that leverages the advantages, and disadvantages, of both approaches. We explore these tradeoffs experimentally.

Intersection cardinality. We design three new protocols for privately computing the cardinality of the set intersection. (Sections 3 and 4.1.) Note that this is a harder problem than PSI, since it leaks strictly less information. These protocols serve as a building block for some of our constructions of $f(PSI)$, though, viewing intersection cardinality as a particular f in the $f(PSI)$ problem, we see these results as independently interesting.

In the LAN setting, our circuit-based protocol improves on the runtime of the 2-party protocol of De Cristefaro et al. [8] by 190-1018X, but may need 3X more bandwidth. Our polynomial-based protocol improves on their communication cost by more than 6X, while improving the runtime by 20-26X. Our hybrid protocol improves the runtime by 28-54X and requires 2X less bandwidth. In WAN setting, the above protocols are faster than [8] 33-84X, 20-25X, and 25-40X respectively. Compared with the generic merge-compare-add protocol implemented with [1] in three-party setting, if the indices have length $\sigma = 80$ bits, our circuit-based PSI-CA is about 5X-25X faster in LAN, 5X-14X faster in WAN, and use 5X-35X less bandwidth.

Computing functions of the intersection payload. We identify an interesting restriction of $f(PSI)$ in which the function f depends only on the payload data, and not on the indices themselves. This is a natural restriction and would arise, for example, when the indices are unique record identifiers, such as social security numbers or public keys. Interestingly, this restriction does not seem to help in the 2-party setting.

Intuitively, to compute f on the z items in the intersection, the parties need to learn which outputs from the intersection computation should be "stitched" into the evaluation of f. Unless the outputs are obliviously shuffled, which requires $O(n \log n)$ secure swaps, the stitching leaks which elements are in the intersection.[2]

In the 3-party setting, the third party (P_3) can help permute the input arrays such that the intersection is at the front of the array, and is easily stitched into the circuit computing f. One of our technical contributions is a concretely efficient protocol, requiring $O(n)$ secure operations (and local sorting) for obliviously permuting an array according to one party's specified permutation (Section 4.3).

Computing functions of the intersection indices. Extending the protocols that compute a function over the intersection payload, we also design two protocols for computing on the indices (with or without payload). Unfortunately, these constructions are quite inefficient, and further work is required here. The more efficient protocol requires non-black box access to a PRF. We instantiate this with AES, and implement it. Despite the size of the AES circuit, we still estimate that this will outperform existing constructions for certain functions f, and for payloads greater than ≈ 160 bits.

Our second protocol makes black-box use of a two-party shared oblivious prf (2soprf). This is a PRF in which the input can be secret shared by two parties. We instantiated this using the 2soprf proposed by Gordon et al. [20], but found that it performed worse than the first protocol. However, we are hopeful that this instantiation can be replaced by something more efficient, perhaps built from OT extension. We leave this to future work.

1.2 Related work

PSI with an untrusted mediator. Kamara et al. [27] proposed the same security model and constructed very efficient PSI protocols. Our protocols also naturally cover the case of PSI, but we do not claim this as a contribution, mainly because of their work. Their protocols are faster than our own, when we restrict to computing

[2] Orlandi et al. [7] and Pinkas et al. [35] propose a way around this problem. We briefly discuss their approaches in Section 1.2.

PSI. Unfortunately, there is no clear way to extend their results to computing cardinality of the PSI or f(PSI). In their protocols, the input parties use dummy elements, and duplicated inputs, to catch a cheating server that tries to remove any records. To verify that the server is honest, they reveal the intersection and verify that all intersecting dummy elements have been included in the output. This is perfectly fine when computing PSI but it reveals too much when we want to compute PSI cardinality or f(PSI). We provide several new, completely different approaches for enforcing honest behavior by the server.

PSI with computation. Recently, Orlandi et al. [7] and Pinkas et al. [34, 35] provide 2-party protocols for computing arbitrary functions over the intersection. They both claim $O(n)$ complexity in the semi-honest setting. To avoid the issue we previously described, which required the output of the intersection computation to be obliviously shuffled, in these works they instead feed all $2n$ values into the circuit for f, together with indicator bits that denote whether an item was in the intersection. Depending on f, this might be a very reasonable solution: for example, if f is a simple summation, then a linear-sized circuit can easily include exactly the right items, using 1 multiplication for each indicator bit. However, for some functions this will result in the $\log n$ overhead that we manage to avoid. For example, if f computes the median of the intersecting items, the best oblivious construction we know of requires $O(n \log n)$ gates.[3] In our construction, since we only feed $t = |X \cap Y|$ into the circuit for f, we would need only $t \log t$ gates to complete the computation.

The protocol of Pinkas et al. is secure in the 2-party, semi-honest setting, while ours is secure in the 3-party malicious setting, assuming an honest majority, so we are forced to compare apples to oranges.[4] In their work, they provided experiment results for PSI cardinality threshold (PSI-CAT), which has $O(n)$ complexity. We compare our PSI cardinality protocols with this (with the assumption that their PSI-CA runtime would be similar to that of PSI-CAT). Our circuit-based protocol is 4.4X-24X faster in LAN, and 3.5X-9X faster in WAN while using 3.5X-25X less bandwidth for $n = 2^{20}$, depending on the size of the intersection. We note that we can choose the most efficient protocol for PSI-CA "on the fly" after learning the intersection size. Also, the performance of their protocol depends on the input length. The gap will be about 2.5X wider if the input has variable length. The detailed comparison is shown in Table 1.

On the other hand, when computing functions on the intersecting indices, the comparison depends on the functionality and the size of the payload. For f that requires only $O(n)$ gates, their protocol is likely much faster than ours, due to our non-blackbox use of AES. However, if f requires $O(n \log n)$ gates for them, and only $O(t \log t)$ gates for us, then on payloads of about 160 bits, we will become competitive.

Other related work. There are multiple works looking at intersection cardinality [15, 28, 8, 11]. These protocols are secure against one malicious party in the two-party setting, however, they are very inefficient. In [8, 11], it takes more than an hour for two parties to find the intersection cardinality for sets of size 2^{20}. Ion et al. [25] compute the sum of all items in the intersection. Another application that relates to $f(PSI)$ is labeled-PSI [4], in which a sender sends a label l_i to a receiver if the item x_i is in the intersection. Labeled-PSI is an efficient two-party protocol to perform computation over the intersection in the asymmetric setting (a server has many items and a client has few ones). However, labeled-PSI leaks the whole intersection, which we aim not to do. It also would compare very unfavorably in the symmetric setting, where input sets are roughly the same size. Finally, their security model is incomparable, as they are secure against a malicious receiver, but can only ensure privacy against a malicious sender, while we assume an honest majority. Finally, a crucial idea underlying all of our protocols is that, by revealing a deterministic encryption of the input to a third party, who can then compute the intersection from the encrypted values. A similar approach was used in other settings for PSI [27, 22].

[3] We note that median is an example of a symmetric function. As Pinkas et al. [35] point out, When avoiding the oblivious shuffle of the intersection, it is necessary that f be a symmetric function, or the output of f may leak something about the intersecting items. While this property is necessary for claiming security, the case of median demonstrates that it might not be sufficient for claiming efficiency.

[4] If we choose to model each player becoming maliciously corrupt with independent probability p, then the probability of 0 or 1 corruptions among 3 parties is strictly smaller than the probability of 0 corruptions among 2 parties, which is needed in the 2-party, semi-honest setting. (Independence of the corruption events is not entirely necessary for this argument, but some restriction on the corruption events is necessary.) Although this is never explicitly modeled in the literature, it does provide a nice justification, *from a security stand point*, for preferring the 3-party, malicious, honest-majority assumption, over the 2-party, semi-honest assumption. Of course, this model of corruption is not always reasonable, but it does seem to be implicitly used in many real-world deployments, where the computing parties are operated by a single agency.

2 Definitions and Notation

2.1 Security Definitions

We prove our protocols secure in the real/ideal paradigm, in the stand-alone setting, and achieving security with abort. Formal definitions can be found in Goldreich [18]. Furthermore, in all of our protocols, we allow P_3 to learn the size of the intersection of the input sets; this is reflected in our ideal functionalities. Of course, it is possible to achieve fairness when only one out of three parties is malicious, and revealing the intersection to P_3 could be avoided as well. However, these relaxations are important for achieving the level efficiency that we demonstrate.

2.2 Secret Sharing

We use several different secret sharing schemes in our protocols. In our constructions based on polynomial interpolation (Section 3), we rely on Shamir secret sharing. In our circuit-based protocols of Section 4.1, we use additive, 2-out-of-2 sharings of field elements, and replicated secret sharing. The latter is a 2-out-of-3 secret sharing scheme where, to share a field element x, 3 field elements are selected at random, subject to $x_1 + x_2 + x_3 = x$. Then, P_1 is given x_1 and x_2, P_2 is given x_2 and x_3, and P_3 is given x_3 and x_1. We will move back and forth between these two sharing schemes, and we will apply them both to binary values, as well as to larger fields.

Notation: we denote $[x]^A, [x]^B$ as replicated arithmetic and binary shares of x respectively, and $\langle x \rangle^A, \langle x \rangle^B$ 2-out-of-2 additive sharings. We sometimes write $[x]^A = (x_1, x_2, x_3)$, ignoring the replication of shares.

2.3 Assumed functionalities

Our protocols are described in a hybrid world, where we assume access to several simple, trusted functionalities. All of these have been implemented securely in our experiments. For completeness, we include detailed descriptions of these functionalities in Appendix A. The secure protocols for these functionalities can be found in the work of Chida et al. [5]. We summarize the functionalities here:

- $\mathcal{F}_{\text{rand}}$ gives a replicated arithmetic sharing of a random element $r \in Z_p$ (Figure 13).
- $\mathcal{F}_{\text{coin}}$ gives all parties the same random element $r \in Z_p$ (Figure 14).
- $\mathcal{F}_{\text{input}}$ secret-shares data owned by one of the parties as replicated arithmetic shares (Figure 17).
- $\mathcal{F}_{\text{mult}}$ takes replicated arithmetic shares of two input values, and outputs shares of their product, up to an addictive attack. That is, the functionality allows the adversary to specify a constant that will be added to the product (Figure 15).
- $\mathcal{F}_{\text{CheckZero}}$ gives **true** to the parties if they hold a replicated arithmetic share of zero, otherwise, it gives **false** (Figure 16).

2.4 Authentication on additive shares

We define $MAC_\alpha(x) \equiv \alpha x$ as the MAC of x, where $\alpha, x \in Z_p$, x is the data, and α is the MAC key. Technically, this is not a secure authentication code, since anybody can recover α after seeing a single authentication. As is standard in MPC work, however, the MAC key and the authentications will always be secret-shared. The key is sampled by calling $\mathcal{F}_{\text{rand}}$. The shared MAC is computed by calling $\mathcal{F}_{\text{mult}}$ on the shared key and data, $([\alpha]^A, [x]^A)$. Note that we allow the MAC to be computed up to an additive attack. The adversary can add an arbitrary additive term d to the MAC. Thus, the parties will hold shares of $\alpha x + d$ in stead of αx. To simplify the presentation, we use the same notation for the MAC with additive attack: $MAC_\alpha(x) \equiv \alpha x + d$.

2.5 Share Conversion

$[x]^A \to [x]^B$: There are scenarios that the parties are holding replicated arithmetic secret shares and they want to compare the shares. It is more efficient for them to convert the shares to replicated binary ones and perform the comparison with a Boolean circuit. We use an approach similar to that of Mohassel and Rindal in their ABY3 system to convert $[x]^A \to [x]^B$ [31], though we extend their technique so that it can be used with arbitrary fields; they only required share conversion for rings. Let $[x]^A = (x_1, x_2, x_3)$, P_1, P_2, and P_3 hold $(x_1, x_2), (x_2, x_3), (x_3, x_1)$ respectively. From $[x]^A$, parties can set $[x_1]^B = (x_1, 0, 0)$, $[x_2]^B = (0, x_2, 0)$, and $[x_3]^B = (0, 0, x_3)$ without interaction. Let k be

the bit length of x_i. The three parties first call k full adders to compute $(c[i], s[i]) \leftarrow FA(x_1[i], x_2[i], x_3[i])$. After this step, they hold $[c]^B$ and $[s]^B$ and execute a ripple carry adder circuit to compute $[x]^B \leftarrow 2[c]^B + [s]^B$. This is correct as $x_1 + x_2 + x_3 = \sum_{i=0}^{k-1} 2^i(x_1[i] + x_2[i] + x_3[i]) = \sum_{i=0}^{k-1} 2^i(2c[i] + s[i]) = 2c + s$. The ripple carry adder can be replaced by a parallel prefixed adder to reduce the round complexity at the cost of $O(k \log k)$ communication (as done in ABY3). The procedure above may leave the participants with a few overflow bits. Assume we are working on a field Z_p, the adders will output $x = (x_1 + x_2 + x_3)$, which can take values in the range $[0, 3p-3]$. However, what we need is $(x \mod p) \in [0, p-1]$. The problem can be solved by repeatedly deducting p from x until the value of x is in the correct range. This can be done by executing the following Boolean circuit twice: $x \leftarrow x - (x > p) \cdot p$. The subtraction is done by executing a ripple borrow subtractor circuit. The above is all computed in a single circuit, and can be executed using any general-purpose 3-party computation using replicated binary sharing.

$[x]^B \rightarrow [x]^A$: we also use the protocol proposed in [31] for this conversion. In summary, from $[x]^B = (x_1, x_2, x_3)$, the parties obtain the shares $[x_1]^A = (x_1, 0, 0), [x_2]^A = (0, x_2, 0), [x_3]^A = (0, 0, x_3)$ non-interactively. For binary values x and y, the XOR operation can be replaced by arithmetic operations as $x \oplus y = x + y - 2xy$. To compute $[x_1]^A \oplus [x_2]^A \oplus [x_3]^A$, we just need to execute the above operations twice. The share conversion is secure against 1 malicious party.

$[x]^A \xrightarrow{1,2} \langle x \rangle^A$: Let $[x]^A = (x_1, x_2, x_3)$ be a replicated sharing of x, held by P_1, P_2 and P_3. P_1 and P_2 want to convert $[x]^A$ to $\langle x \rangle^A$. They can locally set their share to $x_1 + x_2$ and x_3 respectively. Note that this is only secure in the semi-honest setting: if one of them is malicious, he can modify his share arbitrarily, and there is no longer any replication that can be used to catch him later. The conversion between $[x]^A$ and $\langle x \rangle^A$ is needed to achieve the efficient three-party oblivious shuffling protocol in Section 4.3.

$\langle x \rangle^A \xrightarrow{1} [x]^A$: P_2 and P_3 hold a two-out-of-two sharing of x, and wish to create a replicated secret sharing that includes P_1. We describe a protocol that requires sending only 2 elements[5], and, more importantly, prevents P_1 from performing an additive attack. P_1 and P_2 agree on a random value r_1, and P_1 and P_3 agree on random value r_2. P_1 sets his own shares to (r_1, r_2); this prevents an additive attack, since the other parties already know these values. P_2 computes $x_1 - r_1$, P_3 computes $x_2 - r_2$, and they swap values. P_2 sets his shares to $(r_2, (x_2 - r_2) + (x_1 - r_1))$, and P_3 sets his to $((x_1 - r_1) - (x_2 - r_2), r_1)$.

$\mathcal{F}_{\langle X \rangle^A \xrightarrow{i} [X]^A}$ **- Share conversion up to an additive attack**

Inputs: Parties P_{i+1} and P_{i+2} hold $\langle X \rangle^A$, $X = \{x_1, ..., x_n\}$. P_i does not have input.
Functionality:

- Waits for the shares from P_{i+1} and P_{i+2}.
- Waits for the additive terms $D = \{d_1, ..., d_n\}$ from the malicious party (P_{i+1} or P_{i+2}).
- Reconstructs X and distributes the replicated shares $[X + D]$ to the three parties.

Figure 1: $\mathcal{F}_{\langle X \rangle^A \xrightarrow{i} [X]^A}$ Ideal Functionality

3 Set Intersection Cardinality Through Polynomial Interpolation

In this section we present a protocol for intersection cardinality, through polynomial interpolation. In Section 4.1 we use circuit-based techniques from generic secure computation to give two more constructions for cardinality. In all of our cardinality protocols, P_1 and P_2 begin by agreeing on an encryption key for a deterministic encryption scheme. They each encrypt their data and send it to P_3, who can find the intersection by simply comparing the ciphertexts sent by each party. (For the sake of intuition, it helps to ignore this step, and just think of P_3 as operating on cleartext data.) This is a large part of what allows us to construct efficient protocols: in the two party setting (or in a setting where all three parties have input), we cannot entrust the intersection computation to any one party. The main challenge that remains in our setting is to ensure that P_3 is honestly reporting the size of the intersection.

To prove that the intersection has the claimed size using polynomial interpolation, P_3 plays the prover in two, 2-round interactive proofs, one providing an upper bound, and the other providing a lower bound on the size of the intersection. The idea behind these proofs is as follows. When verifying a lower bound of z on the size of the set

[5]this doesn't really impact runtime, since we have no way of distributing the cost: when executing this on n shared elements, we will require 2 parties to each send n elements, rather than having each of the 3 parties send $2n/3$. Nevertheless, the improvement has an impact on the financial cost of running the protocol.

$\Pi_{\langle X\rangle^A \xrightarrow{i} [X]^A}$ **- Share conversion up to an additive attack**

Inputs: Parties P_{i+1} and P_{i+2} hold $\langle X\rangle^A$, $X = \{x_1, ..., x_n\}$. P_i does not have input.
Protocol:

1. P_i and P_{i+1} agree on a random key k_1.
2. P_i and P_{i+2} agree on a random key k_2.
3. P_i set his shares $[x_i] = (f_{k_2}(i), f_{k_1}(i))$.
4. P_{i+1} computes $u_i = \langle x_i\rangle_{i+1} - f_{k_1}(i)$.
5. P_{i+2} computes $v_i = \langle x_i\rangle_{i+2} - f_{k_2}(i)$.
6. P_{i+1} and P_{i+2} swap u_i and v_i, setting $w_i = u_i + v_i$.
7. P_{i+1} sets his shares $[x_i] = (f_{k_1}(i), w_i)$.
8. P_{i+2} sets his shares $[x_i] = (w_i, f_{k_2}(i))$.

Outputs: The parties output $[X]^A$.

Figure 2: $\Pi_{\langle X\rangle^A \xrightarrow{i} [X]^A}$ Share conversion protocol

$\mathcal{F}_{\textbf{PSI-CA}}$

Inputs: P_1 provides $X = \{x_1, ..., x_n\}$, P_2 provides $Y = \{y_1, ..., y_n\}$. P_3 provides no input.
Functionality:

- Waits for input $X = (x_1, ..., x_n)$ and $Y = (y_1, ..., y_n)$ from P_1 and P_2 respectively.
- If there are duplicated items in X or Y, sends abort to all parties.
- Else, gives output $|X \cap Y|$ to P_1, P_2, and P_3.

Figure 3: PSI-CA Ideal Functionality

union, P_1 and P_2 choose a random secret from a sufficiently large field, and secret share the value using a degree $(z-1)$ polynomial $p_2(x)$. Then, assuming input sets of size n, they each evaluate the polynomial at every point in their input set, and each sends the resulting n secret shares to P_3. If the union is smaller than the claimed lower bound of $z > |X \cup Y|$, P_3 will not have enough unique values to interpolate the polynomial $p_2(x)$ and learns nothing about $p_2(0)$. In order to pass the test, P_3 must choose a z such that $z \leq |X \cup Y|$, defining an upper bound of $(2n - z)$ on the size of the intersection. Similar techniques have been used in previous work [36, 39, 10].

When verifying a lower bound of z on the size of the intersection (which is an upper bound on the union), we use a similar idea with an additional twist. P_1 and P_2 again choose a random secret from a sufficiently large field, and secret share the value using a degree $z-1$ polynomial $p_1(x)$. They then encode their data such that two encodings of x reveal a share of this polynomial, $p_1(x)$, while a single encoding of x reveals nothing. Specifically, they use a 2-out-of-2 additive sharing of $p_1(x)$, where the randomness for the sharing is derived deterministically from the value of x: $(\mathcal{F}(k, x), \mathcal{F}(k, x) \oplus p_1(x))$. This allows each of P_1 and P_2 to generate one of the two shares, without knowing whether the other party will create and send the other share. P_3 will learn at most $|X \cap Y|$ shares of p_1, so if he has claimed a value of $z > |X \cap Y|$, he learns nothing about $p_1(0)$.

We present the full protocol for intersection cardinality in Figure 5. In Figure 4 we present the proof of union lower bound by itself, as we will use it in Section 4.1 in our "hybrid" protocol that combines this proof with circuit-based techniques. The reader might find it helpful to look at Figure 4 first, though the cardinality protocol of Figure 5 is self-contained. The cardinality protocol includes simultaneous proofs of the upper and lower bounds that were just informally described. Certain checks are performed by P_3 in order to prevent a selective failure attack by P_1 and P_2. If an element of V_1 or W_1 is encoded incorrectly in Steps 6 or 7, this is caught by P_3 in Step 12b or 12c when P_1 and P_2 reveal the polynomial that they used. Because P_3 needs to learn the randomness used in Steps 6 and 7 in order to perform this check, he sends a commitment to his challenge response in Step 10, before learning the randomness. If an element of V_2 or W_2 is encoded incorrectly in Steps 6 or 7, this is caught by P_3 in Step 9 if the element lies in the intersection, and is caught in Step 12a if the element is in the union; in either case, P_3 aborts in Step 12. Note that if the check in Step 9 were not performed, P_1 or P_2 could perform a selective failure attack to learn whether some particular element is in the intersection: given a bad encoding in V_2 or W_2, P_3 would abort if and only if the encoded element were not in the intersection.

PSI: When computing the actual intersection, rather than just the cardinality, P_3 can provide the encodings of the items in the intersection, together with a proof of the upper bound on the intersection size (or, equivalently, the union cardinality lower bound). Clearly P_3 can't add anything to the intersection, because at least one of P_1 and P_2 would recognize that the item was not in their input and reject. A similar comment applies when computing set union; it suffices to prove only the lower bound on the union, since nothing can be dropped from the union without detection. If a deterministic authenticated encryption scheme is used, there is no need for the lower bound proof on the union. In Figure 4 we present the union cardinality lower bound by itself. After receiving the claimed intersection and verifying the bound, the players simply verify that the claimed intersection has size that is consistent with that proof, and that the claimed intersection is a subset of their own input. The PSI protocol is secure with abort. It allows both parties to learn the output, in contrast to the protocols of [29, 38], which only allow one party to receive output.[6] The communication cost is linear.

Union count lower bound

Inputs: P_1 has $X = \{x_1, ..., x_n\}$, P_2 has $Y = \{y_1, ..., y_n\}$, P_3 does not have inputs.
Protocol:

1. P_1 and P_2 sample a common random key k by making a call to $\mathcal{F}_{\mathsf{coin}}$.
2. P_1 sends $S \leftarrow \{\mathcal{F}(k, x) \mid x \in X\}$ to P_3.
3. P_2 sends $T \leftarrow \{\mathcal{F}(k, y) \mid y \in Y\}$ to P_3.
4. P_3 verifies that all items in S and T are distinct, then sends $\tau = |S \cup T|$ to P_1 and P_2.
5. P_1 and P_2 agree on a random polynomial p of degree $(\tau - 1)$.
6. P_1 sends $V \leftarrow \{p(\hat{x}) \mid \hat{x} \in S\}$ to P_3.
7. P_2 sends $W \leftarrow \{p(\hat{y}) \mid \hat{y} \in T\}$ to P_3.
8. P_3 sets $V \leftarrow \{(\hat{x}, p(\hat{x})) \mid \hat{x} \in S\}$ and $W \leftarrow \{(\hat{y}, p(\hat{y})) \mid \hat{y} \in T\}$
9. If $\exists(\hat{x}, \hat{y}) \in S \times T : \hat{x} = \hat{y} \ \wedge \ p(\hat{x}) \neq p(\hat{y})$, P_3 interpolates a polynomial $\hat{p}$ from τ random values. Otherwise, P_3 interpolates a polynomial $\hat{p}$ from $V \cup W$. He computes $\hat{s} \leftarrow \hat{p}(0)$.
10. P_3 sends a commitment of $\hat{s}$ to P_1 and P_2.
11. P_1 and P_2 send p to P_3.
12. P_3 aborts if he receives different p or $p \neq \hat{p}$. Otherwise, he decommits $\hat{s}$ to P_1 and P_2.
13. The players compare the values received in the previous step, and abort if they differ. They also abort if $p(0) \neq \hat{s}$.

Output: All parties output τ.

Figure 4: The protocol to compute a lower bound on the union of the two input sets.

We note that a 2-round proof for the set union lower bound can be easily extracted, and used for computing set union. We do not present it separately, in order to preserve space.

Theorem 1 *Assuming* $(\mathsf{com}, \mathsf{decom})$ *is a computationally hiding, statistically binding commitment scheme, and that* $\mathcal{F}$ *is a secure PRP, the protocol* $\Pi_{\mathsf{PSI\text{-}CA}}$ *for computing the cardinality of the set intersection (Figure 5) securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{PSI\text{-}CA}}$ *with abort (Figure 3), under a single malicious corruption.*

We first describe a simulator for the cases where P_1 is corrupt. The simulator for P_2 is identical, so we omit the description and corresponding claim. Without loss of generality, we assume the malicious party outputs his entire view in the protocol. Simulated messages appear with '$\tilde{}$' above them.

1. $\tilde{k}_1, \tilde{k}_2, \tilde{\tau}$: $\mathcal{S}$ and P_1 sample random PRP keys $\tilde{k}_1, \tilde{k}_2$ with a call to $\mathcal{F}_{\text{coin}}$. After receiving encrypted inputs from P_1, if there is no duplication in the set of encrypted inputs, $\mathcal{S}$ inverts the PRP using $\tilde{k}_1$, recovering input set X', otherwise, S aborts and outputs whatever P_1 outputs. Otherwise, he submits X' to $\mathcal{F}$, and receives $\tilde{\tau} = |X' \cap Y|$. $\mathcal{S}$ hands $\tilde{\tau}$ to P_1 as the message from P_3.

2. $\tilde{p}_1, \tilde{p}_2, \tilde{c}_1, \tilde{c}_2$: $\mathcal{S}$ simulates the output of $\mathcal{F}_{\text{coin}}$, determining two random polynomials, $\tilde{p}_1$ and $\tilde{p}_2$, of degree $(z - 1)$ and $(2n - z - 1)$ respectively. After receiving V_1 and V_2 from P_1 (step 6), $\mathcal{S}$ verifies whether P_1 has

[6] Of course, in any protocol, the party receiving output can send the value to the first party, but in these protocols there is no way to verify that the received value is correct.

$\Pi_{\text{PSI-CA}}$: Polynomial-Based PSI Cardinality

Inputs: P_1 has $X = \{x_1, ..., x_n\}$, P_2 has $Y = \{y_1, ..., y_n\}$, P_3 does not have inputs.
Protocol:

1. P_1 and P_2 sample random keys k_1, k_2 by making two calls to $\mathcal{F}_{\mathsf{coin}}$..
2. P_1 sends $\widehat{X} \leftarrow \{\mathcal{F}(k_1, x) \mid x \in X\}$ to P_3.
3. P_2 sends $\widehat{Y} \leftarrow \{\mathcal{F}(k_1, y) \mid y \in Y\}$ to P_3.
4. P_3 verifies that all items in $\widehat{X}$ and $\widehat{Y}$ are distinct, then sends $\tau = |\widehat{X} \cap \widehat{Y}|$ to P_1 and P_2.
5. P_1 and P_2 jointly sample two random polynomials: p_1 of degree $(\tau - 1)$ (if $\tau = 0$, both set $p_1 \equiv 0$), and p_2 of degree $(2n - \tau - 1)$. They compute $s_1 \leftarrow p_1(0)$ and $s_2 \leftarrow p_2(0)$.
6. P_1 sends $V_1 \leftarrow \{\mathcal{F}(k_2, \hat{x}) \mid \hat{x} \in \widehat{X}\}$ and $V_2 \leftarrow \{p_2(\hat{x}) \mid \hat{x} \in \widehat{X}\}$ to P_3.
7. P_2 sends $W_1 \leftarrow \{\mathcal{F}(k_2, \hat{y}) \oplus p_1(\hat{y}) \mid \hat{y} \in \widehat{Y}\}$ and $W_2 \leftarrow \{p_2(\hat{y}) \mid \hat{y} \in \widehat{Y}\}$ to P_3.
8. P_3 sets
   (a) $V_1 \leftarrow \{(\hat{x}, \mathcal{F}(k_2, \hat{x})) \mid \hat{x} \in \widehat{X}\}$,
   (b) $W_1 \leftarrow \{(\hat{y}, \mathcal{F}(k_2, \hat{y}) \oplus p_1(\hat{y})) \mid \hat{y} \in \widehat{Y}\}$,
   (c) $V_2 \leftarrow \{(\hat{x}, p_2(\hat{x})) \mid \hat{x} \in \widehat{X}\}$,
   (d) $W_2 \leftarrow \{(\hat{y}, p_2(\hat{y})) \mid \hat{y} \in \widehat{Y}\}$
9. If $\exists(\hat{x}, \hat{y}) \in \widehat{X} \times \widehat{Y} : \hat{x} = \hat{y} \wedge p_2(\hat{x}) \neq p_2(\hat{y})$, P_3 interpolates polynomials $\hat{p}_1$ and $\hat{p}_2$ from τ and $(2n - \tau)$ random values respectively. Otherwise, P_3 computes: $Q_1 \leftarrow \{(a, b \oplus c) \mid (a, b) \in V_1,\ (a, c) \in W_1\}$ and $Q_2 \leftarrow \{(a, b) \mid (a, b) \in V_2 \cup W_2\}$ and then interpolates Q_1, Q_2, resulting in polynomials $\hat{p}_1$ and $\hat{p}_2$ respectively.
10. P_3 computes $\hat{s}_1 \leftarrow \hat{p}_1(0)$, $\hat{s}_2 \leftarrow \hat{p}_2(0)$, then sends $\mathsf{com}(\hat{s}_1)$, $\mathsf{com}(\hat{s}_2)$ to both P_1 and P_2.
11. P_1 and P_2 send k_2, p_1, p_2 to P_3. P_3 aborts if he sees that P_1 and P_2 sent him different values.
12. P_3 aborts if:
    (a) $\hat{p}_1 \neq p_1$ or $\hat{p}_2 \neq p_2$.
    (b) $V_1 \neq \{(\hat{x}, \mathcal{F}(k_2, \hat{x})) \mid \hat{x} \in \widehat{X}\}$
    (c) $W_1 \neq \{(\hat{y}, \mathcal{F}(k_2, \hat{y}) \oplus p_1(\hat{y})) \mid \hat{y} \in \widehat{Y}\}$

    Otherwise, P_3 decommits to $\hat{s}_1, \hat{s}_2$.
13. The players abort if $\hat{s}_1 \neq s_1$ or $\hat{s}_2 \neq s_2$.

Output: The players output τ.

Figure 5: $\Pi_{\text{PSI-CA}}$: A protocol for computing the size of the intersection, using polynomial interpolation to (simulataneously) prove both a lower and upper bound on the intersection size.

generated these sets correctly: $V_1 = \{\mathcal{F}(\widetilde{k}_2, \mathcal{F}(\widetilde{k}_1, x)) \mid x \in X'\}$ and $V_2 = \{p_2(\mathcal{F}(\widetilde{k}_1, x)) \mid x \in X'\}$. If these have been generated correctly, $\mathcal{S}$ computes $\widetilde{s}_1 = \widetilde{p}_1(0), \widetilde{s}_2 = \widetilde{p}_2(0)$. Otherwise, he sets $\widetilde{s}_1 = 0, \widetilde{s}_2 = 0$. $\mathcal{S}$ computes commitments to these values: $\widetilde{c}_1 = \mathsf{com}(\widetilde{s}_1)$, and $\widetilde{c}_2 = \mathsf{com}(\widetilde{s}_2)$. He hands P_1 the commitments as the messages from P_3.

3. $\widetilde{s}_1, \widetilde{s}_2$: $\mathcal{S}$ receives k_2, p_1, p_2 from P_1. If $p_i \neq \widetilde{p}_i$ for either i, or $k_2 \neq \widetilde{k}_2$, $\mathcal{S}$ sends abort to P_1 on behalf of P_3 and outputs the simulated messages. Otherwise, $\mathcal{S}$ opens the previous commitments to P_1: $\widetilde{s}_1 = \mathsf{decom}(\widetilde{c}_1)$ and $\widetilde{s}_2 = \mathsf{decom}(\widetilde{c}_2)$, and sends these to P_1.

4. $\mathcal{S}$ outputs the simulated messages.

Claim 1 *Assuming* $(\mathsf{com}, \mathsf{decom})$ *is a computationally hiding commitment scheme, then, for the simulator* $\mathcal{S}$ *described above and interacting with the functionality* $\mathcal{F}_{\mathsf{PSI\text{-}CA}}$ *on behalf of* P_1*,*

$$\left\{\mathrm{REAL}_{\pi_{\mathsf{PSI\text{-}CA}},\mathcal{A}(z)}(X, Y, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \stackrel{c}{\equiv} \left\{\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{PSI\text{-}CA}},\mathcal{S}(z)}(X, Y, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof:
Case 0: First, we consider the case where P_1 executes the protocol honestly. Because the functionality is deterministic, it suffices to consider the view of the adversary in both worlds, rather than analyzing the joint-distribution of his view with the honest output:

$$\left\{\mathrm{REAL}_{\pi_{\mathsf{PSI\text{-}CA}},\mathcal{A}(z)}(X, Y, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{k_1, k_2, \tau, p_1, p_2, \mathsf{com}(s_1), \mathsf{com}(s_2), s_1, s_2\}$$

$$\left\{\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{PSI\text{-}CA}},\mathcal{S}(z)}(X, Y, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{k}_1, \widetilde{k}_2, \widetilde{\tau}, \widetilde{p}_1, \widetilde{p}_2, \mathsf{com}(\widetilde{s}_1), \mathsf{com}(\widetilde{s}_2), \widetilde{s}_1, \widetilde{s}_2\}$$

The security of $\mathcal{F}_{\mathrm{coin}}$ ensures that k_1, k_2, p_1, and p_2 are distributed uniformly at random, and that they are independent from other messages. The simulation, $\widetilde{k}_1, \widetilde{k}_2$, $\widetilde{p}_1, \widetilde{p}_2$ is therefore perfect. τ and $\widetilde{\tau}$ are fully determined by the first message of P_1, and the input of the honest P_2. Since $\mathcal{S}$ correctly extracts P_1's input in Step 2 using the PRP key, τ and $\widetilde{\tau}$ are identically distributed. In Step 6, the two polynomials p_1 and p_2 are sampled uniformly at random, thus $s_i = p_i(0)$ and $\widetilde{s}_i = \widetilde{p}_i(0)$ are identically distributed. As our function is a deterministic one, the indistinguishability between the two distributions is reduced to the indistinguishability of the commitment messages. Thus, the joint distributions in both worlds are computationally indistinguishable when P_1 is honest.

Case 1: If P_1 sends duplicated ciphertexts in Step 4, abort happens in both the real and ideal worlds. The joint distributions in the real and ideal worlds are $\{k_1, k_2\}$ and $\{\widetilde{k}_1, \widetilde{k}_2\}$, respectively, and are identically distributed.

Case 2: If P_1 deviates in Step 6, in the ideal world, this will be detected immediately and the simulator sends abort at Step 11 or 12, as the simulator knows the values of p_1, p_2, k_1, k_2, and X. The joint distribution in the ideal world is $\{\widetilde{k}_1, \widetilde{k}_2, \widetilde{\tau}, \widetilde{p}_1, \widetilde{p}_2, \mathsf{com}(0), \mathsf{com}(0), \perp\}$. In the real world, P_1 will be caught by P_3 in Step 9 or Step 12, and P_3 will abort at Step 12. Note that in Step 12, P_3 verifies the correctness of the messages sent to him in Steps 6 and 7 by P_1 and P_2, using k_2, p_1, and p_2. The joint distribution in the real world is $\{k_1, k_2, \tau, p_1, p_2, \mathsf{com}(s_1), \mathsf{com}(s_2), \perp\}$. The indistinguishability of these distributions reduces to the hiding property of the commitment scheme.

Case 3: If P_1 deviates in Step 11 by sending the wrong k_2, p_1, or p_2 to P_3. This will be detected in both the ideal and the real worlds. The joint distributions in the ideal and real world are $\{\widetilde{k}_1, \widetilde{k}_2, \widetilde{\tau}, \mathsf{com}(\widetilde{s}_1), \mathsf{com}(\widetilde{s}_2), \perp\}$ and $\{k_1, k_2, \tau, \mathsf{com}(s_1), \mathsf{com}(s_2), \perp\}$ respectively. Follow the same arguments in case 0, the two joint distributions are computationally indistinguishable.

These cases cover all possible behaviors of P_1, proving that the adversarial views are indistinguishable in the two worlds. ■

We now describe a simulator for P_3.

1. $\widetilde{S}, \widetilde{T}$: The simulator $\mathcal{S}$ queries $\mathcal{F}$ to learn the size of the intersection. It then simulates the first messages from P_1 and P_2 by choosing random strings for each encoding, subject to the constraint that the intersection is of appropriate size. Let the messages be two sets $\widetilde{S}$ and $\widetilde{T}$ (from P_1 and P_2 respectively).

2. $\widetilde{V}_1, \widetilde{V}_2, \widetilde{W}_1, \widetilde{W}_2$: $\mathcal{S}$ receives τ from P_3, who sends the value to both P_1 and P_2. If P_3 sends different values to P_1 and P_2, $\mathcal{S}$ aborts and outputs the partial view. Otherwise, $\mathcal{S}$ chooses two random polynomials of degree $(\widetilde{\tau} - 1)$ and $(2n - \widetilde{\tau} - 1)$. Let the polynomials be p_1 and p_2 respectively. $\mathcal{S}$ computes $s_1 = p_1(0)$ and $s_2 = p_2(0)$. $\mathcal{S}$ samples a random key k_2 and computes the following messages: $\widetilde{V}_1 \leftarrow \{\mathcal{F}(k_2, \hat{x}) \mid \hat{x} \in \widetilde{S}\}$, $\widetilde{V}_2 \leftarrow \{p_2(\hat{x}) \mid \hat{x} \in \widetilde{S}\}$, $\widetilde{W}_1 \leftarrow \{\mathcal{F}(k_2, \hat{y}) \oplus p_1(\hat{y}) \mid \hat{y} \in \widetilde{T}\}$, and $\widetilde{W}_2 \leftarrow \{p_2(\hat{y}) \mid \hat{y} \in \widetilde{T}\}$. $\mathcal{S}$ hands the values to P_3.

3. $\widetilde{k}_2, \widetilde{p}_1, \widetilde{p}_2$: $\mathcal{S}$ receives two commitments from P_3: $\mathsf{com}(\widetilde{s}_1)$ and $\mathsf{com}(\widetilde{s}_2)$. $\mathcal{S}$ hands P_3 duplicates of the key k_2 and the polynomials p_1, p_2, simulating the messages from P_1 and P_2.

4. If P_3 aborts, or gives wrong decommitments to $\mathsf{com}(\widetilde{s}_1)$ or $\mathsf{com}(\widetilde{s}_2)$, or $\widetilde{s}_1 \neq s_1$, or $\widetilde{s}_2 \neq s_2$, $\mathcal{S}$ submits abort to the ideal functionality, sends $\perp$ to P_3 on behalf of P_1 and P_2, and outputs the simulated transcript.

Claim 2 *Assuming $\mathcal{F}$ is a secure pseudorandom permutation, and that* $(\mathsf{com}, \mathsf{decom})$ *is a statistically binding commitment scheme, then, for simulator $\mathcal{S}$ corrupting party P_3 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{PSI\text{-}CA}}$,*

$$\left\{\text{REAL}_{\pi_{\mathsf{PSI\text{-}CA}},\mathcal{A}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \stackrel{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{PSI\text{-}CA}},\mathcal{S}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof:

Hybrid$_0$: Real execution.

Hybrid$_1$: Same as hybrid$_0$, except that the PRP is replaced with random encoding.

Hybrid$_2$: Ideal execution.

It's clear that **Hybrid**$_0$ and **Hybrid**$_1$ are computationally indistinguishable by a reduction to the PRP. We prove that **Hybrid**$_1$ and **Hybrid**$_2$ are also computationally indistinguishable.

Case 0: We first consider the case where P_3 executes the protocol honestly. Because the functionality is deterministic, it suffices to consider the view of the adversary in both worlds, in place of analyzing the joint-distribution of his view with the honest output.

$$\left\{\text{HYBRID}_1{}_{\pi_{\mathsf{PSI\text{-}CA}},\mathcal{A}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widehat{X},\widehat{Y},V_1,V_2,W_1,W_2,k_2,p_1,p_2\}$$

$$\left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{PSI\text{-}CA}},\mathcal{S}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{S},\widetilde{T},\widetilde{V}_1,\widetilde{V}_2,\widetilde{W}_1,\widetilde{W}_2,\widetilde{k}_2,\widetilde{p}_1,\widetilde{p}_2\}$$

The reader can verify by inspection that the following two distributions are identical: $\{V_1,V_2,W_1,W_2,k_2,p_1,p_2 \mid \widehat{X},\widehat{Y}\}$ and $\{\widetilde{V}_1,\widetilde{V}_2,\widetilde{W}_1,\widetilde{W}_2,\widetilde{k}_2,\widetilde{p}_1,\widetilde{p}_2 \mid \widetilde{S},\widetilde{T}\}$ (each of the messages are distributed uniformly at random and independent from one another).

Case 1: Assume P_3 cheats in Step 4 by sending different $\widetilde{\tau}$ to P_1 and P_2. Both parties immediately abort in both worlds. The joint distributions generated in the hybrid$_1$ and ideal world are $\{\widehat{X},\widehat{Y},\perp\}$ and $\{\widetilde{S},\widetilde{T},\perp\}$ respectively. They are identically distributed.

Case 2: If P_3 cheats in Step 4 by providing the same incorrect $\widetilde{\tau}$ to P_1 and P_2, all parties continue up to Step 11, and the partial views up to this point in the hybrid$_1$ and ideal world are $\{\widehat{X},\widehat{Y},V_1,V_2,W_1,W_2,k_2,p_1,p_2\}$ and $\{\widetilde{S},\widetilde{T},\widetilde{V}_1,\widetilde{V}_2,\widetilde{W}_1,\widetilde{W}_2,\widetilde{k}_2,\widetilde{p}_1,\widetilde{p}_2\}$ respectively. As argued in case 0, these partial views are computationally indistinguishable. It remains to argue about the output of the honest parties. Note that in the ideal world, the honest parties output $\perp$, as this deviation is always detected by $\mathcal{S}$, who then tells the trusted party to abort. On the other hand, in the hybrid-world, if P_3 deviates in Step 4 by sending both parties the wrong intersection size, $\tau \neq |\widetilde{S}\cap\widetilde{T}|$, he will not be able to correctly interpolate both $\widetilde{p}_1$ and $\widetilde{p}_2$ in Step 9: if $\widetilde{\tau} < |\widetilde{S}\cap\widetilde{T}|$, P_3 will not be able to recover the value of $\widetilde{p}_2(0)$, and if $\widetilde{\tau} > |\widetilde{S}\cap\widetilde{T}|$, he will not be able to recover the value of $\widetilde{p}_1(0)$. In Step 11 or Step 12, if he decides to abort, then in both worlds the joint distributions are $\{\widehat{X},\widehat{Y},V_1,V_2,W_1,W_2,k_2,p_1,p_2,\perp\}$ and $\{\widetilde{S},\widetilde{T},\widetilde{V}_1,\widetilde{V}_2,\widetilde{W}_1,\widetilde{W}_2,\widetilde{k}_2,\widetilde{p}_1,\widetilde{p}_2,\perp\}$; they are identically distributed If P_3 does not abort in Step 11 and Step 12, assuming $(\mathsf{com},\mathsf{decom})$ is statistically binding, P_3 has at most $1/|\mathbf{F}|$ chance of successfully guessing $p_1(0)$ or $p_2(0)$, as $\widetilde{p}_1(0)$ and $\widetilde{p}_2(0)$ are randomly distributed in $\mathbf{F}$. If P_3 guesses the wrong value, the joint distributions are identical in both worlds, as every party aborts. (Note that the probability that P_3 guesses these values correctly when $\tau \neq |X\cap Y|$ is independent of his view.) However, if P_3 guesses $\widetilde{p}_i(0)$ correctly, the joint distributions are distinguishable: in the real world, this would go undetected, and the honest parties might output some $\tau \neq |X\cap Y|$. This is not possible in the ideal world, and thus the simulator $\mathcal{S}$ fails to simulate P_3's behavior. But, as argued above, this happens with probability $1/|\mathbf{F}|$.

Case 3: If P_3 is honest in Step 4, but deviates at any other steps, the joint distributions are computationally indistinguishable. (Note that after Step 4, all the remains for P_3 to do is to interpolate the polynomials, and send proper commitments and decommitments to their roots.)

In conclusion, the joint distributions in both worlds are computationally indistinguishable. ■

4 Circuit-based protocols

4.1 Circuit-based Intersection cardinality

The protocol of Section 3 has low communication cost, but requires $O(n \log^2 n)$ computational steps by all parties. We present a construction using techniques from generic 3-party computation that requires more communication, but less computation. It also allows us to compute on the payloads of the items in the intersection. Interestingly, we also provide a hybrid protocol that offers a third point in the continuum. In this hybrid protocol, we remove $2n - 2z$ comparison gates from our circuit by using the proof of the union lower bound from Section 3 (Figure 4) (We note that to prove the union lower bound, the three parties start at Step 5 in Figure 4. At this point, P_3 has received $\{\mathcal{F}(k, x), x \in X\}$ and $\{\mathcal{F}(k, y), y \in Y\}$ from P_1 and P_2 respectively, and has sent the size of the intersection to P_1 and P_2. In fact, a union lower bound proof with inputs as X and Y will allow a malicious party to send $\mathcal{F}(k, z^*)$ instead of $\mathcal{F}(k, z)$ where $z \in X$ or $z \in Y$. By using the existing prp values and starting the proof at Step 5, the input consistency is guaranteed.). Both variants of our circuit-based protocol are described in Figure 8.

$\mathcal{F}_{\text{unionLB}}$

Inputs: All parties provide t and $[Z]^A$ where $Z = X||Y$.
Functionality:

- Wait for shares $[Z]^A$ from P_1, P_2, and P_3. If the shares are valid, reconstruct Z. Else, send abort.
- If the items in X are not all distinct, or the items in Y are not all distinct, send abort.
- If the parties don't agree on t, send abort.
- If $t \leq |X \cup Y|$, output true to all players.
- If $t > |X \cup Y|$, output false to all players.

Figure 6: Verify a union count lower bound

$\mathcal{F}_{\text{f(PSI)}}$

Inputs: P_1 provides $X = \{x_1, ..., x_n\}$, P_2 provides $Y = \{y_1, ..., y_n\}$. P_3 provides no input.
Functionality:

- Waits for input X and Y from P_1 and P_2 respectively.
- If there are duplicated items in X or Y, sends abort to all parties.
- Else, gives output $|X \cap Y|$ to all parties and $f(X \cap Y)$ to P_1 and P_2.

Figure 7: f(PSI) Ideal Functionality

In the two-party setting, one naive way of computing the cardinality of the intersection is as follows. The two parties each sort their inputs locally, and then perform a generic secure two-party computation of the following algorithm:

- Obliviously merge the two input arrays.
- Obliviously scan the input, comparing neighbors for equality, and counting the number of duplicates.

The oblivious merge requires $O(n \log n)$ AND gates, which we can avoid in the three-party setting. The main tool we employ is a cheap, linear-time,[7] three-party protocol for sorting the input according to a permutation specified by P_3. We defer the description of this sub-routine until Section 4.3. The users send encodings of their inputs to P_3, as in the protocol of Section 3, and P_3 finds the items in the intersection. (If either party sent duplicate items, he aborts). He then chooses a permutation on the $2n$ items that a) places all matching encodings in the front of the array (preserving the duplicate values), and b) sorts the remaining encodings according to their lexicographic ordering. P_3 reports $z = |S \cap T|$. The three parties then perform a generic computation that

- verifies the equality of neighboring pairs for the first $z = |S \cap T|$ pairs, and

[7] Sorting requires $O(n \log n)$ time, of course, but we are measuring the number of secure, interactive operations, and the sorting is done locally.

π_{cb}: Circuit-Based PSI computations

Inputs: P_1 has $X = \{x_1, ..., x_n\}$, P_2 has $Y = \{y_1, ..., y_n\}$, P_3 does not have inputs.
For $f(\mathsf{PSI})$: P_1 provides payload values $D^{(1)} = \{d_1^{(1)}, \ldots, d_n^{(1)}\}$ and P_2 provides $D^{(2)} = \{d_1^{(2)}, \ldots, d_n^{(2)}\}$
Protocol:

1. P_1 and P_2 sample a random PRP key k by making a call to $\mathcal{F}_{\mathsf{coin}}$.
2. P_1 computes $V_1 \equiv \{\mathcal{F}(k, x) | x \in X\}$ and distributes it as replicated shares among the three parties.
   For $f(\mathsf{PSI})$: P_1 also distributes shares of the payloads.
3. P_2 computes $V_2 \equiv \{\mathcal{F}(k, y) | y \in Y\}$ and distributes it as replicated shares among the three parties.
   For $f(\mathsf{PSI})$: P_2 also distributes shares of the payloads.
4. The parties open $Z = V_1 || V_2$ to P_3.
5. P_3 verifies that all items in V_1 are distinct, and that all items in V_2 are distinct. If this is not true, he aborts.
6. P_3 fixes a permutation π that moves items in the intersection ($Z_1 = V_1 \cap V_2$) to the top, placing each item next to its duplicate, and that moves the rest (Z_2) to the bottom, in sorted order. The three parties call $\mathcal{F}_{\mathsf{Shuffle}}$ to shuffle the shares according to π.
   For $f(\mathsf{PSI})$: the payloads are shuffled along with their indices.
7. P_3 sends the size of the intersection, t, to P_1 and P_2. P_1 and P_2 abort if they received differing values.
8. For Z_1, the parties verify that there are t duplicate pairs, using secure arithmetic comparisons. (The relevant functionalities are defined in Section 2.3.)
   - They make t calls to $\mathcal{F}_{\text{coin}}$ to receive $R = \{r_1, ..., r_t\}$.
   - They run $\mathcal{F}_{\text{CheckZero}}([\sum_{i=1}^{t} r_i(z_{2i} - z_{2i-1})])$, where $z_i \in Z^{(1)}$, and abort if the output is False.
9. **Circuit-based protocol:**
   For Z_2, the parties make a call to $\mathcal{F}_{[x]^A \to [x]^B}$ to convert $[Z_2]^A$ to $[Z_2]^B$. They then run a sequence of 3PC comparison circuits, verifying that the items are in sorted order. If not, they abort.
   Hybrid protocol:
   All parties make a call to $\mathcal{F}_{\mathsf{unionLB}}$ with input $([Z]^A, t)$. If the output is false, they abort.
10. **For $f(psi)$:** For each pair of duplicates in Z_1, the parties use the replicated sharings as input to a circuit for f. The output of the circuit is $[f(psi)]$. Players reveal f(psi) to P_1 and P_2.

Output:
For $f(psi)$: P_1 and P_2 output the result of f and $t = |Z_1|/2$. P_3 outputs $t = |Z_1|/2$.
For set cardinality: P_1, P_2, and P_3 output $|Z_1|/2$.

Figure 8: We give four protocols in this box: two for computing PSI cardinality, and and two for computing arbitrary f on the payloads of the intersecting items. The difference in the protocol variants (in either computation) lies only in how P_1 and P_2 verify the upper-bound on the union size: using Boolean comparisons to verify a strict ordering, or through polynomial interpolation.

- verifies that item i is strictly greater than item $i-1$, for $i \in \{2z+2, \ldots, 2n\}$.

The resulting circuit requires a single batched equality check of arithmetic values for the z items in the intersection, and $2n-2z$ Boolean comparison circuits. To verify the equality of z pairs, the parties need to communicate only $O(1)$ field elements. To verify the order for $(2n-2z)$ items, $O(2n-2z)$ bits are required. Both have linear runtime complexity. Note that as z goes from 0 to n, the number of required circuits goes from $2n$ to 0.

Comparing the protocols: We provide concrete comparisons in Section 6, and give some intuition for the trade-offs here. We compare the protocols based on the three criteria: computational complexity, communication cost, and round complexity. In terms of computational cost, the circuit-based protocol has linear computational complexity while the polynomial-based and hybrid ones run in $O(n \log^2 n)$ time due to the polynomial interpolation and evaluation subroutines [3]. All the three protocols have linear communication cost, in which the polynomial-based approach requires the least bandwidth and the circuit-based approach requires the most. This is due to the bandwidth required by the share conversion and comparison circuits. The polynomial-based protocol has the least number of rounds while the circuit-based has the most. We note that they all have a constant number of rounds.

When the input size is small, the number of rounds dominates the total runtime, due to network latency. When the input size is large, the circuit-based protocol performs best in LAN setting, since the network is not an issue. In the WAN setting, the circuit-based protocol is better only when the input set size is very large (e.g. 2^{20}), as then the network latency is not the dominate cost. (This is demonstrated experimentally, in Table 1). Interestingly, the parties do not need to commit to their choice of protocol until after they have learned the intersection size: all three protocols begin the same way, with P_3 determining and reporting the size on encoded values. This flexibility allows the parties to pick the protocol that works best for them according to their available resources and network configuration.

4.2 Computing on the payloads of intersecting indices

Our protocol for $f(PSI)$, where f depends only on the payloads, $f(PSI) \equiv f(D)$ where $D \equiv \{(d_1, d_2) \mid \exists w \in X \cap Y : (w, d_1) \in (X, D^{(1)}) \wedge (w, d_2) \in (X, D^{(2)})\}$, also appears in Figure 8. The modifications to the circuit-based cardinality protocol are minimal, and marked in green. As before, the parties begin by agreeing on a PRP key, k, and use it to deterministically encode their inputs: for input pair (x, d), where d is the payload and x is the index, the party computes $\hat{x} = \mathcal{F}(k, x)$, and then creates a replicated sharing of $(\hat{x}, d)$. The main insight is that this sharing can be viewed as a commitment to the input values (due to the replication). With these commitments in place, the parties can securely and consistently

1. Open the encoded indices to P_3 for determining the intersection, and the sorting permutation π.
2. Provide input to $\mathcal{F}_{\mathsf{Shuffle}}$.
3. Use the shares as input to a three-party computation on the payloads of the indices in the intersection.

It is instructive to consider why we can only use this to compute on the payloads, and not on the indices themselves. For P_3 to determine the permutation π, P_1 and P_2 need to send encoded indices. But to compute some function f on these values, they need to supply the plaintext indices to f. The replicated sharing no longer gives a guarantee that the input to f is consistent with the encodings sent to P_3. We explore ways of providing this consistency guarantee in Section 5.

Theorem 2 *The protocols of Figure 8 for computing PSI cardinality securely realize the ideal functionality $\mathcal{F}_{\mathsf{PSI\text{-}CA}}$ (Figure 3) with abort, under a single malicious corruption. The variants for computing on the payloads of the intersecting items securely realize the ideal functionality $\mathcal{F}_{\mathsf{f(PSI)}}$ (Figure 7).*

We begin by simulating P_1 in both $f(PSI)$ protocols, and argue that the protocol remains secure when P_1 is malicious.

1. $\widetilde{k}$: $\mathcal{S}$ samples $\widetilde{k}$ uniformly at random and sends it to P_1.
2. $\mathcal{S}$ extracts the input $(X', D'^{(1)})$ of P_1 from the shares sent to P_2 and P_3. If there are any inconsistencies, or if P_1 sends any duplicates, $\mathcal{S}$ sets $\mathsf{abort} = 1$. Otherwise, $\mathcal{S}$ sends the input to the ideal functionality, and receives $f(D)$, and $t = |X' \cap Y|$. We note that $D \equiv \{(d_1, d_2) | \exists w \in X' \cap Y : (w, d_1) \in (X', D'^{(1)}) \wedge (w, d_2) \in (X, D^{(2)})\}$.
3. $[\widetilde{V}_2], [\widetilde{D}^{(2)}]$: $\mathcal{S}$ sends random field elements to simulate the input shares of P_2.

4. If P_1 sends incorrect shares during the opening of Z in Step 4, $\mathcal{S}$ sends abort to the functionality, outputs the partial view of P_1, and terminates.

5. $[\widetilde{Z_1}], [\widetilde{Z_2}]$: If P_1 sends incorrect shares as input to $\mathcal{F}_{\mathsf{Shuffle}}$, $\mathcal{S}$ sends abort to the functionality, outputs the partial view of P_1, and terminates. Otherwise, $\mathcal{S}$ sends random field elements to simulate the output of $\mathcal{F}_{\mathsf{Shuffle}}$.

6. $\widetilde{t}$: $\mathcal{S}$ sends the value received from the ideal functionality, $\widetilde{t} = |X \cap Y|$ to P_1. If P_1 reports a different value while verifying $\widetilde{t}$ with P_2, $\mathcal{S}$ sends abort to the functionality, outputs the partial view of P_1, and terminates.

7. $\widetilde{R}, \widetilde{b}_1$: $\mathcal{S}$ simulates the outputs of $\mathcal{F}_{\text{coin}}$ using random bits. If P_1 submits $\zeta \neq \sum_{i=1}^{t} r_i[(z_{2i} - z_{2i-1})]$ to $\mathcal{F}_{\text{CheckZero}}$, $\mathcal{S}$ sends $\widetilde{b}_1 = \mathsf{false}$ to P_1, sends abort to the functionality, outputs the partial view of P_1, and terminates. Otherwise, he returns $\widetilde{b}_1 = \mathsf{true}$ as the output of $\mathcal{F}_{\text{CheckZero}}$.

8. **Circuit-based protocol:** $\mathcal{S}$ simulates the output of the share conversion, $[Z_2]^A \rightarrow [Z_2]^B$, by sending random Boolean values for the replicated shares. If P_1 submits correct shares to the computations of the comparison circuits, $\mathcal{S}$ simulates the output by sending $\widetilde{b}_2 = \mathsf{true}$.

   Hybrid protocol: $\widetilde{b}_3$: If P_1 sends incorrect shares to $\mathcal{F}_{\mathsf{unionLB}}$, $\mathcal{S}$ simulates the output of $\mathcal{F}_{\mathsf{unionLB}}$ by sending abort to P_1. Otherwise, $\mathcal{S}$ sends $\widetilde{b}_3 = \mathsf{true}$.

9. For $f(PSI)$, $\widetilde{f(psi)}$: If P_1 alters his shares $[D]$ before sending them to the functionality that compute the circuit $f(psi)$, $\mathcal{S}$ aborts and outputs the partial view. Else, $\mathcal{S}$ hands P_1 f(psi) (obtained from the ideal functionality).

Claim 3 *For the simulator $\mathcal{S}$ corrupting party P_1 as described above, and interacting with the functionality $\mathcal{F}_{f(psi)}$,*

$$\left\{\text{HYBRID}_{\pi_{cb},\mathcal{A}(z)}\left((X, D^{(1)}), (Y, D^{(2)}), \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{F}_{f(psi)},\mathcal{S}(z)}\left((X, D^{(1)}), (Y, D^{(2)}), \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof:
Case 0: First, we consider the case where P_1 executes the protocol honestly. In this case, the joint distributions in the hybrid and in the ideal worlds are:

For circuit-based protocol:

$\{\text{HYBRID}_{\pi_{\mathsf{cb}},\mathcal{A}(z)}((X, D^{(1)}), (Y, D^{(2)}), \kappa)\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{k, [V_2], [D^{(2)}], [\pi(D^{(1)}||D^{(2)})], [Z_1], [Z_2], t, R, b_1, [Z_2]^B, b_2, o_1, o_2, o_3\}$

$\{\text{IDEAL}_{\mathcal{F}_{\mathsf{f(psi)}},\mathcal{S}(z)}((X, D^{(1)}), (Y, D^{(2)}), \kappa)\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{k}, [\widetilde{V}_2], [\widetilde{D}^{(2)}], [\pi(\widetilde{D}^{(1)}||\widetilde{D}^{(2)})], [\widetilde{Z}_1], [\widetilde{Z}_2], \widetilde{t}, \widetilde{R}, \widetilde{b}_1, [\widetilde{Z}_2]^B, \widetilde{b}_2, \widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3\}$

For hybrid protocol:

$\{\text{HYBRID}_{\pi_{\mathsf{cb}},\mathcal{A}(z)}((X, D^{(2)}), (Y, D^{(2)}), \kappa)\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{k, [V_2], [D^{(2)}], [\pi(D^{(1)}||D^{(2)})], [Z_1], [Z_2], t, R, b_1, b_3, o_1, o_2, o_3\}$

$\{\text{IDEAL}_{\mathcal{F}_{\mathsf{f(psi)}},\mathcal{S}(z)}((X, D^{(1)}), (Y, D^{(2)}), \kappa)\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{k}, [\widetilde{V}_2], [\widetilde{D}^{(2)}], [\pi(\widetilde{D}^{(1)}||\widetilde{D}^{(1)})], [\widetilde{Z}_1], [\widetilde{Z}_2], \widetilde{t}, \widetilde{R}, \widetilde{b}_1, \widetilde{b}_3, \widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3\}$

As k, $\widetilde{k}$,R, $\widetilde{R}$, and the shares are sampled uniformly at random and independently from one another, t and $\widetilde{t}$ are identical ($\mathcal{S}$ can extract P_1's input), $b_i = \widetilde{b}_i = \mathsf{true}$, and the outputs are identical, the joint distributions in both worlds are identically distributed.

Case 1: Now we consider the case where P_1 deviates from the protocol. It is easy to verify that if P_1 deviates at any points prior to Step 8, abort happens in both worlds and the partial views are identically distributed. If P_1 does not deviate from the protocol before Step 8, the partial views up to Step 8 in the hybrid and ideal worlds are $\{k, [V_2], [D^{(2)}], [\pi(D^{(1)}||D^{(1)})], [Z_1], [Z_2], t\}$ and $\{\widetilde{k}, [\widetilde{V}_2], [\widetilde{D}^{(2)}], [\pi(\widetilde{D}^{(1)}||\widetilde{D}^{(1)})], [\widetilde{Z}_1], [\widetilde{Z}_2], \widetilde{t}\}$ respectively. These partial views are distributed uniformly at random and independent from the rest of the joint distributions, thus, we can omit these partial views in the following analysis.

From Step 8 to Step 10, the only way that P_1 can cheat is to submit the wrong shares to one of the following functionalities: $\mathcal{F}_{\text{CheckZero}}$, $\mathcal{F}_{[x]^A\rightarrow[x]^B}$, the 3PC comparison circuit (or $\mathcal{F}_{\mathbf{unionLB}}$, depending on which is being used), or the 3PC circuit to compute $f(psi)$. In the ideal world, $\mathcal{S}$ detects the cheating right away and aborts. In the hybrid world, the same thing happens, except for the case of $\mathcal{F}_{\text{CheckZero}}$, where abort happens with the probability of $1 - 1/|F|$. Thus, the joint distributions in both world are statistically close if P_1 deviates at one of these steps.

In conclusion, the joint distributions in both worlds are computationally indistinguishable. ■

We now present the simulation of P_3.

1. $[\widetilde{V}_1], [\widetilde{D^{(1)}}], [\widetilde{V}_2], [\widetilde{D^{(2)}}]$: $\mathcal{S}$ queries the ideal functionality and receives $t = |X \cap Y|$. $\mathcal{S}$ chooses $2n - t$ random strings from the domain of the PRP, without replacement. He duplicates the first t strings and create two sets, $\widetilde{V}_1$, $\widetilde{V}_2$, each with one copy of the duplicated items, and $(n - t)$ other items. He randomly shuffles V_i, creates random replicated sharings of these elements, and sends shares to P_3, on behalf of P_1 and P_2. He also sends P_3 random strings to simulate the data shares $[\widetilde{D^{(1)}}]$, $[\widetilde{D^{(2)}}]$ that P_3 receives from P_1 and P_2.

2. $\widetilde{Z}$: $\mathcal{S}$ simulates the opening of $\widetilde{Z}$ by sending the missing share of each value on behalf of both P_1 and P_2.

3. $[\pi(\widetilde{Z})], [\pi(\widetilde{D}^{(1)}||\widetilde{D}^{(2)})]$: $\mathcal{S}$ receives π from P_3, and uses π to shuffle Z, computing $\pi(\widetilde{z})$. $\mathcal{S}$ simulates the output of $\mathcal{F}_{\mathsf{Shuffle}}$ by creating new replicated shares of $\pi(\widetilde{Z})$; shares are random strings. $\mathcal{S}$ sends P_3 random strings as shares $[\pi(\widetilde{D}^{(1)}||\widetilde{D}^{(2)})]$. $\mathcal{S}$ then observes the message t P_3 sends to P_1 and P_2 in Step 7, indicating the supposed intersection size. If P_3 sends different t to each of P_1 and P_2, $\mathcal{S}$ sends abort to the trusted party, and outputs the partial view. If P_3 sends the same t to P_1 and P_2, but Z_1 is not in the correct format, $\mathcal{S}$ sets $\mathsf{abort}_1 = 1$. If Z_2 is not in strictly increasing order, he sets $\mathsf{abort}_2 = 1$. If P_3 sends the same $t' < t$ to P_1 and P_2, $\mathcal{S}$ sets $\mathsf{abort}_3 = 1$.

4. $\widetilde{R}, \widetilde{b}_1$: $\mathcal{S}$ simulates the outputs of $\mathcal{F}_{\text{coin}}$ using random bits. If P_3 submits $\zeta \neq \sum_{i=1}^{t} r_i[(z_{2i} - z_{2i-1})]$ to $\mathcal{F}_{\text{CheckZero}}$ or if $\mathsf{abort}_1 = 1$, $\mathcal{S}$ sends $\widetilde{b}_1 = \mathsf{false}$ to P_3, sends abort to the functionality, outputs the partial view of P_3, and terminates. Otherwise, he returns $\widetilde{b}_1 = \mathsf{true}$ as the output of $\mathcal{F}_{\text{CheckZero}}$.

5. **Circuit-based protocol:** $[\widetilde{Z}_2]^B, \widetilde{b}_2$: $\mathcal{S}$ simulates the output of the share conversion, $[Z_2]^A \rightarrow [Z_2]^B$, by sending random Boolean values for the replicated shares. If P_3 submits correct shares to the computations of the comparison circuits and $\mathsf{abort}_2 = 0$, $\mathcal{S}$ simulates the output by sending $\widetilde{b}_2 = \mathsf{true}$. Otherwise, $\mathcal{S}$ hands P_3 $\widetilde{b}_2 = \mathsf{false}$.

   Hybrid protocol: $\widetilde{b}_3$: If P_3 submits correct shares to $\mathcal{F}_{\mathsf{unionLB}}$, and $\mathsf{abort}_3 = 0$, $\mathcal{S}$ simulates the output by sending $\widetilde{b}_3 = \mathsf{true}$. Otherwise, $\mathcal{S}$ hands P_3 $\widetilde{b}_3 = \mathsf{false}$.

   Let $[D]$ be the shares of the data that go with Z_1.

6. For $f(PSI)$: If P_3 alters his shares $[D]$ before sending them to the functionality that computes the circuit $f(D)$, $\mathcal{S}$ aborts and outputs the partial view. Else, $\mathcal{S}$ outputs whatever P_3 outputs.

Claim 4 *For the simulator $\mathcal{S}$ corrupting party P_3 as described above, and interacting with the functionality $\mathcal{F}_{f(psi)}$,*

$$\left\{\text{HYBRID}_{\pi_{cb},\mathcal{A}(z)}\left((X, D^{(1)}), (Y, D^{(2)}), \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{F}_{f(psi)},\mathcal{S}(z)}\left((X, D^{(1)}), (Y, D^{(2)}), \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof:

Hybrid$_0$: Real execution.

Hybrid$_1$: Same as hybrid$_0$, except that the PRP is replaced with random encoding.

Hybrid$_2$: Ideal execution.

It's clear that **Hybrid**$_0$ and **Hybrid**$_1$ are computationally indistinguishable by a reduction to the PRP. We prove that **Hybrid**$_1$ and **Hybrid**$_2$ are also computationally indistinguishable.

First, we consider the partial views up to Step 5 in the protocol π_{cb}. The partial view in the hybrid$_1$ is $\{[V_1], [D^{(1)}], [V_2], [D^{(2)}], [\pi(D^{(1)}||D^{(2)})], Z\}$ and in the ideal world is $\{[\widetilde{V}_1], [\widetilde{D}^{(1)}], [\widetilde{V}_2], [\widetilde{D}^{(2)}], [\pi(\widetilde{D}^{(1)}||\widetilde{D}^{(2)})], \widetilde{Z}\}$. In both worlds, these messages are distributed uniformly at random and independently from one another and from the rest of the joint distributions, thus, the partial views in both worlds are identically distributed.

Next, we consider every possible deviation by the adversary, and show that, in each case, the joint distributions in both worlds remain statistically indistinguishable. If P_3 deviates from the protocol by sending the wrong shares to one of the functionalities ($\mathcal{F}_{\text{CheckZero}}$, $\mathcal{F}_{[x]^A\rightarrow[x]^B}$, the 3PC comparison circuit (or $\mathcal{F}_{\mathbf{unionLB}}$), or the 3PC circuit to compute $f(psi)$), or by sending different values of t to P_1 and P_2 in Step 7, then, in the ideal world the simulator detects this right away and aborts, while in the hybrid world the parties will abort with probability at least $1 - O(1/|F|)$. Thus, the joint distributions in both worlds are statistically close under these deviations.

Beside the deviations mentioned above, P_3 can purposely mess up the shuffling process in Step 6 by providing a bad permutation to $\mathcal{F}_{\mathsf{shuffle}}$. If this is the case, in the ideal world, the simulator detects this immediately, but still

hands P_3 random elements to simulate the shares he receives from $\mathcal{F}_{\mathsf{shuffle}}$: $[\widetilde{Z}^{(1)}]$, $[\widetilde{Z}^{(2)}]$. In the hybrid world, P_3 receives $[Z^{(1)}], [Z^{(2)}]$. The executions now continue in Step 7. At this point, we have two different cases.

Case 1: P_3 sends correct t to P_1 and P_2, however, the permutation is wrong.

- If $\mathsf{abort}_1 = 1$: P_3 purposely messes up Z_1. In the ideal execution, $\mathcal{S}$ aborts and outputs the joint distribution $\{[\widetilde{V}_1], [\widetilde{D}^{(1)}], [\widetilde{V}_2], [\widetilde{D}^{(2)}], [\pi(\widetilde{D}^{(1)}||\widetilde{D}^{(2)})], \widetilde{Z}, \widetilde{R}, \widetilde{b}_1 = \mathsf{false}, \perp\}$. We claim that in the hybrid world, $\mathcal{F}_{\text{CheckZero}}$ will outputs $b_1 = \mathsf{false}$ with probability at least $(1 - O(1/|\mathbf{F}|))$, and the joint distribution in the hybrid world is $\{[V_1], [D^{(1)}], [V_2], [D^{(2)}], [\pi(D^{(1)}||D^{(2)})], Z, R, b_1 = \mathsf{false}, \perp\}$, which is identically distributed as that in the ideal world. In Step 8, $Pr[b_1 = \mathsf{true}|\mathsf{abort}_1 = 1] = Pr[\mathsf{sum} = 0] + Pr[\mathsf{sum} \neq 0 \wedge \mathcal{F}_{\text{CheckZero}}(\mathsf{sum}) = \mathsf{true}]$, where $\mathsf{sum} = \sum_{i=1}^{t} r_i(z_{2i} - z_{2i-1})$. At least one of the terms $(z_{2i} - z_{2i-1})$ is non-zero (otherwise, the simulator would set $\mathsf{abort}_1 = 0$), thus $\mathsf{sum} = 0$ if and only if $r_i(z_{2i} - z_{2i-1}) = -\sum_{j \neq i} r_j(z_{2j} - z_{2j-1})$. This happens with probability $1/|F|$, thus $Pr[\mathsf{sum} = 0] = 1/|F|$. In case $\mathsf{sum} = 0$, the probability that $\mathcal{F}_{\text{CheckZero}}(\mathsf{sum})$ outputs true is $1/|F|$. It follows that $Pr[b_1 = \mathsf{true}|\mathsf{abort}_1 = 1] = \frac{1}{|F|} + (1 - \frac{1}{|F|})\frac{1}{|F|} < \frac{2}{|F|}$, and the joint distributions in both worlds are statistically close.
- If $\mathsf{abort}_1 = 0, \mathsf{abort}_2 = 1, \mathsf{abort}_3 = 1$: The only extra thing that P_3 can do here is to submit the wrong shares to one of the functionalities in Step 8 and Step 9, which causes abort to happen in the ideal world with the probability of 1, and in the hybrid world with the probability of at least $1 - O(1/|F|)$. In this case, the joint distributions in both worlds are statistically close. If P_3 submits the correct shares, then in the ideal world, the simulator returns $\widetilde{b}_2 = \mathsf{false}$ ($\widetilde{b}_3 = \mathsf{false}$ in the hybrid branch) in Step 9 and aborts, while in the hybrid world, the parties output $b_2 = \mathsf{false}$ ($b_3 = \mathsf{false}$ in the hybrid branch) with the probability of at least $1 - O(1/|F|)$ and abort. In all cases, the joint distributions in both worlds are statistically close.

We note that if P_3 cheats in Step 6, no matter what P_3 does in the next steps, either $\mathsf{abort}_1 = 1$ or ($\mathsf{abort}_2 = 1/\mathsf{abort}_3 = 1$) must be true.

Case 2: P_3 sends $t \neq |X \cap Y|$ to P_1 and P_2. In this case, the analysis is very similar to case 1. If $t < |X' \cap Y|$, $\mathsf{abort}_2 = 1, \mathsf{abort}_3 = 1$, else if $t > |X' \cap Y|$, $\mathsf{abort}_1 = 1$. With similar arguments in case 1, the claim that the joint distributions in both worlds are statistically close holds.

If P_3 does not deviate from the protocol, it is easy to see that the joint distributions in both worlds are identically distributed.

In conclusion, the joint distributions in both worlds are computationally indistinguishable. ■

4.3 Three Party Oblivious Shuffling

$\mathcal{F}_{\mathsf{Shuffle}}$

Inputs: P_1, P_2, P_3 submit a set of replicated arithmetic shares $[X]^A = \{[x_1]^A, ..., [x_{2n}]^A\}$. P_3 submits a permutation π.

Functionality:

- If there are any inconsistencies among the input shares, output abort to every party.
- Shuffle the shares using permutation π.
- Re-randomize the replicated shares.

Outputs: $[\pi(X)]^A$

Figure 9: Ideal Functionality for Shuffling

We construct a low-bandwidth protocol for permuting replicated arithmetic shares in the three party setting, where P_3 chooses the permutation π that remains hidden from the two other players. Our protocol runs in linear time and has communication cost of $5n$ field elements when permuting n values. The formal description of the protocol appears in Figure 10. Informally, the parties begin by sampling replicated shares of a random field element, α, and compute replicated sharing $[\alpha x]^A$ through a call to $\mathcal{F}_{\text{mult}}$. P_3 and P_2 begin by converting the replicated sharing into a 2-out-of-2 sharing, and then apply a random permutation σ_1 to the resulting shares. P_2 sends P_1 his shares (after re-randomizing), and P_3 sends to P_1 $\pi \circ \sigma_1^{-1}$. They both permute their shares, and the three parties convert the two-out-of-two sharings back into to replicated shares. Because neither P_1 nor P_2 sees both permutations, neither learns anything about the composed permutation π. However, going from replicated shares to two-out-of-two additive shares in order to hide the permutation allows the adversary to modify the shared values. To prevent this, the parties expose α, and perform several checks to ensure that everyone behaved honestly.

We note one important subtlety about how the correctness of the shares is validated. Letting $X^{(3)}$ denote permuted the output array, and $Y^{(3)}$ denote the authenticate array, it does not suffice to simply verify at the end of the computation that $\alpha X^{(3)} = Y^{(3)}$. Without the verification that $\alpha X = Y$, a malicious P_1 could learn some information about the permutation with non-negligible probability: in Step 2, he adds d_i to the i^{th} shares in Y so that $y_i = \alpha x_i + d_i$. In Step 7, he adds $-d_i$ to the j^{th} shares in $Y^{(3)}$ before converting it back to replicated sharing. If the check in Step 10 fails, he learns that $\pi(i) \neq j$. If the check passes (with probability $1/n$), he knows $\pi(i) = j$. By having both checks in Step 9 and 10, P_1 or P_2 can only modify the shares of the data and MACs once, and they will be caught with high probability if they choose to do so. If any of them attempts to modify the data, the checks fail with high probability.

Π_{Shuffle}: Three Party Shuffling

Inputs: P_1, P_2, and P_3 have replicated shares $[X]^A = \{[x_1]^A, ..., [x_n]^A\}$, P_3 has a permutation π. (As all shares are of arithmetic values, we suppress the superscript indicating this going forward.)
Protocol:

1. P_1, P_2, and P_3 call $\mathcal{F}_{\text{rand}}$ to sample a shared random MAC key, $\alpha \in \mathcal{Z}_p^*$. The key is distributed as replicated shares $[\alpha]$.
2. The parties make n calls to $\mathcal{F}_{\text{mult}}$, computing $[Y] = \{[\alpha x] \mid x \in X\}$.
3. P_2 and P_3 locally compute 2-out-of-2 shares of X and Y: $[X] \xrightarrow{2,3} \langle X^{(1)} \rangle$ and $[Y] \xrightarrow{2,3} \langle Y^{(1)} \rangle$. (The superscript denotes a possible change in the shared value by an adversary.)
4. P_2 and P_3 call $\mathcal{F}_{\text{coin}}$ to sample a random permutation σ_1. They permute their shares according to σ_1.
5. P_2 and P_3 call $\mathcal{F}_{\text{coin}}$ and use the resulting randomness for re-randomizing their shares: $\langle X^{(2)} \rangle \leftarrow \mathsf{reRand}(\langle \sigma_1(X^{(1)}) \rangle)$ and $\langle Y^{(2)} \rangle \leftarrow \mathsf{reRand}(\langle \sigma_1(Y^{(1)}) \rangle)$. P_2 sends his shares to P_1.
6. P_3 sends $\sigma_2 = \pi \circ (\sigma_1)^{-1}$ to P_1. They permute their shares according to σ_2.
7. The three parties transform the permuted shares into replicated sharings: $\langle \sigma_2(X^{(2)}) \rangle \xrightarrow{2} [X^{(3)}]$, $\langle \sigma_2(Y^{(2)}) \rangle \xrightarrow{2} [Y^{(3)}]$.
8. The parties securely open α and locally compute $[Z^{(1)}] = [\alpha \cdot X - Y]$ and $[Z^{(2)}] = [\alpha \cdot X^{(3)} - Y^{(3)}]$.
9. The parties make n calls to $\mathcal{F}_{\text{coin}}$, receiving $R_1 = \{r_1, ..., r_n\}$. They run $\mathcal{F}_{\text{CheckZero}}([\sum r_i z_i])$ where $z_i \in Z^{(1)}$ and abort if the output is False.
10. The parties make n calls to $\mathcal{F}_{\text{coin}}$, receiving $R_2 = \{s_1, ..., s_n\}$. They run $\mathcal{F}_{\text{CheckZero}}([\sum s_i z_i])$ where $z_i \in Z^{(2)}$ and abort if the output is False.

Output: P_1, P_2, and P_3 output replicated sharing $[X^{(3)}]$.

Figure 10: A protocol for securely permuting the replicated sharing of an array.

Theorem 3 *The protocol Π_{Shuffle} for shuffling the shares obliviously (Figure 10) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{Shuffle}}$ (Figure 9) with abort, under a single malicious corruption.*

We first describe a simulator for the case where P_1 is corrupt, and argue that the protocol remains secure under the corruption of P_1. The simulation and argument are almost identical for P_2. The simulation of P_3 appears below that of P_1.

0. The simulator, $\mathcal{S}$, receives input shares of X from the distinguisher, and places them on the tape of P_1.
1. $[\widetilde{\alpha}]^A$: $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{rand}}$, sending P_1 random field elements as his shares of α.
2. $[\widetilde{Y}]^A$: $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{mult}}$. It receives the corrupted party's shares of X and α, and the adversary's specified output shares $[y_i]_1$. If any of the shares of X differ from the ones placed by $\mathcal{S}$ on P_1's input tape, or if the shares of α are inconsistent with the simulated values from Step 1, $\mathcal{S}$ sets $\mathsf{abort}_1 = 1$. Regardless, $\mathcal{S}$ sends $\mathcal{A}$ a random field element for every share of Y that he expects to receive from P_2: $[\widetilde{y_i}]_2$.
3. $\langle \widetilde{X}^{(2)} \rangle, \langle \widetilde{Y}^{(2)} \rangle$: $\mathcal{S}$ simulates the message that P_1 receives from P_2 in Step 5 by sending random field elements.
4. $\widetilde{\sigma}_2$: $\mathcal{S}$ sends P_1 a random permutation σ_2 on behalf of P_3. He permutes his local state with σ_2 to mirror the expected behavior of P_1.

5. $([\widetilde{X}^{(3)}], [\widetilde{Y}^{(3)}])$: $\mathcal{S}$ receives P_1's shares of $\langle X^{(2)}\rangle, \langle Y^{(2)}\rangle$ as input to the two (independent) calls to the share conversion functionality. If there is any discrepancy with the simulated values from Step 3, $\mathcal{S}$ set $\mathsf{abort}_2 = 1$. Regardless, he queries the ideal functionality to receive $[\pi(X)]$, and uses these to simulate the output of the first call to the share conversion functionality. He sends random field elements to simulate the output of the second call to the share conversion functionality: $[Y^{(3)}]$.

6. $\widetilde{\alpha}$: $\mathcal{S}$ sends random elements to simulate the opening of α.

7. $\widetilde{R}_1, \widetilde{b}_1$: $\mathcal{S}$ sends n random elements to simulate the calls to $\mathcal{F}_{\text{coin}}$. If $\mathsf{abort}_1 = 1$, or if P_1 sends the wrong shares to $\mathcal{F}_{\text{CheckZero}}$, $\mathcal{S}$ submits abort to the ideal functionality and simulates the output of $\mathcal{F}_{\text{CheckZero}}(Z^{(1)})$ by sending $\widetilde{b}_1 = 0$ to $\mathcal{A}$.

8. $\widetilde{R}_2, \widetilde{b}_2$: $\mathcal{S}$ sends n random elements to simulate the calls to $\mathcal{F}_{\text{coin}}$. If $\mathsf{abort}_2 = 1$, or if P_1 sends the wrong shares to $\mathcal{F}_{\text{CheckZero}}$, $\mathcal{S}$ submits abort to the ideal functionality and simulates the output of $\mathcal{F}_{\text{CheckZero}}(Z^{(2)})$ by sending $\widetilde{b}_2 = 0$ to $\mathcal{A}$.

Claim 5 *For the simulator $\mathcal{S}$ corrupting party P_1 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{Shuffle}}$,* $\left\{\text{HYBRID}_{\pi_{\mathsf{Shuffle}},\mathcal{A}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \stackrel{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{Shuffle}},\mathcal{S}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$

Proof: The joint distribution of the view of P_1 and the output, in the real and ideal executions, respectively, are:

$\left\{\text{HYBRID}_{\pi_{\mathsf{Shuffle}},\mathcal{A}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \left\{[\alpha], [Y], \langle X^{(2)}\rangle, \langle Y^{(2)}\rangle, \sigma_2, [X^{(3)}], [Y^{(3)}], \alpha, R_1, b_1, R_2, b_2, o_1, o_2, o_3\right\}$

$\left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{Shuffle}},\mathcal{A}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \left\{[\widetilde{\alpha}], [\widetilde{Y}], \widetilde{\langle X^{(2)}\rangle}, \widetilde{\langle Y^{(2)}\rangle}, \widetilde{\sigma_2}, \widetilde{[X^{(3)}]}, \widetilde{[Y^{(3)}]}, \widetilde{\alpha}, \widetilde{R}_1, \widetilde{b}_1, \widetilde{R}_2, \widetilde{b}_2, \widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3\right\}$

We consider a partial view of the protocol, through Step 8, just before the two calls to $\mathcal{F}_{\text{CheckZero}}$: $\{[\alpha], [Y], \langle X^{(2)}\rangle, \langle Y^{(2)}\rangle, \sigma_2, [X^{(3)}], [Y^{(3)}], \alpha\}$ and $\{[\widetilde{\alpha}], [\widetilde{Y}], \langle \widetilde{X}^{(2)}\rangle, \langle \widetilde{Y}^{(2)}\rangle, \widetilde{\sigma}_2, [\widetilde{X}^{(3)}], [\widetilde{Y}^{(3)}], \widetilde{\alpha}\}$ in the hybrid world and ideal world, respectively. The reader can verify by inspection that every such partial view has identical probability weight in both the hybrid and ideal worlds (whether P_1 cheats or not). Each of the messages in the views are distributed uniformly at random and independent from one another.

Case 0: If P_1 executes the protocol honestly, it is clear that $R_1, R_2, \widetilde{R}_1, \widetilde{R}_2$ are all distributed uniformly at random and $b_i = \widetilde{b}_i = \mathsf{true}$. Thus, the joint distributions in both worlds are identical.

Case 1: If P_1 cheats in Step 2 or Step 7, we argue that the joint distributions in the hybrid and in the ideal worlds are statistically close. First, in the ideal world, any deviation of P_1 will be detected immediately by the simulator. The simulator will output the partial view $\{[\widetilde{\alpha}], [\widetilde{Y}], \langle \widetilde{X}^{(2)}\rangle, \langle \widetilde{Y}^{(2)}\rangle, \widetilde{\sigma}_2, [\widetilde{X}^{(3)}], [\widetilde{Y}^{(3)}], \widetilde{\alpha}, \widetilde{R}_1, \widetilde{b}_1 = \mathsf{false}\}$ if P_1 deviates from the protocol in Step 2, and he'll output $\{[\widetilde{\alpha}], [\widetilde{Y}], \langle \widetilde{X}^{(2)}\rangle, \langle \widetilde{Y}^{(2)}\rangle, \widetilde{\sigma}_2, [\widetilde{X}^{(3)}], [\widetilde{Y}^{(3)}], \widetilde{\alpha}, \widetilde{R}_1, \widetilde{b}_1 = \mathsf{true}, \widetilde{R}_2, \widetilde{b}_2 = \mathsf{false}\}$ if P_1 deviates in Step 7. In both cases, he tells the ideal functionality to abort. However, in the hybrid world, P_1 has a chance to cause $\mathcal{F}_{\text{CheckZero}}$ to accept and the hybrid world execution results in the view $\{[\alpha], [Y], \langle X^{(2)}\rangle, \langle Y^{(2)}\rangle, \sigma_2, [X^{(3)}], [Y^{(3)}], \alpha, R_1, b_1 = \mathsf{true}, R_2, b_2 = \mathsf{true}\}$ which is distinguishable from that in the ideal world. We claim that this occurs with probability at most $\frac{3}{|F|}$.

We first consider the case where $\mathsf{abort}_1 = 1$, which happens when P_1 cheats in Step 2. Let $D = \{d_1, ..., d_n\}$ be the vector of additive terms added to the MAC values by P_1. That is, after Step 2 (in the protocol), $[Y] = \{[\alpha x_i + d_i] | x_i \in X\}$ and $[Z^{(1)}] = \alpha[X] - [Y] = \{[-d_i] | i = 1...n\}$. Note that at least one of the d_i is non-zero, which is why the simulator aborts. Assume $d_j \neq 0$, then $\sum_{i=1}^n -r_i d_i = 0$ if and only if $r_j d_j = -\sum_{i\neq j} r_i d_i$. When $r_j \neq 0$, the equation is satisfied with probability $\frac{1}{|F|}$ as $r_j d_j$ is uniformly distributed in F. When $r_j = 0$, the sum is zero if the adversary only cheats on the j^{th} shares. Thus, $Pr[\sum_{i=1}^n r_i d_i = 0] \leq \frac{1}{|F|} + (1 - \frac{1}{|F|})\frac{1}{|F|} < \frac{2}{|F|}$. In the real world execution, the check passes with probability 1 if $\sum_{i=1}^n r_i d_i = 0$ and with probability $\frac{1}{|F|}$ otherwise. Thus, the probability that the views in two worlds are different is at most $\frac{2}{|F|} + (1 - \frac{2}{|F|})\frac{1}{|F|} < \frac{3}{|F|}$.

Next we consider the case that $\mathsf{abort}_1 = 0$ and $\mathsf{abort}_2 = 1$. In Step 7, P_1 can introduce an additive attack to alter the shares of the data and the MAC. Let the additive terms be $E = \{e_1, ..., e_n\}$ and $F = \{f_1, ..., f_n\}$, such that $[X^{(2)}] = \{[x_i + e_i] | i = 1...n\}$ and $[Y^{(2)}] = \{[\alpha x_i + f_i] | i = 1...n\}$. In the real world execution, the check passes with probability 1 if $\sum_{i=1}^n s_i(\alpha e_i - f_i) = 0$, and it passes with probability $\frac{1}{|F|}$ if $\sum_{i=1}^n s_i(\alpha e_i - f_i) \neq 0$. Assume that $e_j \neq 0$ or $f_j \neq 0$. Then $\sum_{i=1}^n s_i(\alpha e_i - f_i) = 0$ if and only if $s_j(\alpha e_j - f_j) = \sum_{i\neq j} s_i(f_i - \alpha e_i)$. If $\alpha e_j - f_j \neq 0$, then the above equation is satisfied with probability $\frac{1}{|F|}$, as $s_j(\alpha e_j - f_j)$ has uniform distribution over F. Furthermore, $\Pr[\alpha e_j - f_j = 0] = \frac{1}{|F|}$, since α was unknown at the time that e_j and f_j were chosen. In total, the chance that $\sum_{i=1}^n s_i(\alpha e_i - f_i) = 0$ is at most $\frac{1}{|F|} + (1 - \frac{1}{|F|})\frac{1}{|F|} < \frac{2}{|F|}$. Thus, the chance that the check passes in the real world execution is at most $\frac{2}{|F|} + (1 - \frac{2}{|F|})\frac{1}{|F|} < \frac{3}{|F|}$.

In both cases, the view in the real world is different from that in the hybrid world with the probability of at most $\frac{3}{|F|}$. It follows that the distribution on views in the two worlds are statistically close.

Case 2: If P_1 is honest up to Step 8, however, it cheats by sending the wrong shares to any of the $\mathcal{F}_{\text{CheckZero}}$, abort happens in both worlds and the joint distribution in the hybrid world is identical to that in the ideal world.

In conclusion, the joint distributions in both worlds are statistically close. To complete the proof of the Theorem, we note that the honest output is independent of the view of P_1. ■

Now we describe a simulator for a corrupted P_3.

1. $[\widetilde{\alpha}]^A$: $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{rand}}$ by receiving the corrupted party's shares of $\widetilde{\alpha}$.

2. $[\widetilde{Y}]^A$: $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{mult}}$. It receives the corrupted party's shares of X and $\widetilde{\alpha}$, and the adversary's specified output shares $[y_i]_3$. If any of the shares of X differ from the ones placed by $\mathcal{S}$ on P_3's input tape, or if the shares of $\widetilde{\alpha}$ are inconsistent with the values specified by P_3 in Step 1, $\mathcal{S}$ sets $\mathsf{abort}_1 = 1$. Regardless, $\mathcal{S}$ sends $\mathcal{A}$ a random field element for every share of Y that he expects to receive from P_1: $[\widetilde{y_i}]_1$.

3. $\widetilde{\sigma}_1, \widetilde{k}_1$: $\mathcal{S}$ chooses a random permutation $\widetilde{\sigma}_1$ and a random key $\widetilde{k}_1$, simulating the output of $\mathcal{F}_{\text{coin}}$. $\widetilde{k}_1$ is the randomness that P_2 and P_3 use to re-randomize the shares. $\mathcal{S}$ uses $\widetilde{\sigma}_1$ and $\widetilde{k}_1$ to compute $\langle X^{(2)} \rangle$ and $\langle Y^{(2)} \rangle$, mirroring the adversary's action.

4. $[\widetilde{X}^{(3)}]$, $[\widetilde{Y}^{(3)}]$: $\mathcal{S}$ receives the permutation σ_2 that P_3 sends to P_1 and extracts P_3's input $\pi' = \sigma_2 \circ \sigma_1$. $\mathcal{S}$ uses σ_2 compute $\langle \sigma_2(X^{(2)}) \rangle$ and $\langle \sigma_2(Y^{(2)}) \rangle$. $\mathcal{S}$ observes the messages that P_3 sends to $\mathcal{F}_{\langle x \rangle \to [x]}$ in Step 7. $\mathcal{S}$ compares those messages with the one he computes locally. If there is any mismatch, $\mathcal{S}$ sets $\mathsf{abort}_2 = 1$. Regardless, $\mathcal{S}$ submits $([X], \pi')$ to the ideal functionality and receives $[\pi'(X)]$. $\mathcal{S}$ hands P_3 $[\pi'(X)]$ as $[\widetilde{X}^{(3)}]$, and hands him random elements as shares of $[\widetilde{Y}^{(3)}]$.

5. $\widetilde{\alpha}$: $\mathcal{S}$ sends random element to simulate the opening of α.

6. $\widetilde{R}_1, \widetilde{b}_1$: $\mathcal{S}$ sends n random field elements to simulate the calls to $\mathcal{F}_{\text{coin}}$. If $\mathsf{abort}_1 == 1$, or if P_3 sends the wrong shares to $\mathcal{F}_{\text{CheckZero}}$, $\mathcal{S}$ submits abort to the ideal functionality and simulates the output of $\mathcal{F}_{\text{CheckZero}}(Z^{(1)})$ by sending $\widetilde{b}_1 = 0$ to $\mathcal{A}$.

7. $\widetilde{R}_2, \widetilde{b}_2$: $\mathcal{S}$ sends n random field elements to simulate the calls to $\mathcal{F}_{\text{coin}}$. If $\mathsf{abort}_2 == 1$, or if P_3 sends the wrong shares to $\mathcal{F}_{\text{CheckZero}}$, $\mathcal{S}$ submits abort to the ideal functionality and simulates the output of $\mathcal{F}_{\text{CheckZero}}(Z^{(2)})$ by sending $\widetilde{b}_2 = 0$ to $\mathcal{A}$.

Claim 6 *For the simulator $\mathcal{S}$ corrupting party P_3 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{Shuffle}}$,*
$\left\{\text{HYBRID}_{\pi_{\mathsf{Shuffle}}, \mathcal{A}(z)}(X, Y, \kappa)\right\}_{z \in \{0,1\}^*, \kappa \in \mathbb{N}} \overset{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{S}(z)}(X, Y, \kappa)\right\}_{z \in \{0,1\}^*, \kappa \in \mathbb{N}}$

Proof:
Case 0: If P_3 follows the protocol honestly, the joint distribution of the view of P_3 and the output, in the real and ideal executions, respectively, are:

$\left\{\text{HYBRID}_{\pi_{\mathsf{Shuffle}}, \mathcal{A}(z)}(X, Y, \kappa)\right\}_{z \in \{0,1\}^*, \kappa \in \mathbb{N}} = \left\{[\alpha], [Y], \sigma_1, [X^{(3)}], [Y^{(3)}], \alpha, R_1, b_1 = \mathsf{true}, R_2, b_2 = \mathsf{true}, out\right\}$

$\left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{A}(z)}(X, Y, \kappa)\right\}_{z \in \{0,1\}^*, \kappa \in \mathbb{N}} = \left\{[\widetilde{\alpha}], [\widetilde{Y}], \widetilde{\sigma_1}, [\widetilde{X}^{(3)}], [\widetilde{Y}^{(3)}], \widetilde{\alpha}, \widetilde{R}_1, \widetilde{b}_1 = \mathsf{true}, \widetilde{R}_2, \widetilde{b}_2 = \mathsf{true}, \widetilde{out}\right\}$

where $out = (o_1, o_2, o_3)$ and $\widetilde{out} = (\widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3)$.

As all the shares, $\alpha, \widetilde{\alpha}$, $R_i, \widetilde{R}_i$ are uniformly distributed, and $([X^{(3)}], out)$ and $([\widetilde{X}^{(3)}, \widetilde{out})$ are identical, the joint distributions in both worlds are identically distributed.

Case 1: If P_3 cheats in Step 2 or Step 7, causing $\mathsf{abort}_1 = 1$ or $\mathsf{abort}_2 = 1$ in the ideal world, the simulator outputs the view $\{[\widetilde{\alpha}], [\widetilde{Y}], \widetilde{\sigma_1}, [\widetilde{X}^{(3)}], [\widetilde{Y}^{(3)}], \widetilde{\alpha}, \widetilde{R}_1, \widetilde{b}_1 = \mathsf{false}\}$ if P_3 deviates in Step 2, and he outputs the view $\{[\widetilde{\alpha}], [\widetilde{Y}], \widetilde{\sigma_1}, [\widetilde{X}^{(3)}], [\widetilde{Y}^{(3)}], \widetilde{\alpha}, \widetilde{R}_1, \widetilde{b}_1 = \mathsf{true}, \widetilde{R}_2, \widetilde{b}_2 = \mathsf{false}\}$ if P_3 deviates in Step 7. In the hybrid world, P_3 has a chance to cause the execution to result in a view that is distinguishable from that in the ideal world, if he can cause $\mathcal{F}_{\text{CheckZero}}$ to output 1. Following the similar analysis in the previous proof, the chance that this happens is at most $O(1/|F|)$. Thus, the joint distributions in both worlds are statistically close.

Case 2: If P_3 is honest up to Step 8, the partial views up to this step in both worlds are identically distributed. If P_3 cheats in either Step 9 or Step 10 by sending the incorrect shares to $\mathcal{F}_{\text{CheckZero}}$, both the hybrid execution and the ideal execution abort. Thus, the joint distributions are also identically distributed. ■

5 Computing on intersecting indices

The approach in Section 4.2 fails when the indices are needed to compute the function f. An attack is possible due to the fact the $\mathcal{F}(k,\cdot)$ values are computed locally and are not bound to the input x. For example, a malicious P_1 can supply an incorrect triple $([x]^A, [d]^A, \mathcal{F}(k,x'))$, causing trouble if x' is in the intersection and x is not, or vice versa. We propose two different protocols to perform f(PSI) on indices. In both, we make use of a *shared oblivious PRF* (soPRF). This is a pseudorandom function that allows for distributed evaluation: the parties, holding secret shares of the PRF key, and secret shares of the input value, can compute the output of the PRF.

In our first construction, in Figure 11, we use a 3-party soPRF (3soPRF), but we do so in a non-black box way. That is, the code of the 3soPRF is embedded inside a larger circuit, and the whole circuit is evaluated securely in an MPC. In our experiments, we instantiated the 3soPRF using both AES, and the Naor-Reingold PRF [33] and found the construction from AES to be more efficent.

In our second protocol (Figure 12), we make black-box use of a 2-party shared oblivious PRF (2soPRF). We instantiated this using the 2soPRF designed by Gordon et al. [19], which is a shared variant of the Naor-Reingold PRF. This construction also failed to beat the non-black box use of AES, but we are hopeful that a more efficient 2soPRF might be found to replace this one, making the construction in Figure 12 more efficient than the one in Figure 11. It is worth noting that highly efficient primitives related to oblivious PRFs have recently been constructed from OT-extension and used in other PSI protocols [29], so something similar here is quite plausible, though still unknown.

The difference between protocols that compute a function on intersection payloads and on intersection indices is how the encoded indices are computed. In Figure 11, this is done by a secure evaluation of a 3soPRF, however, in Figure 12, it is handled differently. First, P_1 and P_2 commit their inputs (indices and payloads) by secret-sharing them as replicated shares. For each index $x \in X$, P_1 computes $\mathcal{F}(k,x)$, secret-sharing it, opening it to P_3. For the same input, P_2 and P_3 compute 2soPRF by using the shared key and shared data. If all parties follow the protocol honestly, P_3 will receive the same values. If P_1 cheats by sending $\mathcal{F}(k,x')$ or P_2 cheats by providing the wrong share for the 2soPRF, P_3 will catch them with high probability. If P_3 is malicious, the only thing he can do is to claim that he receives different encodings, causing all parties to abort. The same thing happens when the three parties compute the encoded indices for P_2's inputs. This procedure enforces the binding between $[x]$ and $[\mathcal{F}(k,x)]$ in Figure 12 . The simulation and security proof for $f(PSI)$ on intersection indices are very similar to that of protocols in Figure 8, thus we do not list them here.

The protocols in Figures 11 and 12 have linear computational complexity and linear communication cost. They are asymptotically better than the merge-compare-shuffle approach [23] that requires $O(n \log n)$ runtime and $O(n \log n)$ bandwidth. In concrete numbers, the AES circuit has 6800 AND gates, thus $2 \times 6800 \times N$ AND gates in total, while the merge-compare-shuffle circuit has at least $4\sigma N \log(2N)$ AND gates, where σ is the total length of the index and the payload in bits, and N is the number of inputs. For $N = 2^{20}$, the AES circuit will have less AND gates if $\sigma \geq 162$. When compared with the protocol for computing only on payloads in Section 4.2, computing on indices is about 5-10X slower in a LAN, and 11-35X slower in a WAN. We also note that our circuit has a constant number of rounds, while the merge-compare-shuffle has $O(\sigma \log n)$ rounds.

Size hiding PSI cardinality: The protocols in Figure 11 can be modified slightly to give a PSI cardinality protocol that gives output only to P_1 and P_2. The intuition is: the PRF key is unknown to anyone, thus after computing the 3soPRF and shuffling the output shares, the PRF values can be safely revealed to P_1 and P_2, instead of P_3. However, this requires some extra steps to ensure that P_1 and P_2 provide valid inputs, i.e. all items in their set are unique. Otherwise, a malicious party can include item x in his set twice: if x is not in the other party's set, the protocol will compute $f(PSI \cup \{x\})$. Otherwise, the honest party will see 3 copies of $\mathcal{F}(k,x)$ in the encoded set and will abort. In both cases, the adversary learns whether x is in the intersection.

To prevent this, P_1 and P_2 sort their input locally and secret-share them as replicated arithmetic shares. They run a 3PC share conversion circuit to convert the shares to replicated Boolean shares. Now for each set of input, the parties run 3PC comparison circuit to verify that they are both in increasing order. After the verification, the parties execute the remainder of the protocol, using the same arithmetic shares to ensure input consistency.

It is tempting to try and compute $f(PSI)$ while hiding the size of the intersection from P_3 in a similar manner. However, if we wish to involve P_3 in the 3-party computation of $f(Z_1)$, we need to reveal the intersection size when we choose the circuit representing f. Without leaking the size of the intersection, P_1 and P_2 can evaluate multiple circuits with P_3, each with a different size, but, depending on f, this might require them to execute n circuits. One interesting direction to explore in future work is the possibility of leaking a noisy intersection size to P_3, preserving differential privacy.

3PC Circuit-Based PSI computations via 3soPRF/AES

Inputs: P_1 has $X = \{x_1, ..., x_n\}$ and payload values $D^{(1)} = \{d_1^{(1)}, \ldots, d_n^{(1)}\}$, P_2 has $Y = \{y_1, ..., y_n\}$ and $D^{(2)} = \{d_1^{(2)}, \ldots, d_n^{(2)}\}$, P_3 does not have inputs.

Protocol:

1. P_1, P_2, and P_3 call $\mathcal{F}_{rand}$ to sample a shared key $[k]^B$
2. P_1 and P_2 calls $\mathcal{F}_{input}$ to share their input and payloads as replicated arithmetic shares $([X]^A, [D^{(1)}]^A)$ and $([Y]^A, [D^{(2)}]^A)$ respectively among the three parties.
3. The parties concatenate the shares $[Z]^A \leftarrow [X]^A || [Y]^A$ and $[W]^A \leftarrow [D^{(1)}]^A || [D^{(2)}]^A$.
4. P_3 samples a random permutation π. The three parties call $\mathcal{F}_{\mathsf{Shuffle}}$ to shuffle the replicated arithmetic shares according to π. Let $Z^{(1)} \equiv \pi(Z)$ and $W^{(1)} \equiv \pi(W)$.
5. They call $\mathcal{F}_{[x]^A \to [x]^B}$ to convert $[Z^{(1)}]^A$ to $[Z^{(1)}]^B$.
6. They execute the 3PC 3soPRF/AES circuit on shared key $[k]^B$ and shared data $[z]^B$ for $z \in Z^{(1)}$ and obtain $[T]^B \equiv [3soPRF(k, z)]^B$ or $[T]^B \equiv [AES(k, z)]^B$ for $z \in Z^{(1)}$.
7. The parties open T to P_3.
8. P_3 uses π^{-1} to shuffle the set T. If P_3 receives duplicated values from either P_1 or P_2, he aborts.
9. They then reveal T to P_1 and P_2. For each pair of duplicated $t \in T$, the parties use the corresponding replicated sharings of index and payload as input to a circuit for f.

Output:
P_1 and P_2 output the result of f and the intersection size s. P_3 outputs the size of the intersection s.

Figure 11: The protocol to compute a function of both indices in the intersection and the payloads associating with the indices. AES can be replaced by a 3PC soPRF.

6 Experiments and Results

We implemented all protocols in C/C++ with the use of NTL and EMP library and tested them with AWS instances (r4.8xlarge). For the LAN configuration, all the instances are in Northern Virginia region. For the WAN configuration, we used the instances in Northern Virginia, Oregon, and North California. In all protocols, the field has to be large enough so that the PRP encodings have negligible collision probability. Let $2^{-\lambda}$ be the desired probability for collision to happen, and let n be the input size. Then the field size needs to be at least $\lambda + 2\log 2n - 1 = \lambda + 2\log n + 1$ bits.

We focus primarily on f(PSI) and PSI cardinality protocols in the three-party setting with honest majority. Kamara et al. [27] computes PSI in this setting, but there is no related work that computes f(PSI) or PSI cardinality in the 3-party setting. In order to have a meaningful comparison, we implemented the generic merge-compare-shuffle protocol using one of the most efficient three-party protocols with honest majority [1], and compared our f(PSI) results against this implementation. For the PSI cardinality (PSI-CA), we compare our results against the merge-compare-add version. For generic three party protocols, Araki et al. [1] achieves the best communication cost with 7 bits per AND gate per party, however, the number of rounds depends on the depth of the circuit. We note here that the merge-compare-add may be slightly faster if implemented with ABY3 [31]: instead of doing the addition by a Boolean circuit (with $O(n)$ AND gates and depth $O(\log^2 n)$), using the ABY3 framework, we could convert the binary shares into arithmetic shares after the comparison phase, and then perform addition on the arithmetic shares for free. However, the dominant cost for the merge-compare-add circuit is the merge step, which requires $2\sigma n \log(2n)$ AND gates and has the depth of $O(\sigma \log n)$: the speed-up from using ABY3 would be less than 2X. We have not taken the time to implement their protocol, but it might be interesting to do so. For the f(PSI) case, the most efficient way to implement the merge-compare-shuffle is by Boolean circuit, thus, there would be no difference between using the constructions of Araki et al. [1] and Mohassel et al. [31], as the latter use the former when it executes Boolean circuits.

It would also be interesting to compare our results with Mohassel et al. [32], which is based on garbled circuits, has a constant number of rounds, and also assumes 1 malicious party out of 3. However, as the implementation is not available, we instead compare with an implementation of semi-honest, two-party garbled circuits; the three-party version is not as efficient as the semi-honest protocol, so this comparison is conservative. Beside comparing our protocols against the generic protocol in the three party setting, we also provide the comparison against the relevant state-of-the-art two party protocols such as [35].

3PC Circuit-Based PSI computations via 2soPRF

Inputs: P_1 has $X = \{x_1, ..., x_n\}$ and payload values $D^{(1)} = \{d_1^{(1)}, \ldots, d_n^{(1)}\}$, P_2 has $Y = \{y_1, ..., y_n\}$ and $D^{(2)} = \{d_1^{(2)}, \ldots, d_n^{(2)}\}$, P_3 does not have inputs.
Protocol:

1. P_1 and P_2 sample random keys k by calling $\mathcal{F}_{\mathsf{coin}}$. They then distribute the keys as replicated shared keys $[k]^A$ among 3 parties using the same randomness.
2. P_1 and P_2 calls $\mathcal{F}_{\mathsf{input}}$ to share their input and payloads as replicated arithmetic shares $([X]^A, [D^{(1)}]^A)$ and $([Y]^A, [D^{(2)}]^A)$ respectively among the three parties.
3. They call $\mathcal{F}_{[x]^A \to [x]^B}$ to convert $[X]^A, [Y]^A$ to $[X]^B, [Y]^B$.
4. P_2 and P_3 convert $[k]$ to $\langle k \rangle$. For each $x \in X$:
   P_1 computes $\mathcal{F}(k, x)$, secret shared as $[\mathcal{F}(k, x)]^A$, and opens the value to P_3.
   P_2 and P_3 convert $[x]^B$ to $\langle x \rangle^B$ locally and compute $\langle 2soPRF(k, x) \rangle^A$. They open the prf to P_3.
5. P_1 and P_3 convert $[k]$ to $\langle k \rangle$. For each $y \in Y$:
   P_2 computes $\mathcal{F}(k, y)$, secret shared as $[\mathcal{F}(k, y)]^A$ and open the value to P_3.
   P_1 and P_3 convert $[y]^B$ to $\langle y \rangle^B$ locally and compute $\langle 2soPRF(k, y) \rangle^A$. They open the prf to P_3.
6. P_3 verifies that for the same input z, the prf values are the same. If P_3 sees duplicated prf values from P_1 or P_2, he aborts.
7. P_3 fixes a permutation π that moves items in the intersection ($Z_1 = X \cap Y$) to the top, placing each item next to its duplicate, and that moves the rest of (Z_2) to the bottom, in sorted order. The three parties call $\mathcal{F}_{\mathsf{Shuffle}}$ to shuffle the shares according to π. The payloads are shuffled along with their indices.
8. P_3 sends the size of the intersection, t, to P_1 and P_2. P_1 and P_2 verify that they receive the same value. Otherwise, they abort.
9. For Z_1, the parties verify that there are t duplicate pairs, using secure arithmetic comparisons.
   - They make t calls to $\mathcal{F}_{\text{coin}}$ to receive $R = \{r_1, ..., r_t\}$.
   - They run $\mathcal{F}_{\text{CheckZero}}([\sum_{i=1}^{t} r_i(z_{2i} - z_{2i-1})])$, where $z_i \in Z^{(1)}$, and abort if the output is False.
10. **Circuit-based protocol:**
    For Z_2, the parties convert $[\mathcal{F}(k, y)]^A$ to $[\mathcal{F}(k, y)]^B$.They then run a sequence of 3PC comparison circuits to verify that the items are in sorted order.
    Hybrid protocol:
    For Z_2, the parties run the union lower bound proof from Section 3 (Figure 4).

    If the proof fails to verify, abort.
11. For each pair of duplicates in Z_1, the parties use one of the corresponding replicated sharings as input to a circuit for f.

Output:
P_1 and P_2 output the result of f and the intersection size s. P_3 outputs the size of the intersection s.

Figure 12: The protocol to compute a function of both indices in the intersection and the payloads associating with the indices.

Table 1: Runtime in seconds in LAN/WAN setting and communication cost in megabytes. The best results for f(PSI) are in blue, those for PSI cardinality are in red, and those for both are in green. In our protocols, we consider the data with variable bitlength. When bitlength (σ) is not specified, the protocols are independent of the bitlength.

n	2^8	2^{12}	2^{16}	2^{20}	2^8	2^{12}	2^{16}	2^{20}	2^8	2^{12}	2^{16}	2^{20}
PSI-CA	**LAN**				**WAN**				**Total Comm (MB)**			
Cristefaro et al [8]	0.89	14.4	230	3677	-	-	-	-	0.16	2.5	40	640
Davidson et al [11]	11.8	176	2837	-	-	-	-	-	2.83	45.3	724	-
Circuit + 2D cuckoo hashing [35]												
Iterative Separate PSI-CA ($\sigma = 32$)	-	2.43	11.3	122	-	11.2	57.5	548	-	72.3	826.1	9971
Iterative Combine PSI-CA ($\sigma = 32$)	-	2.22	9.08	86.6	-	10.1	45.3	390	-	52.7	639	6951
Circuit-based PSI-CA [34]												
No-Stash PSI-CA	-	1.20	8.49	121	-	5.91	22.1	262	-	9	149	2540
3PC Merge-Compare-Add [1]												
$\sigma = 32$	0.63	1.13	4.45	35.6	38.5	49.9	74.6	234	0.46	9.97	201	3891
$\sigma = 64$	1.33	2.15	6.43	70.2	72.6	92.8	140	487	0.93	19.9	402	7782
$\sigma = 80$	1.65	2.74	7.86	91.2	86.5	116	162	602	1.16	24.9	503	9726
(Our) Polynomial PSI-CA												
$\|PSI\| \approx 0$	0.04	0.41	6.9	138	0.4	0.8	8.0	143	0.03	0.4	6.3	100
$\|PSI\| = 0.25$n	0.04	0.45	7.9	162	0.4	0.8	8.9	166	0.03	0.4	6.3	100
$\|PSI\| = 0.50$n	0.04	0.47	8.4	173	0.4	0.9	9.4	176	0.03	0.4	6.3	100
$\|PSI\| = 0.75$n	0.04	0.49	8.8	182	0.4	0.9	9.9	186	0.03	0.4	6.3	100
$\|PSI\| \approx$ n	0.04	0.50	8.0	162	0.4	0.9	9.4	165	0.03	0.4	6.3	100
PSI-CA and **f(PSI)**												
(Our) Circuit PSI-CA/f(PSI)												
$\|PSI\| \approx 0$	0.20	0.37	2.6	19.4	27.5	28.3	34.7	110	0.48	7.66	123	1962
$\|PSI\| = 0.25$n	0.21	0.34	2.0	15.8	26.9	27.1	33.3	93.4	0.38	6.028	96	1540
$\|PSI\| = 0.50$n	0.20	0.30	1.6	11.9	26.6	27.3	30.7	76.8	0.27	4.37	70	1119
$\|PSI\| = 0.75$n	0.22	0.26	1.3	8.00	26.4	27.9	29.8	61.5	0.17	2.72	44	698
$\|PSI\| \approx$ n	0.20	0.24	0.8	3.61	25.9	26.7	29.4	43.5	0.07	1.08	17	276
(Our) Hybrid PSI-CA/f(PSI)												
$\|PSI\| \approx 0$	0.04	0.37	6.52	130	1.00	1.7	10.2	146	0.08	1.31	21	336
$\|PSI\| = 0.25$n	0.03	0.34	5.84	117	1.02	1.7	9.5	133	0.08	1.29	20.7	331
$\|PSI\| = 0.50$n	0.03	0.32	5.58	111	1.02	1.65	9.3	127	0.08	1.27	20.4	326
$\|PSI\| = 0.75$n	0.03	0.29	4.88	98.9	1.02	1.68	8.5	114	0.08	1.25	20.1	321
$\|PSI\| \approx$ n	0.02	0.26	3.50	67.9	1.02	1.63	7.4	91.8	0.07	1.08	17	276
(Our) f(PSI) on indices	0.58	2.51	23	290	41	50	151	2974	8.90	142	2270	36327
f(PSI)												
3PC Merge-Compare-Shuffle [1]												
$\sigma = 32$	0.32	0.77	4.77	65.7	39	50	82	404	0.9	19	391	7596
$\sigma = 64$	0.73	1.35	11.4	132	77	100	144	872	1.8	38	782	15196
$\sigma = 80$	0.88	1.63	13.7	159	91	124	180	1303	2.2	48	975	18989
2PC Merge-Compare-Shuffle [23]												
$\sigma = 32$	1.04	22.5	86.0	-	1.28	24.3	286	-	7.5	166	14040	-

Table 2: Time (in seconds) taken for shuffling the indices and for the whole circuit PSI-CA/f(PSI). The time taken to shuffle data depends only on the input length. We show results for the case $|PSI| = 0.5n$. Runtime of circuit PSI-CA/f(PSI) for different intersection size can be found in Table 1.

n		2^8	2^{12}	2^{16}	2^{20}
LAN	Shuffle	0.004	0.022	0.21	2.34
	Circuit f(PSI)	0.20	0.30	1.6	11.9
WAN	Shuffle	0.32	0.44	1.00	8.61
	Circuit f(PSI)	26.6	27.3	30.7	76.8

Table 3: Experiments with payload in LAN setting: runtime in seconds, length of the indices $\sigma = 80$, number of items $n = 65536$, $|PSI| = 0.5n$

payload length	0	80	160	240	320	400	480	560	640	720	800
3PC MCS [23]	13.7	23.0	34.4	45.7	54.5	67.5	78.0	87.2	99.5	108	121
Our Circuit f(PSI)	1.60	1.78	1.98	2.21	2.37	2.56	2.74	2.92	3.14	3.31	3.56

Table 4: Experiments with payload: communication cost in megabytes, length of the indices $\sigma = 80$, number of items $n = 65536$, $|PSI| = 0.5n$

payload length	0	80	160	240	320	400	480	560	640	720	800
3PC MCS [23]	975	1950	2925	3900	4875	5850	6825	7800	8775	9750	10725
Our Circuit f(PSI)	70	83	97	110	124	137	151	164	178	191	205

6.1 Computing functions of the intersection

6.1.1 Computing on the payloads

Both of our protocols that compute f(PSI) on payloads (circuit/hybrid f(PSI)) are strictly better than the generic protocols in every setting that we consider: varying input length, network configuration (LAN, WAN), and measuring runtime or communication cost. For the case of empty payload, the results are shown in Table 1. Consider the case that the indices have length of 80 bits (or they have variable length). For input sets of size 2^{20} items, circuit f(psi) is 8X-44X faster than the generic three-party merge-compare-shuffle in LAN, about 12X-30X in WAN, and it uses 9X-68X less bandwidth. For hybrid f(PSI), it is 9X-14X faster in WAN, and uses 56X-68X less bandwidth. The two party garbled circuit implementation of the generic protocol is strictly worse than the three party protocol, thus, we also outperform [32].

When there is an attached payload, the gap between our protocols and the generic ones is even larger. The only cost added to our f(PSI) protocols is the cost to shuffle the payloads together with the indices during the oblivious shuffling step. For our protocols, the shuffling takes very little time compared with the circuit execution step (as shown in Table 2), thus the extra overhead is relatively cheap. However, this is not the case for the generic merge-compare-shuffle circuit. The size and depth of the circuit increases super-linearly with respect to the total length of the indices and the payloads. If the payload length is equal to the indices' length, the runtime and communication cost will increase more than twice. In Table 3 and Table 4, we show how payloads of different length affect the performance of our circuit f(PSI) and that of the generic merge-compare-shuffle protocol in LAN. Without the payload, our protocol is just 8X faster and uses 14X less bandwidth. However, when the payload's length is 800 bits, ours is 34X faster and uses 52X less bandwidth.

6.1.2 Computing on the indices

As discussed in Section 5, our f(PSI) on indices does not perform very well against the generic protocol when the payload is small. For example, when input sets are of size 65536, there is no payload, and indices are 80 bits, our protocol is about 2X slower in LAN. This is due to the fact that the players have to securely evaluate the PRP on their indices instead of each computing them locally. However, when the payload is larger than 162 bits, our protocol is the faster one. For the input sets of size 65536, indices of length 80, and the payload length of 800, it takes our protocol 25 seconds to finish, which is 5X faster than the generic protocol. At the same time, it uses 4X less bandwidth.

Table 5: Runtime in seconds in LAN/WAN setting and communication cost in megabytes. In [38], the communication cost does not include the cost to perform base OT. When bitlength (σ) is not specified, the protocols are independent of the bitlength.

n	2^8	2^{12}	2^{16}	2^{20}	2^8	2^{12}	2^{16}	2^{20}	2^8	2^{12}	2^{16}	2^{20}
PSI	**LAN**				**WAN**				**Total Comm (MB)**			
Kamara et al [27]												
$\|PSI\| \approx 0$	**0.005**	**0.05**	**0.6**	**8.74**	**0.20**	**0.68**	**1.77**	**15.3**	**0.07**	**1.0**	**14**	**192**
$\|PSI\| = 0.25$n	**0.004**	**0.05**	**0.6**	**8.36**	**0.20**	**0.75**	**1.85**	**14.7**	**0.08**	**1.2**	**16**	**216**
$\|PSI\| = 0.50$n	**0.005**	**0.06**	**0.6**	**8.40**	**0.26**	**0.81**	**1.84**	**15.8**	**0.09**	**1.3**	**18**	**240**
$\|PSI\| = 0.75$n	**0.005**	**0.06**	**0.6**	8.37	**0.26**	**0.81**	**1.90**	**16.4**	**0.11**	**1.5**	**20**	**264**
$\|PSI\| \approx$ n	**0.005**	**0.07**	0.6	8.30	**0.26**	**0.81**	**2.01**	**16.5**	0.12	1.6	22	**288**
Rindal et al [38]*												
EC-ROM	0.13	0.19	0.94	12.6	0.67	1.5	16	255	0.29	4.8	79	1322
DE-ROM	0.13	0.23	1.3	18	0.9	1.2	6.3	106	0.25	3.5	61	1092
SM, $\sigma = 32$	0.15	0.48	3.5	56	1.3	8	78	1322	2.3	40	451	7708
SM, $\sigma = 64$	0.19	0.84	8.0	134	1.9	16.8	226	3782	5.3	92	1317	22183
Our (Circuit PSI)												
$\|PSI\| \approx 0$	0.21	0.37	2.2	19.4	26.4	27.5	36.3	113	0.48	7.66	123	1962
$\|PSI\| = 0.25$n	0.22	0.34	1.6	15.7	27.6	28.4	33.9	94.8	0.38	6.04	97	1546
$\|PSI\| = 0.50$n	0.21	0.31	1.3	11.8	27.5	27.0	32.1	83.1	0.28	4.41	70.6	1129
$\|PSI\| = 0.75$n	0.20	0.27	0.9	**7.97**	28.3	27.7	31.5	63.9	0.17	2.78	44.5	713
$\|PSI\| \approx$ n	0.20	0.23	**0.5**	**3.63**	26.1	26.7	29.1	44.4	**0.07**	**1.16**	**18.5**	296

6.2 PSI Cardinality

We compare the performance of our protocols with the generic merge-compare-add protocol implemented by [1]. Beside that, we also compare them with other customized PSI cardinality protocols in two party setting such as [8], [11], [35]. Cristofaro et al. [8] did not provide experiment results in their paper, however, we found the experiment results in the LAN setting for their protocol in [13]. The execution time and communication cost of PSI cardinality protocols are shown in Table 1.

In terms of communication cost, polynomial PSI-CA requires the least bandwidth while hybrid PSI-CA and circuit PSI-CA needs 3X and 3X-20X more bandwidth respectively depending on the size of the intersection. The generic merge-compare-add protocol needs 97X more bandwidth (for the case $\sigma = 80$, while [8] and [11] requires 6.4X and at least 100X more communication respectively).

In the LAN setting, hybrid and polynomial PSI-CA is faster when the input length is small (2^8), while circuit-base PSI-CA is faster when the input size is large ($2^8, 2^{16}, 2^{20}$). When the input size is small, round complexity plays a more important role in the total runtime, even in the LAN setting. Polynomial and hybrid PSI-CA have only a few rounds while circuit PSI-CA has a few hundred rounds. When $\sigma = 80$ and input length is 2^{20}, our circuit PSI-CA is 4.7X-25X faster than the generic merge-compare-add protocol, 190X-1018X faster than [8], and 6X-33X faster than [34]. For [35], we only have the results for $\sigma = 32$ bits, however, it is still slower than our circuit PSI-CA 4.4X-24X.

In the WAN setting, the round complexity is an important factor. For circuit PSI-CA, the network delay contributes to around 25 seconds in the total runtime. With the least bandwidth required, polynomial PSI-CA perform the best for input size $n = 2^8, 2^{12}, 2^{16}$. When the input size is large $n = 2^{20}$, circuit PSI-CA is faster. When compared with the generic protocol for input length of 2^{20} and $\sigma = 80$, circuit PSI-CA is 5.5X-14X faster, and requires 5X-35X less bandwidth.

6.3 When f(PSI) computes the PSI

Our protocols are designed to focus on f(PSI) and PSI cardinality, however, as PSI is an important application, we ran experiments for f(PSI) for the specific case that f(PSI) is PSI and compare our results against other state-of-the-art PSI protocols in two/three-party setting such as [38] and [27] to complete the picture. We can obtain PSI with any of our PSI-CA/f(PSI) protocols. The only extra thing that needs to be done is to have the third party to send the intersection to the other two parties, and they verify that they receive the same set and that set is a subset of theirs. The results in Table 5 shows that our PSI results are quite competitive, especially when the intersection

size is large. When the input length is large $(2^{16}, 2^{20})$, our PSI is similar to [38] in Random Oracle model, and 7X-37X faster in standard model (for $\sigma = 64$). In some cases, we are even faster than [27] (input length 2^{20} and the intersection size is at least 3/4 of the input length).

Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-15-C-4070.

References

[1] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862. IEEE Computer Society Press, May 2017.

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 805–817. ACM Press, October 2016.

[3] A. Borodin and R. Moenck. Fast modular transforms. *J. Comput. Syst. Sci.*, 8(3):366–386, June 1974.

[4] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM CCS 18*, pages 1223–1237. ACM Press, 2018.

[5] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018.

[6] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. Efficient three-party computation from cut-and-choose. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 513–530. Springer, Heidelberg, August 2014.

[7] 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.

[8] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *CANS 12*, volume 7712 of *LNCS*, pages 218–231. Springer, Heidelberg, December 2012.

[9] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. Cryptology ePrint Archive, Report 2010/469, 2010. http://eprint.iacr.org/2010/469.

[10] Bernardo Machado David, Ryo Nishimaki, Samuel Ranellucci, and Alain Tapp. Generalizing efficient multiparty computation. In Anja Lehmann and Stefan Wolf, editors, *ICITS 15*, volume 9063 of *LNCS*, pages 15–32. Springer, Heidelberg, May 2015.

[11] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In Josef Pieprzyk and Suriadi Suriadi, editors, *ACISP 17, Part II*, volume 10343 of *LNCS*, pages 261–278. Springer, Heidelberg, July 2017.

[12] 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 13*, pages 789–800. ACM Press, November 2013.

[13] Changyu Dong and Grigorios Loukides. Approximating private set union/intersection cardinality with logarithmic complexity. Cryptology ePrint Archive, Report 2018/495, 2018. https://eprint.iacr.org/2018/495.

[14] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 360–385. Springer, Heidelberg, November / December 2015.

[15] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 1–19. Springer, Heidelberg, May 2004.

[16] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.

[17] Juan A. Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC 2007*, volume 4450 of *LNCS*, pages 330–342. Springer, Heidelberg, April 2007.

[18] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge University Press, 2009.

[19] Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure computation with sublinear amortized work. Cryptology ePrint Archive, Report 2011/482, 2011. http://eprint.iacr.org/2011/482.

[20] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 513–524. ACM Press, October 2012.

[21] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. LNCS, pages 59–85. Springer, Heidelberg, December 2018.

[22] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 175–203. Springer, Heidelberg, March 2017.

[23] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[24] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738, 2017. https://eprint.iacr.org/2017/738.

[25] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738, 2017. http://eprint.iacr.org/2017/738.

[26] Stanislaw Jarecki and Boyang Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 360–378. Springer, Heidelberg, July 2018.

[27] Seny Kamara, Payman Mohassel, Mariana Raykova, and Seyed Saeed Sadeghian. Scaling private set intersection to billion-element sets. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 195–215. Springer, Heidelberg, March 2014.

[28] 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.

[29] 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 16*, pages 818–829. ACM Press, October 2016.

[30] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS 09*, volume 5888 of *LNCS*, pages 1–20. Springer, Heidelberg, December 2009.

[31] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS 18*, pages 35–52. ACM Press, 2018.

[32] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 591–602. ACM Press, October 2015.

[33] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS*, pages 458–467. IEEE Computer Society Press, October 1997.

[34] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based psi with linear communication. In *EUROCRYPT 2019*, volume 11478 of *Advances in Cryptology*, pages 122–153. Springer, 2019.

[35] 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.

[36] Shrisha Rao, Mainak Chatterjee, Prasad Jayanti, C. Siva Ram Murthy, and Sanjoy Kumar Saha, editors. *Distributed Computing and Networking, 9th International Conference, ICDCN 2008, Kolkata, India, January 5-8, 2008*, volume 4904 of *Lecture Notes in Computer Science*. Springer, 2008.

[37] 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.

[38] 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 17*, pages 1229–1242. ACM Press, October / November 2017.

[39] Tamir Tassa. Generalized oblivious transfer by secret sharing. *Des. Codes Cryptogr.*, 58(1):11–21, 2011.

A Assumed Ideal Functionalities

FUNCTIONALITY $\mathcal{F}_{\text{rand}}$ - Generating Random Replicated Arithmetic Shares

Let P_i be the malicious adversary. Upon receiving r_i from P_i, the ideal functionality $\mathcal{F}_{\text{rand}}$ chooses a random $r \in F$ and samples r_{i+1}, r_{i+2} such that $r = r_i + r_{i+1} + r_{i+2}$. $\mathcal{F}_{\text{rand}}$ gives $\{r_j, r_{j+1}\}$ to P_j.

Figure 13: Generating shares of a random field element

FUNCTIONALITY $\mathcal{F}_{\text{coin}}$ - Generating Random Value

The ideal functionality $\mathcal{F}_{\text{coin}}$ chooses a random $r \in F$ then gives r to all the parties.

Figure 14: Sample a random field element

B Implementation details

B.1 Addition/Subtraction/Comparison Circuits

There are multiple ways to implement the addition/subtraction circuits, there are trade-offs between them in terms of round complexity, runtime, and communication cost. For example, the simple ripple carry adder has O(k) AND gates and O(k) rounds while a parallel prefix adder has $O(k \log k)$ AND gates and $O(\log k)$ rounds. In our implementation, we adopted the linear round addition and subtraction circuits (ripple carry adder and ripple

FUNCTIONALITY $\mathcal{F}_{\text{mult}}$ - Secure Mult. Up To Additive Attack

Input: Parties have replicated arithmetic shares of x, y. Let P_i be the malicious adversary. P_i specifies one of his output shares, z_i.

1. Upon receiving shares from the honest parties, the ideal functionality $\mathcal{F}_{\text{mult}}$ computes x, y and $[x]_i^A = (x_i, x_{i+1}), [y]_i^A = (y_i, y_{i+1})$. Let $[x']_i^A = (x'_i, x'_{i+1}), [y']_i^A = (y'_i, y'_{i+1})$ be the shares submitted by the adversary. The ideal functionality computes $d = (x_i y_i + x_i y_{i+1} + x_{i+1} y_i) - (x'_i y'_i + x'_i y'_{i+1} + x'_{i+1} y'_i)$
2. $\mathcal{F}_{\text{mult}}$ hands $[x]_i^A$, $[y]_i^A$, and d to the adversary/simulator $\mathcal{S}$.
3. The functionality $\mathcal{F}_{\text{mult}}$ computes $z = xy + d$ and samples z_{i+1} uniformly at random, then it defines $z_{i+2} = z - (z_i + z_{i+1})$.
4. The ideal functionality $\mathcal{F}_{\text{mult}}$ hands each party P_j its share $[z]_j^A = \{z_j, z_{j+1}\}$.

Figure 15: Multiplication up to an additive attack

FUNCTIONALITY $\mathcal{F}_{\text{CheckZero}}$ - Checking equality to 0

Input: Parties hold replicated shares $[x]$
Functionality: Upon receiving the shares from the parties, the functionality reconstruct x. Then:

- If $x = 0$, the functionality sends $\mathsf{abort} = 0$ to the parties.
- If $x \neq 0$, with probability $\frac{1}{|F|}$ the functionality sends $\mathsf{abort} = 0$ to the parties, and with probability $1 - \frac{1}{|F|}$ it sends $\mathsf{abort} = 1$ to the parties.

Figure 16: Checking Equality to 0.

FUNCTIONALITY $\mathcal{F}_{\text{input}}$ - Sharing of Inputs

Let P_j be the corrupted party.

1. Functionality $\mathcal{F}_{\text{input}}$ receives inputs $v_1, ..., v_M \in F$ from the parties. For every $i = 1, ..., M$, $\mathcal{F}_{\text{input}}$ also receives from $\mathcal{S}$ the shares v_i^j of the corrupted parties for the i^{th} input.
2. For every $i = 1, ..., M$, $\mathcal{F}_{\text{input}}$ computes all shares $(v_i^j, v_i^{j+1}, v_i^{j+2})$ such that v_i^{j+1} is sampled uniformly at random and $v_i = v_i^j + v_i^{j+1} + v_i^{j+2}$.
   For every $i = 1, ..., n$, $\mathcal{F}_{\text{input}}$ sends P_j its output shares $[v_i]_j^A = \{v_i^j, v_i^{j+1}\}$.

Figure 17: Sharing inputs

borrow subtractor) due to their simplicity compared to parallel prefix adders/subtractors. Due to the high number of rounds, the protocols that use these circuits suffer due to network delay when operating in the WAN setting. When the input size is small, the network delay is the main contributor to the total runtime.

For the comparison circuit $(x > y)$, we adopt the approach used in [30]. Even though there exist $O(\log k)$ and O(1) round circuits [17], we decided to implement a linear round comparison circuit due to its simplicity. The inputs to the circuit are $[x]^B = [x_{k-1}]^B||...||[x_0]^B$ and $[y]^B = [y_{k-1}]^B||...||[y_0]^B$. The circuit is described as the following recursive relation: $t_0 = 0$, $t_{i+1} = t_i \oplus ((t_i \oplus x_i).(t_i \oplus y_i))$. The output of the circuit is $[t_k]^B$. If x ¿ y, $t_k = 1$, otherwise, $t_k = 0$.

B.2 Approximate x mod p circuit

As previously discussed in Section A, when converting $[x]^A$ to $[x]^B$, we need to compute $[x \mod p]^B$ where $x \in [0, (3p-3)]$. This can be done by executing the circuit $x \leftarrow x - (x > p).p$ twice. However, it can be more efficient with a circuit that computes an approximation of x mod p. Let k be the bitlength of p, x can be written as $x = x_{k+1}x_kx_{k-1}...x_0$. As $x \in [0, (3p-3)]$, both x_{k+1} and x_k cannot be 1, otherwise $x > 3p$. Thus, we can have a faster way to compute x mod p: $x = x - (2p.x_{k+1}) \oplus (p.x_k)$. This approach, however, would fail if $x_{k-1}...x_0 \in [p, 2^k - 1]$. This happens with the probability of at most $\frac{(2^k-p)}{2^k}$. To make this probability small, we pick p such that $2^k - p$ is small. For example, k = 80, $p = 2^{80} - 65$, the failure probability is at most 2^{-73}.

B.3 Field implementation based on built-in C/C++ 128-bit integer type

Our protocols operate on a field of size of at least 80 bits. NTL is a natural choice for the implementation, however, NTL is quite slow. In our project, we implemented the field using built-in 128-bit unsigned integer. Our experiments showed that our field multiplication operation is about 15X faster than that of NTL when the field size is less than 85 bits. Our circuit/hybrid PSI and PSI cardinality protocols used this implementation.

Assume that we are working on the prime field Z_p where $p = 2^{2k} - r$, r is chosen to be small and $3k < 127$. For a concrete example, let $k = 40, r = 65$, thus $p = 2^{80} - 65$. Let $x = 2^k x_2 + x_1, y = 2^k y_2 + y_1$. We perform field operations as below:

- Addition(x, y): $z \leftarrow x + y \mod p$
- Subtraction(x, y): $z \leftarrow (p + x - y) \mod p$
- Multiplication(x, y): $z = (2^k x_2 + x_1)(2^k y_2 + y_1) \mod p = 2^{2k} x_2 y_2 + 2^k (x_2 y_1 + x_1 y_2) + x_1 y_1 \mod p = r x_2 y_2 + 2^k (x_2 y_1 + x_1 y_2) + x_1 y_1 \mod p$
- Fast modulo: $x \mod p$
  1. $x \leftarrow r(x \gg 2k) + x \,\&\, (2^{2k} - 1)$
  2. $while(x \geq p)\ x \leftarrow (x - p)$

B.4 Parallelization with multi-threading

We did not use parallelization in our implementation, even though the protocols are highly parallelizable. For polynomial evaluation and interpolation, without multi-threading, the tree has to be built to the top level. However, with multi-threading, we can divide the input into 2^t groups, then evaluate each group with a thread. For the interpolation, the step to build and evaluate the tree can be done the same way. When we reconstruct the polynomial, the parallelization can be applied till the t^{th} layer away from the root of the tree. For circuit execution, gates in the same layer can be partitioned into groups so that each group can be executed independently by a thread.

B.5 Fast polynomial evaluation/interpolation over n arbitrary points

We implemented the $O(n \log^2 n)$ algorithm [3] to evaluate and interpolate a polynomial over n arbitrary points (Algorithm 1, 2, 3). Each point is an element of a prime field F_p where p is large. In our implementation, n is not required to be a power of two, and the field can have arbitrary size. These algorithms were implemented with NTL library.

Algorithm 1 BuildTree

Input: $X = \{x_0, ..., x_{n-1}\}$
Output: Binary tree T

if n == 1 then T.val = $(x - x_0)$
else
    T.left ← **BuildTree**($\{x_0, ..., x_{n/2-1}\}$)
    T.right ← **BuildTree**($\{x_{n/2}, ..., x_{n-1}\}$)
    T.val ← T.left*T.right

Algorithm 2 Evaluation

Input: T ← **BuildTree**$(x_0, ..., x_{n-1})$, $p(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$
Output: $\{p(x_0), ..., p(x_{n-1})\}$

if (deg(T.val) == 1) then $p(x_0)$ ← **rem**(p(x), T.val).
else
    $\{p(x_0), ..., p(x_{n/2-1})\}$ ← **Eval**(T.left, **rem**(p(x), T.val)).
    $\{p(x_{n/2}), ..., p(x_{n-1})\}$ ← **Eval**(T.right, **rem**(p(x), T.val)).

Algorithm 3 Interpolation

Input: T ← **BuildTree**$(x_0, ..., x_{n-1})$, $f(x)$ ← (T.val)', $\{p(x_0), ..., p(x_{n-1})\}$
Output: $p(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$

if (deg(T.val) == 1) then
    y ← **rem**(f(x), T.val)
    T.p ← $(p(x_0)^{-1} * y)$
else
    Interp(T.left, **rem**(f(x), T.val), $\{p(x_0), ..., p(x_{n/2})\}$)
    Interp(T.right, **rem**(f(x), T.val), $\{p(x_{1+n/2}), ..., p(x_{n-1})\}$)
    T.p ← T.left.val*T.right.p + T.right.val*T.left.p
Return T.p