Private Set Intersection with Linear Communication from General Assumptions

Brett Hemenway Falk
University of Pennsylvania

Daniel Noble
University of Pennsylvania

Rafail Ostrovsky
UCLA

March 1, 2018

Abstract

This work presents an improved hashing-based algorithm for Private Set Intersection (PSI) in the honest-but-curious setting. The protocol is generic, modular and provides both asymptotic and concrete efficiency improvements over eisting PSI protocols.

If each player has m elements, our scheme requires only $O(m\lambda)$ communication between the parties, where λ is a security parameter. This is the first protocol that to achieve PSI using only asymptotically linear communication under standard cryptographic assumptions and without Random Oracles. Our protocol also provides 10-15% reduction in communication costs under real-world parameter choices.

Our protocol builds on the hashing-based PSI protocol of Pinkas et al. (USENIX 2014, USENIX 2015), but we replace one of the sub-protocols (handling the cuckoo "stash") with a special-purpose PSI protocol that is optimized for comparing sets of unbalanced size. This brings the asymptotic communication complexity of the overall protocol down from $\omega(\lambda m)$ to $O(m\lambda)$, and provides concrete performance improvements over the most efficient existing PSI protocols.

Our protocol is simple, generic and benefits from the permutation-hashing optimizations of Pinkas et al. (USENIX 2015) and the Batched, Relaxed Oblivious Pseudo Random Functions of Kolesnikov (CCS 2016).

1 Introduction

Private Set Intersection (PSI) is a secure computation protocol that allows two parties, who each hold a private list of elements from some universe $\mathcal{U}$, to compute the intersection of their (private) lists, without revealing information about the elements outside the intersection. PSI is an important cryptographic tool, and is a building block for many more complex functionalities and thus has received a lot of attention from the cryptographic community. In this work, we focus on PSI in the honest-but-curious setting, and thus we focus on comparing our protocol to other protocols that target the honest-but-curious security model.

A simple, generic method for privately computing set intersection would be to securely compute all pair-wise comparisons between the elements. If each player has m elements, this would require m^2 comparisons. The number of comparisons can be reduced by first hashing the elements into bins. The players would agree on a hash function $h : \mathcal{U} \to [n]$, and then hash their elements into bins, where each bins of size b. Then for each bin, the players can perform a secure comparison of all the elements. This basic hashing protocol requires nb^2 secure comparisons. If $n = O(m)$, then with high probability, the maximum bin size is $\log n / \log \log n$, so this would require $n \left({}^{\log n}/_{\log \log n}\right)^2$ secure comparisons. If, instead of performing all pair-wise comparisons in each bin, we recursed, hashing each bin into sub-bins, we obtain a protocol that requires $O(n \log n)$ secure comparisons.

This scheme can be improved by replacing one player's hash table with a cuckoo hash table [36, 34, 28]. In this modification, the players choose two hash functions, and Alice hashes each of her elements into two bins, and Bob uses uses the two hash functions to hash his elements into a cuckoo-hash table with a stash. Then, for each location in Bob's cuckoo hash table, the players compute a secure comparison between the single element in that bucket and every element in Alice's corresponding bucket. Finally, they compare every element in Bob's stash against every element in Alice's table. Since Bob only has a single element in each of his buckets, this method only requires $O(m)$ comparisons to compare Bob's cuckoo table against Alice's table. Unfortunately, comparing Bob's stash (of size s) against Alice's table still requires $O(ms\lambda)$ communication in the protocols of [36, 34, 28]. The entire protocol then requires $O(ms\lambda)$ communication, and somewhat counterintuitively, the communication complexity of the protocol is dominated by computation of the small ($\omega(1)$-sized) stash.

In this work, we make three improvements to this hashing-based PSI protocol

1. We observe that Bob's hashing protocol does not need to support dynamic insertions and deletions, and thus we can replace the cuckoo-hashing scheme with a 1-out-of-k hashing scheme, and compute the optimal allocation of his elements in an offline pre-processing phase. Although the size of the hash table, n, remains $n = O(m)$, this allows us to shrink the constants, and remove some of the heuristics from the failure probability analysis. This is described in Section 4.

2. We show how to use efficient protocols for unbalanced PSI [26, 4] to reduce the communication complexity of the comparing Bob's stash to Alice's set. This provides both asymptotic and practical improvements in communication complexity of existing PSI protocols. Our protocol reduces the communication cost of the stash-comparison step from $\omega(\lambda m)$ to $O(\lambda m)$, thus reducing the communication complexity of the entire protocol from $\omega(\lambda m)$ to $O(\lambda m)$.

   We give details of our construction in Section 5, and concrete performance numbers in Section 9.

3. Finally, we show how to modify the protocol so that the participants learn only a secret-sharing of the intersection, rather than the intersection itself. This is necessary for many secure computations that use PSI as a building block (e.g. secure computation of cross-tabs and secure database joins). Our protocols exhibit better asymptotic communication complexity than existing generic PSI protocols for computing secret sharings of an intersection. We give details of this protocol in Section 8.

Together, these modifications yield an extremely efficient PSI protocol based on general assumptions that achieves linear communication complexity. In addition to the asymptotic improvements in communication complexity outlined above, our protocols also provide concrete improvements in communication complexity for real-world set sizes. In Section 9 we calculate the actual communication cost of our protocol, and identify the set sizes at which our modifications begin to offer concrete improvements in communication cost.

2 Previous work

There have been many approaches to the problem private set intersection (PSI), and early works focused on building custom protocols to perform PSI (some examples include [12, 27, 17, 22, 23, 5, 6, 7]). Over the years, many special-purpose PSI protocols have been proposed and implemented. Many of the early protocols were designed around Oblivious Polynomial Evaluation (OPE). In this framework, Alice interpolates a polynomial with roots at their elements, and then Alice and Bob work together to privately evaluate this polynomial at Bob's private elements. The private evaluation can be done using any additively homomorphic cryptosystem. Some PSI protocols that fall into this framework include [12, 18, 27, 5]. Appendix A in [7] provides a nice overview of many of the special-purpose PSI protocols.

In the face of the plethora of custom PSI protocols, [19] proposed the idea, that *generic* (circuit-based) PSI protocols had many advantages, most notably that they were easy to implement and integrate into other, more complex secure computation protocols.

One natural approach is to use Oblivious Pseudo-Random Functions (OPRFs) [11]. An oblivious PRF is a protocol where Alice holds a PRF key κ, and Bob holds an input x, and the OPRF protocol allows Bob to learn $\mathrm{PRF}(\kappa, x)$ while Alice learns nothing about x.

OPRFs provide a natural method for a linear-communication PSI protocol in the semi-honest model. If Alice has a set X and Bob has a set Y, Alice will generate a key, κ, for an Oblivious PRF, and for every $x \in X$, they will use the OPRF protocol to give Bob $\mathrm{PRF}(\kappa, x)$. Then Alice will locally evaluate $\mathrm{PRF}(\kappa, y)$ for all $y \in Y$, and send these evaluations to Bob. Bob can then locally compute the intersection by comparing his evaluations to those received from Alice.

OPRFs can be implemented generically, using secure Multiparty Computation to compute an ordinary PRF, where Alice's private input is the PRF key, and Bob's private input is his evaluation point. This approach was taken in [35] where they used garbled circuits to compute to obliviously compute the AES-based PRF. There have also been many special-purpose constructions of Oblivious PRFs designed specifically for set intersection protocols. The work of [17] shows how to instantiate an oblivious version of the Naor-Reingold PRF (based on the DDH assumption), and make it secure against malicious adversaries. The works [6, 7] use the one-more RSA assumption in Random Oracle Model to build an OPRF-based linear-time PSI protocol, and the work of [23] uses an OPRF-based PSI protocol to provide security against malicious adversaries based on the one-more gap Diffie-Hellman problem in the Random Oracle Model. The work of [22] builds an OPRF secure in the standard model under the Decisional Composite Residuosity assumption.

In [26] it was observed that the OPRF-based PSI protocols are well-suited to applications where the parties hold sets unequal size. In particular, if Bob's set Y is much smaller than Alice's set X, then using an OPRF-based PSI protocol, they only need $|Y|$ OPRF calls, followed by $|X|$ communication. In particular, they show that the natural approach of using garbled circuits to implement an AES-based PRF is extremely efficient when Bob's set is sufficiently small. The recent work of [4] uses levelled fully homomorphic encryption to create a PSI protocol for unbalanced set sizes.

Unfortunately, when the set sizes are roughly balanced, the generic OPRF protocols that use general-purpose MPC machinery to obliviously evaluate a PRF are not as efficient as the custom-PSI protocols, whereas the custom OPRF-based PSI protocols like [6, 7, 23] achieve linear complexity and practical efficiency, but under strong and non-standard cryptographic assumptions.

The work of [19] identified three natural PSI protocols that could easily be implemented by an off-the-shelf MPC protocol. If the universe $\mathcal{U}$ of elements is known in advance, and is not too large, the players can simply encode their sets as characteristic vectors in $\{0,1\}^{|\mathcal{U}|}$, and then perform $|\mathcal{U}|$ secure bit-wise AND operations to compute their intersection (called the Bit-Wise And (BWA) protocol). When the universe is not known in advance (or is too large), the players can securely perform all pairwise comparisons (this requires m^2 secure comparisons to intersect two sets of size m), this is called the Pair-Wise Comparison (PWC) protocol, and is included only as a baseline, or straw-man protocol. Finally, they introduce the Sort-Compare-Shuffle (SCS) paradigm, where each player locally sorts their sets, then they engage in an secure computation to securely sort their joint set (using the Waksman sorting network). After the joint multi-set is sorted, all elements in the intersection will occur twice in two adjacent positions. Thus the intersection can be computed using $2m-1$ secure comparisons (comparing element i and $i+1$ for $i = 1, \ldots, 2m-1$). Finally, before the intersection can be revealed, it must be randomly shuffled (again using the Waksman sorting network) to hide information carried by the position of the intersected elements. The Waksman sorting network requires $O(m \log m)$ comparisons to sort a m elements, and thus the total number of comparisons required by the SCS approach is $O(m \log m)$.

If the players agree a hash function (or hash functions), they can use the hash functions to locally sort their elements into bins, and then perform pairwise comparisons on the bins. In its simplest form, the players agree on a hash function, $h : \mathcal{U} \to [n]$, and some bucket size b. Then they locally hash their m elements into n buckets of size b. If any bucket receives more than b elements, the protocol will fail, so b must be chosen to be large enough so that this probability is sufficiently small. Then for each bucket, the players will engage in a secure computation to compare all elements within that bucket. If they use brute-force comparison within the bucket, this requires b^2 comparisons, and the entire protocol requires nb^2 comparisons to compute the intersection. If the players use the SCS method (described above) within each bucket, the number of comparisons drops to $O(nb \log b)$. If $n = O(m)$, then b must be $O(\log m / \log \log m)$, and the entire protocol is $O(m \log m)$.

The works of [36, 34] outline an optimization of this approach, where one player uses a traditional hash, while the other uses cuckoo hashing. To do this, Alice and Bob agree on k hash functions $h_1, \ldots, h_k$, and Alice hashes each of her elements into the k buckets defined by these hash functions. Alice's buckets will be sized to store as many elements as necessary. Bob, on the other hand, uses $h_1, \ldots, h_k$ to build a cuckoo hash table (with a stash), and hashes each element into this cuckoo hash table. For each of the n bins, Alice and Bob engage in a secure computation to compare the single element in that bin Bob's cuckoo hash table to the b elements Alice has in her bin. Finally, they compare each element in Bob's stash to every one of Alice's elements. If the stash has size s, this requires $nb + ns$ secure comparisons. Using cuckoo hashing, we can set $n = O(m)$, $s = \omega(1)$, and as above $b = O(\log(m))$, and thus the protocol uses $O(m \log m)$ secure comparisons. The protocols of [36, 34] use an OT-masking protocol to replace the $(b+s)n$ secure comparisons to simply sending $(1+s)n$ pseudo random masks. This reduces the communication complexity of these protocols to $\omega(n\lambda) = \omega(m\lambda)$. The primary improvement introduced in [34] is the notion of *permutation-based hashing* which reduces the complexity of each secure comparison (but not the number of secure comparisons). Permutation-based hashing can be used to improve the performance of all the hashing-based PSI protocols (including ours), and we review the details of permutation-based hashing in Section 3.2. The performance of [34] can be further improved by viewing the OT-based solution as a special OPRF-based solution, and instantiating it with novel, special-purpose OPRFs [28].

Bloom filters provide a natural, generic method for computing set intersections. If each participant inserts their m elements into a Bloom filter of size n, then they can use a secure bitwise-AND calculation to calculate the intersection of their Bloom filters. The players can locally query this "intersected" Bloom filter on each of their elements to find the intersection. This approach was taken in [32]. It is straightforward to check that if an element appears in the intersection, it will also show up in the intersected Bloom filter. It is not too hard to see that the "intersected" Bloom filter created in this way may have extra ones that would not appear in a fresh Bloom filter created by inserting only the elements in the intersection of the two private sets. These extra ones, have the potential to leak information about the underlying sets, and thus this simple method of computing a set intersection using Bloom filters cannot be made to meet the security definitions of PSI.

Nevertheless, Bloom filters can be used to perform PSI. The protocol of [8] introduces the notion of a garbled Bloom filter. In a traditional Bloom filter, an element x is inserted by ORing a 1-bit into the k locations defined by the hash functions $h_1, \ldots, h_k$. In a garbled Bloom filter, each entry holds a string (rather than a single bit), and an element x is inserted by secret-sharing x using a k-out-of-k secret sharing scheme, $(x = s_1 + \cdots + s_k)$ and inserting the shares s_i into the location determined by h_i. If the slot $h_i(x)$ is occupied, we *re-use* the existing share in that location. As long as one of the k slots is unoccupied, there will be enough freedom to make the s_i sum to x. If all k slots are occupied, then the insertion fails (just as in a regular Bloom filter). This approach can also be made secure against malicious adversaries [39].

The garbled Bloom filter can be made into a PSI protocol as follows. Alice will create a standard Bloom filter encoding her set, while Bob will create a garbled Bloom filter encoding his set. Denote these Bloom filters $A \in \{0,1\}^n$ and $B \in \left(\{0,1\}^\lambda\right)^n$. Then for each entry $i \in [n]$, if $A[i] = 0$, then set $C[i] \overset{\$}{\leftarrow} \{0,1\}^\lambda$. If $A[i] = 1$, then $C[i] = B[i]$. It is not hard to check that this is a garbled Bloom filter that encodes all elements in the intersection. The somewhat surprising result from [8] is that this resulting garbled Bloom filter has exactly the same distribution as a garbled Bloom filter created by the intersection, and thus it leaks no information beyond the intersection. Implementing this PSI protocol using MPC requires n secure (single-bit) comparisons. Note that in the semi-honest setting, Alice can generate all the random values (for when $A[i] = 0$) and then Bob can receive the garbled Bloom filter, compute the intersection, and send the intersection to Alice, and thus there is no need to generate the random values within the MPC protocol. For a false-positive rate ϵ, a Bloom filter holding m elements needs to be of size $O(m \log(1/\epsilon))$, thus when $\epsilon = O(m^{-1})$, $n = O(m \log m)$.

3 Preliminaries

3.1 Cryptographic primitives

Our constructions make use of several standard cryptographic primitives, Oblivious Transfer (OT) [37, 10]. Pseudorandom Functions (PRFs) and secure multiparty computation (MPC). These are all standard cryptographic primitives, and we assume the reader has some familiarity with them, and thus we only provide brief reviews. Formal definitions can be found in most cryptographic textbooks, e.g. [14, 15].

A PRF is a keyed function, $F(\cdot,\cdot)$, with the property that if a secret key, κ, is chosen uniformly, the outputs of $F(\kappa,\cdot)$ are indistinguishable from the independent, uniformly chosen random values. An *oblivious* PRF (OPRF) [11] is a two-party secure computation for computing a PRF, where one party holds the key, and the other holds the input value. In particular, an OPRF is a protocol between Alice and Bob, where Alice holds a key, κ, Bob holds an evaluation point x, such that at the end of the protocol, Alice learns nothing, and Bob learns nothing beyond $F(\kappa, x)$. There are many OPRF protocols in the literature, and OPRFs can be implemented by using any generic MPC protocol to compute a PRF. The OPRF functionality is described schematically in Figure 1.

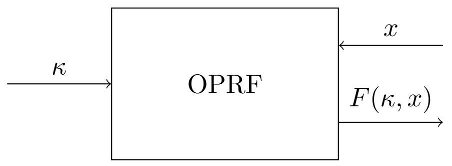

Figure 1: The Oblivious-PRF (OPRF) functionality

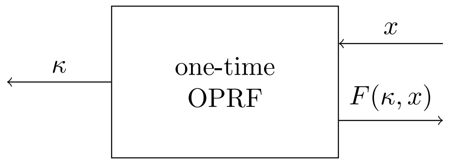

Figure 2: The one-time OPRF functionality

We also consider a weaker notion, a one-time OPRF, which is similar to an OPRF, except that instead of providing a key, a random key is generated by the protocol, and Alice receives the key as an output. Thus a one-time PRF securely realizes the functionality, where Alice provides no input, Bob provides an input x, Alice receives a (uniformly random) key, κ, and Bob receives $F(\kappa, x)$. The qualifier one-time indicates that a new key is generated at each instantiation, so the protocol can only be used to evaluatio $F(\kappa, \cdot)$ once. The one-time OPRF functionality is described schematically in Figure 2.

3.2 Permutation-based hashing

Our constructions also benefit from the idea of permutation-based hashing [3, 34]. Permutation-based hashing can be used with any hashing-based protocol to reduce the size of the elements stored in each hash bucket. In general, it takes $\log |\mathcal{U}|$ bits to store an element $x \in \mathcal{U}$. In a traditional hash table, a hash function, $h : \mathcal{U} \to [n]$ is chosen, and the element x is stored in the location indexed by $h(x)$. Notice, however, that the bucket index, $h(x)$, carries $\log n$ bits of information, so it should be possible to reduce the information stored in the bucket from $\log |\mathcal{U}|$ bits to $\log |\mathcal{U}| - \log n$ bits, while still retaining the ability to uniquely recover an element x. This reduction in storage is possible, if we choose our hash function carefully. Permutation-based hashing provides a method for doing this, using a Feistel-style trick. Suppose an element x has bit-representation $x = x_1 || x_2$, where x_1 has length $\log n$, and x_2 has length $\log \mathcal{U} - \log n$. If $f : \{0,1\}^{\log \mathcal{U} - \log n} \to \{0,1\}^{\log n}$, then we define $h(x) = x_1 \oplus f(x_2)$, and we store x_2 in the bin defined by $h(x)$. The crucial observation here is that if x and y are in the same bin, and the stored values are the same (i.e., $x_2 = y_2$), then that means $f(x_2) = f(y_2)$, and since $h(x) = h(y)$, we conclude that $x_1 = y_1$, which means $x = y$. The permutation-based hashing trick can be used in all hashing-based PSI protocols, including this one. In the situation where there are multiple hash functions, the id of the hash function must also be stored in the bin to allow equality tests [29].

4 Static (offline) hashing

Instead of cuckoo hashing, we propose using multiple-choice hashing [38], where there are k hash functions, and each element is hashed into d buckets (using some choice of d of the hash functions) such that each bucket contains at most one element. A simple observation is that if player 1 uses a d_1-out-of-k hashing scheme, and player 2 uses a d_2-out-of-k scheme with $d_1 + d_2 > k$, then any element in the intersection must appear in at least one overlapping bucket. Then the intersection can be computed efficiently by simply comparing the buckets. The schemes of [36, 34, 28] which use cuckoo hashing, essentially do this (in the online setting) with $d_1 = k$, and $d_2 = 1$. Our schemes will also focus on the case where $d_1 = k$, and $d_2 = 1$.

A simple variant of this idea is when $d_1 = d_2 = 2$, and $k = 3$, *i.e.*, both players use two-out-of-three hashing scheme, which guarantees that every common element will collide in at least one of its three buckets.

Two-out-of-three hashing has been used to build data structures that support efficient set intersection queries in the setting where privacy is not a concern [2, 9]. In [2], two-out-of-three hashing is used to build dynamic data structures (supporting insertions and deletions) that allow efficient set intersection queries, but they do not consider the notion of *private* set intersection. In [9], they consider the notion of two-out-of-three cuckoo hashing, which improves performance over standard two-out-of-three hashing.

Cuckoo hashing is a specific type of multiple-choice hashing procedure that is designed for *dynamically* adding elements to the hash table. In the PSI setting, however, both parties know their sets in advance, and can find the optimal static allocation. In fact, it remains an open question whether the dynamic cuckoo hashing insertion algorithm can obtain the optimum load threshold achievable by a static allocation [33].

We review the bounds for offline hashing in Appendix A.1, and review an efficient algorithm for finding the optimal allocation in Section A.2.

5 Construction

At a high level, our construction is as follows: Alice and Bob choose k hash functions, $h : \{0,1\}^* \to [n]$, Then Alice hashes each of her elements into k buckets, and Bob uses multiple-choice hashing to hash each of his elements into 1 (out of k possible) buckets. Then they perform pairwise comparisons on the buckets. If k is constant (relative to m) we can set n to be $O(m)$ and the hashing will succeed with probability $1 - o(n)$. To make this failure probability negligible, we allow Bob to keep a super-constant sized "stash" of elements that could not be allocated to a single bucket. See Appendix A.1 for a more detailed analysis of the failure probability.

Then, we use secure comparison protocol (like those described in [36, 34, 28] to compare the element in each of Bob's buckets to the elements in Alice's corresponding bucket. Finally, we need to compare Bob's stash (of size $\omega(1)$ to Alice's elements (of size $O(m)$). To do this, we use an unbalanced PSI protocol like those described in [26].

1. Set $k \geq 1$, and $n = O(m)$, and the players choose k hash functions $h_i : \mathcal{U} \to [n]$.
2. Bob will hash his elements into n buckets using a 1-out-of-k hashing scheme, such that each bucket obtains at most one element. Elements that cannot be allocated to a single bucket will be put in a "stash", Stash_B, of size $m' = \omega(1)$. Bob can compute this allocation efficiently as described in Appendix A.2. Let $B[i]$ denote the element stored in Bob's ith bucket. Let $\mathsf{Stash}_B[i]$ for $i \in m'$ denote the ith element of the stash.
3. Alice will hash her elements into n buckets, using a k-out-of-k hashing scheme. Thus with high probability some of Alice's buckets will have $O(\log m / \log\log m)$ elements. Let $C[i]$ denote the number of elements in Alice's ith bucket, and let $A[i][j]$ denote the element in the bucket for $1 \leq i \leq n$, $1 \leq j \leq C[i]$.
4. Alice and Bob will engage in n parallel executions of a one-time-OPRF, where for $i \in [n]$, Alice learns a key κ_i, and Bob learns $S_B^i \overset{\text{def}}{=} \mathrm{PRF}(\kappa_i, B[i])$.
5. For each $i \in [n]$, Alice will locally compute $S_{A,\ell}^i = \mathrm{PRF}(\kappa_i, A[i,\ell])$ for $j = 1, \ldots, C[i]$.
6. Alice will shuffle $\left\{S_{A,\ell}^i\right\}_{i \in [n], \ell \in [C[i]]}$ and send the shuffled set to Bob. Note that this set will have exactly km elements.
7. Bob will locally compute the intersection by finding which of the S_B^i are in the set received from Alice.
8. To handle the stash, Alice will generate a key, κ, for an OPRF, and Alice and Bob will engage in m' executions of an OPRF protocol, where Bob learns $R_B^i \overset{\text{def}}{=} \mathrm{PRF}(\kappa, \mathsf{Stash}_B[i])$ for $i \in m'$.

9. Alice will compute $R_A^i \overset{\text{def}}{=} \mathrm{PRF}(\kappa, A[i])$ for $i \in n$, shuffle the set, and send it Bob who will locally compare these to values to the $\left\{R_B^i\right\}_{i \in m'}$ to find the intersection of Alice's set with the stash.

Figure 3: The high-level outline of our algorithm

In the next sections, we describe alternative methods for implementing the one-time OPRF using methods from [34] and [28], and how to implement the OPRF using methods from [26].

5.1 A OT-masking-based protocol

In this section, we describe how our PSI protocol can implemented using OT. This protocol is essentially a modification of the [36, 34] where we can use OT to create randomized "masks" and then the players can compute equality tests of private elements by comparing their masks in the clear. The main improvements over existing work is the use of static (offline) 1-out-of-k hashing scheme instead of cuckoo hashing, and the replacement of the stash comparison with an efficient unbalanced PSI protocol. The first change removes some of the heuristics from the security analysis, and the second change reduces the communication complexity from super-linear to linear as well as resulting in concrete communication improvements for real-world parameters.

Our protocol also naturally benefits from the permutation-based hashing described in [34] (Section 3.2) and the OPRF-based improvements described in [28] (Section 6).

1. Set $k \geq 1$, and $n = O(m)$, and the players choose k hash functions $h_i : \mathcal{U} \to [n]$.

2. Bob will hash his elements into n buckets using a 1-out-of-k hashing scheme, such that each bucket obtains at most one element. Elements that cannot be allocated to a single bucket will be put in a "stash", Stash_B, of size $m' = \omega(1)$. Bob can compute this allocation efficiently as described in Section A.2. Let $B[i]$ denote the element stored in Bob's ith bucket. Let $\mathsf{Stash}_B[i]$ for $i \in m'$ denote the ith element of the stash.

3. Alice will hash her elements into n buckets, using a k-out-of-k hashing scheme. Thus with high probability some of Alice's buckets will have $O(\log m / \log\log m)$ elements. Let $C[i]$ denote the number of elements in Alice's ith bucket, and let $A[i][j]$ denote the jth element in the bucket for $1 \leq i \leq n$, $1 \leq j \leq C[i]$.

4. For each $i \in [n]$, let $B[i]_1 || \cdots || B[i]_t$ denote the base-N representation of the element $B[i]$. Thus $t = \log_N |\mathcal{U}|$ (or $t = \log_N |\mathcal{U}| - \log_N n + \log_N k$ if we use permutation-hashing).

5. For each $i \in [n]$, the players engage in t parallel 1-out-of-N string OTs (for random strings). To keep the failure probability below $2^{-\sigma}$, the strings should be of length $\sigma + 2\log m$ (as in [36, 34, 28]). For each $i \in [n]$ and $j \in [t]$, Bob provides input $B[i]_j \in [N]$ to the $\binom{N}{1}$-OT, and receives a random string $M^{i,j,B[i]_j}$ while Alice gets N random masks $\{M^{i,j,\gamma}\}$ for $\gamma \in [N]$. Then for each $i \in [n]$ and $\ell \in C[i]$ Alice computes
$$S_A^{i,\ell} = \oplus_{j=1}^t M^{i,j,A[i,\ell]_j}$$
and Bob computes
$$S_B^i = \oplus_{j=1}^t M^{i,j,B[i]_j}$$
At this point $S_A^{i,\ell} = S_B^i$ if $A[i,\ell] = B[i]$, and from Bob's perspective $S_A^{i,\ell}$ is uniformly random otherwise.

6. Alice will shuffle $\left\{S_A^{i,\ell}\right\}_{i \in [n], \ell \in [C[i]]}$ and send the shuffled set to Bob. Note that this set will have exactly km elements.

7. Bob will locally compute the intersection by finding which of the S_B^i are in the set received from Alice.

8. To handle the stash, Alice will generate a key, κ, for an OPRF, and Alice and Bob will engage in m' executions of an OPRF protocol, where Bob learns $R_B^i \overset{\text{def}}{=} \text{PRF}(\kappa, \mathsf{Stash}_B[i])$ for $i \in m'$.

9. Alice will compute $R_A^i \overset{\text{def}}{=} \text{PRF}(\kappa, A[i])$ for $i \in n$, shuffle the set, and send it Bob who will locally compare these to values to the $\left\{R_B^i\right\}_{i \in m'}$ to find the intersection of Alice's set with the stash.

Figure 4: The OT-masking-based algorithm

This protocol requires

- nt $\binom{N}{1}$-string OTs (for random strings). Using OT extension [20], these can be generated using λ base OTs.
- Using permutation-based hashing, $t = \log_N |\mathcal{U}| - \log_N n + \log_N k$.
- $m' = O(\log n)$ secure computations of the PRF
- Alice will send $km + n$ PRF outputs to Bob

6 Relaxed PRFs

In this section, we describe how to use our hashing scheme with the Batched, relaxed-key OPRFs introduced in [28]. At a high level, this protocol is very similar to the OT-based protocol described in Section 5.1. As this is the most efficient instantiation of the one-time OPRF used in our scheme we provide a description in more detail below.

Definition 1 (k-Hamming Correlation Robustness [28])**.** Let $H : \{0,1\}^* \to \{0,1\}^v$ be a hash function such that for all $z_1, \ldots, z_m \in \{0,1\}^*$, and $a_i, b_i \in \{0,1\}^n$ for $i = 1, \ldots, m$, with $\text{wt}(b_i) \geq k$, we have

$$\left\{ \{H(z_i || a_i \oplus [b_i \cdot s])\}_{i \in [m]} \;\middle|\; s \leftarrow \{0,1\}^n \right\}$$
$$\approx$$
$$\left\{ \{r_i\}_{i \in m} \;\middle|\; r_i \overset{\$}{\leftarrow} \{0,1\}^v \right\}$$

This generalizes the original definition of correlation robustness in [20] which required $k = n$, i.e., $b_i \cdot s = s$. A pseudo-random code is a relaxation of error-correcting code such that for any two distinct messages, their encodings have a large minimum distance with high probability over the choice of a specific code from the family.

Definition 2 (Pseudo-Random Code [28])**.** A family of function $\mathcal{C}$ is called a (d, ϵ) pseudorandom code (PRC) if for all strings $x \neq x'$,

$$\Pr_{C \leftarrow \mathcal{C}} [\text{wt}\, C(x) \oplus C(x') < d] \leq 2^{-\epsilon}$$

Definition 3 (m-related key PRFs [28])**.** An m-related key relaxed PRF is a pair of functions $F, \tilde{F}$, and security is defined relative to the following game:

1. The adversary chooses input strings $\{x_j\}_{j \in [n]}$ and pairs $\{(j_i, y_i)\}_{i \in [m]}$, with $y_i \neq x_{j_i}$.
2. The challenger chooses PRF keys $k^*, k_1, \ldots, k_n$ and a challenge bit $b \in \{0,1\}$.
    - If $b = 0$, the challenger sends $\left\{\tilde{F}((k^*, k_j), x_j)\right\}_j$ and $\{F((k^*, k_{j_i}), y_i)\}_i$ to the adversary.

- If $b = 1$, the challenger generates m random strings $z_i \leftarrow \{0,1\}^v$, and sends $\left\{\tilde{F}((k^*, k_j), x_j)\right\}_j$ and $\{z_i\}_i$ to the adversary.

3. The adversary outputs a guess b', and the adversary wins if $b' = b$. We say the adversary's advantage is $\Pr[b' = b] - 1/2$.

We say the pair $F, \tilde{F}$ is secure if the adversary's advantage is negligible. Intuitively, the pair $F, \tilde{F}$ is an m-related key relaxed PRF if $\tilde{F}$ does not reveal information that would allow an adversary to distinguish F from a true random function.

Lemma 1 (Lemma 5 in [28]). If C is a $(d, \epsilon + \log m)$-pseudo random code, and $2^{-\epsilon}$ is negligible and H is a d-Hamming correlation robust hash function, then

$$F\left(((C,s),(q,j)),r\right) = H(j||q \oplus [C(r) \cdot s])$$
$$\tilde{F}\left(((C,s),(q,j)),r\right) = (j, C, q \oplus [C(r) \cdot s])$$

is an m-related key PRF as in Definition 3.

1. Alice chooses a random PRC $C \overset{\$}{\leftarrow} \mathcal{C}$ and sends the code to the receiver.
2. Alice chooses a key $s \overset{\$}{\leftarrow} \{0,1\}^k$.
3. Bob generates two matrices $T_0, T_1 \in \{0,1\}^{m \times k}$ as follows. For $j = 1, \dots, m$,
   (a) The jth row of T_0 is generated uniformly at random $t_{0,j} \overset{\$}{\leftarrow} \{0,1\}^k$.
   (b) The jth row of T_1 is defined as $t_{1,j} = C(r_j) \oplus t_{0,j}$.

   The ith columns of T_0 and T_1 are denoted t_0^i, t_1^i respectively.
4. Alice and Bob engage in k parallel instances of $\binom{2}{1}$-OT for strings of length m as follows:
   - Bob acts as sender and his inputs are the k columns of T_0 and T_1, i.e., Bob's inputs to the OT are $\left\{t_0^i, t_1^i\right\}_{i \in k}$
   - Alice acts as receiver and her inputs are the k bits of s, i.e., $\{s_i\}_{i \in k}$
   - Alice receives k outputs (each of length m) denoted $\left\{q^i\right\}_{i \in k}$

   Alice creates the $m \times k$ matrix Q whose columns are the received vectors $\left\{q^i\right\}$. Thus the ith column of Q is $t_{s_i}^i$. Let q_j denote the jth row of Q. Then

   $$q_j = ((t_{0,j} \oplus t_{1,j}) \cdot s) \oplus t_{0,j} = t_{0,j} \oplus (C(r_j) \cdot s)$$
5. For $j \in [m]$, Alice keeps the PRF seed $((C,s),(j,q_j))$
6. For $j \in [m]$, Bob keeps the relaxed PRF output $(j, C, t_{0,j})$.

Figure 5: Instantiating a BaRK-OPRF using OT [28].

At the end of the protocol in Figure 5, Alice has the keys $k^* = (C,s)$, and $k_j = (j, q_j)$, which allows her to evaluate the BaRK-OPRF

$$F((k*, k_j), r) = F((C,s),(j,q_j)),r) = H(j||q_j \oplus [C(r) \cdot s])$$

for any r, and Bob has the relaxed PRF outputs

$$(j, C, t_{0,j}) = (j, C, q_j \oplus [C(r_j) \cdot s]) = \tilde{F}((k^*, k_j), r_j)$$

which allows him to compute $F((k^*, k_j), r_j)$.

1. Set $k \geq 1$, and $n = O(m)$, and the players choose k hash functions $h_i : \mathcal{U} \to [n]$.
2. Bob will hash his elements into n buckets using a 1-out-of-k hashing scheme, such that each bucket obtains at most one element. Bob can compute this allocation efficiently as described in Appendix A.2. Let $B[i]$ denote the element stored in Bob's ith bucket.
3. Alice will hash her elements into n buckets, using a k-out-of-k hashing scheme. Thus with high probability some of Alice's buckets will have $O(\log m / \log\log m)$ elements. Let $C[i]$ denote the number of elements in Alice's ith bucket, and let $A[i][j]$ denote the element in the bucket for $1 \leq i \leq n$, $1 \leq j \leq C[i]$.
4. Alice and Bob will use the BaRK-OPRF protocol (Figure 5) with Bob providing the contents of his n buckets as his inputs. At the end of the protocol, Alice has keys:

   $$k^*, k_1, \ldots, k_n$$

   and Bob has relaxed PRF outputs $\tilde{F}((k^*, k_i), B[i])$ which allows him to compute $S_B^i \overset{\text{def}}{=} F((k^*, k_i), B[i])$.
5. Alice will compute $F((k^*, k_i), A[i, \ell])$ for $i \in [m]$, $\ell \in C[i]$. Alice will shuffle this set and send the shuffled set to Bob. Note that this set will have exactly km elements.
6. Bob will locally compute the intersection by finding which of the S_B^i are in the set received from Alice.

Figure 6: BaRK-OPRF-based PSI protocol

7 Security

The PSI protocols outlined in Section 5 are formed by taking existing one-time OPRF-based PSI protocols, and combining them (in parallel) with an OPRF-based PSI protocol to handle the unbalanced stash. The proof of security (against honest-but-curious) adversaries then follows immediately from the security of the two underlying protocols.
Full proofs of security for the one-time OPRF protocols can be found in [36, 34, 28], and a proof of security for the unbalanced OPRF-based PSI protocol can be found in [26]. These proofs are straightforward, and we do not repeat them here.

8 Generic PSI protocols for computing sharings

The generic one-time OPRF protocol (Section 5) and its instantiation with BaRKs (Section 6) are very efficient, but they reveal the intersection to the players. In many situations, however, PSI is used as a sub-protocol in a larger secure computation, and the intersection itself should never be revealed (e.g. in secure database-joins). In these situations, the protocols outlined in Section 5 are not appropriate, and a different solution is needed.
In this section, we outline a generic PSI protocol with $O(mt \log\log m)$ communication for t-bit values, by combining a basic hashing scheme with the SCS protocol to compare elements within each bucket. This

provides asymptotic communication improvements over existing schemes, using only generic MPC techniques, and is naturally composable with larger secure computations.

8.1 Improving SCS via hashing

Our scheme is a simple hashing-based scheme, but we use a small number of hash buckets. In particular, instead of setting $n > m$ (the number of buckets is larger than the number of elements being hashed) we set $n = m/b$, for some $b > 1$, and allow each bucket to have $(1+\delta)b$ elements.

1. Set $n = m/b$, and let $h : \mathcal{U} \to [n]$ be a hash function
2. Alice and Bob will each hash all of their elements to n buckets, using the hash function, h. If any bucket contains more than $(1+\delta)b$ elements, Alice and Bob abort the protocol. Since the abort leaks information, we will show that this happens with probability that is negligible (in m).
3. For each bucket, Alice and Bob will engage in a 2-party secure computation, implementing the sort-compare-shuffle (SCS) protocol of [19].

Figure 7: Combining Sort-Compare-Shuffle and hashing to reduce the asymptotic communication cost.

First, we examine the failure probability, *i.e.*, the probability that a bucket contains more than $(1+\delta)b$ elements. Fix a player, and a bucket, and let X_i denote the random variable that is 1 if the player's ith element lands in the chosen bucket, and 0 otherwise. Let $X = \sum_{i=1}^{m} X_i$ denote the number of elements that land in the given bucket. If we model h as a truly random hash function, each X_i is an independent random variable with $\Pr[X_i = 1] = E[X_i] = \frac{1}{n}$. Then, $E[X] = b$, and by a Chernoff Bound,

$$\Pr\left[X > (1+\delta)b\right] \leq e^{-\frac{\delta^2 b}{3}}$$

Taking a union bound over the 2 players and the n buckets, we find that the probability of abort (and hence information leakage) is upper bounded by

$$\frac{me^{-\frac{\delta^2 b}{3}}}{b} < 2^{-\frac{\delta^2 b}{3\log m}}$$

Asymptotically, setting $\delta = 1$, and $b = \log^2 m$, we have that the failure probability is negligible in m. Since the SCS protocol requires $O(tb\log b)$ AND gates to compare each bucket of size $O(b)$, the total number of AND gates is $O(mt\log\log m)$. Using the GMW protocol [16], each AND gate can be implemented using 2 OTs. Using OT-extensions, the entire secure two-party computation can be implemented using computation and communication that is linear in the size of the circuit [21].

Concretely, to keep the overall failure probability below $2^{-\sigma}$, we set

$$\begin{aligned} \frac{me^{-\frac{\delta^2 b}{3}}}{b} &< 2^{-\frac{\delta^2 b}{3\log m}} < 2^{-\sigma} \\ &\Rightarrow \log m - .48\delta^2 b < -\sigma \\ &\Rightarrow \frac{2.1(\log m + \sigma)}{\delta^2} < b \end{aligned}$$

In our situation, each bucket has $\ell = (1+\delta)b$ elements and each element is of length $\tau = t - \log n$ bits, thus running the SCS protocol requires $\frac{\tau\ell}{3}(5\log\ell + 14)$ AND gates ([19] Table 1).

Setting $\sigma = 40$ (as in [34, 28]), we can set the values of b and δ to minimize the number of AND gates needed. Table 1 shows that this protocol requires only a few hundred AND gates per element.

Thus for real-world parameter choices, the entire PSI protocol can be computed by a circuit using approximately $500m$ AND gates, and this circuit can be evaluated with constant overhead (independent of the security paramter) using the generic techniques of [21].

$\log(m)$	t	δ	b	Number of AND gates per element
16	24	0.60	327	515
20	24	0.81	190	392
24	32	0.58	395	530
28	36	0.58	430	537
32	36	0.77	255	411

Table 1: Number of AND gates required per element in the hashing-SCS protocol. m is the total number of elements, t is the bit length of each element, b is the expected bucket size, and $(1+\delta)b$ is the maximum alotted bucket size. Thus the overall computation requires m/b SCS sub-computations, each on sets of size $(1+\delta)b$.

9 Concrete communication benchmarks

In [25] it was shown that under the assumption that the dynamic (online) cuckoo hashing algorithm achieves the optimal load, allocating m elements to $n = O(m)$ buckets with a stash of size s will succeed with probability at least $1 - O(n^{-s})$. The protocols of [36, 34, 28], set $n = 1.2m$. Thus, under the assumption that online cuckoo hashing achieves the optimal offline load, to achieve a negligible probability of failure, the stash size, s, must be $s = \omega(1)$ (e.g. $s = O(\log n)$).
In the context of PSI, a cuckoo-hashing failure reveals information about the players' sets, so the failure probability should be set below the security threshold. In the work of [36, 34], they set security threshold to be a constant 2^{-40} independent of the set sizes. In [34], they empirically tested the failure probability with 2-way cuckoo hashing, and different stash sizes for m up to about 2^{14}. Then they extrapolated these failure probabilities up to larger set sizes, and used these values to choose the stash size (See [34] Figure 2). The empirical values they found were consistent with the asymptotic failure rate of $O(n^{-s})$ found [25], with an implicit constant of about 1. Thus to achieve a failure probability of 2^{-40}, they could set the stash size s to be about $s > 40/\log(n)$. The scheme of [28] uses similar stash parameters even though they are hashing with three hash functions instead of two.
In practice, fixing a concrete security parameter that does not increase with m, means the necessary stash size that *decreases* as the set sizes increases, whereas an asymptotic analysis (which assumes that the failure probability should be negligible in m) requires that the stash size *increase* as the set sizes increase. Thus this choice of a concrete security parameter means that the concrete and asymptotic *performance* metrics diverge as the set sizes increase.
For a hash table of size n, and a stash of size s, the hashing protocol of [28] requires n applications of a one-time OPRF, and the communication of n (truncated) PRF outputs in the main phase, and s one-time OPRF evaluations and the communication of ns PRF outputs in the stash phase.
Replacing the stash computation with an unbalanced PSI protocol based on a standard (reusable) OPRF, reducing the communication in the stash phase to s evaluations of an OPRF followed by sending n PRF outputs.
In practice, because equality testing is done by comparing the pseudorandom masks (PRF outputs), there is no need to transmit the entire mask. Instead, to achieve error probability less than $2^{-\sigma}$, it is sufficient to transmit only about $v = \sigma + 2\log m$ bits of the mask, and in this case, by the birthday bound, the probability of a spurious collision between masks is negligible in m. This is what is done in practice by [34, 28].
Let d denote the number of bits required to for a one-time OPRF application, and $d' > d$ be the number of bits required for an OPRF evaluation. Then our protocol obtains a concrete performance improvement whenever $sd' + n\lambda < sd + ns\lambda$. Ignoring the sd term, we obtain a concrete improvement whenever

$$\frac{sd'}{v(s-1)} < n$$

In [1], they report that a single (GMW-based) AES evaluation using ABY can be computed using only 170 kb of communication. Using their custom, "MPC-friendly" PRF, the garbled circuit can be computed using only 23 kb of communication ([1] Table 6). Similarly, 1024 garbled AES representations were computed

OPRF	**Comm. Cost**	**Break-even point**
AES (Obliv-C)	8 Mb	2^{24}
AES (GMW - ABY)	170 Kb	2^{16}
AES (Yao)	177 Kb	2^{16}
LowMC	23 Kb	2^{12}

Table 2: The minimum valueof n for which the OPRF instantiation has a lower communication cost than the one-time OPRF instantiation in [28] This table shows the communication cost of a single OPRF evaluation, and the minimum number of buckets, for which replacing the OPRF-based comparison of the stash in [28] would be improved by switching to this protocol. The AES Obliv-C benchmarks were obtained via our internal tests. The ABY benchmarks were taken from [1], the Yao benchmarks were taken from [26], and the LowMC benchmarks were taken from [1].

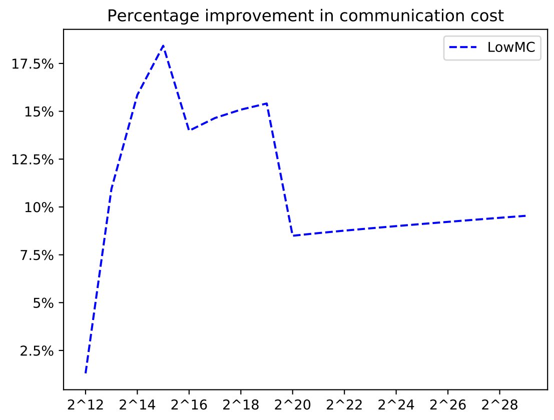

Figure 8: The percentage improvement in overall communication cost when modifying the BaRK protocol to use the LowMC-based PRF to compare the stash. For most values of m, our modification reduces the overall communication cost of the protocol by 10 to 15%. The points of non-differentiability in the graph correspond to the places where the stash sizes drop (we use the same stash sizes as [34, 28])

using 185 Mb of communication ([26] Table 5]), which reduces to about 177 kb per AES circuit (including pregenerating the OTs). Table 2 summarizes the communication costs of these different OPRFs, and the set sizes at which our protocol starts to improve over the [28] protocol.
Thus using an off-the-shelf AES128 implementation, we start to see concrete improvements in communication costs by replacing the one-time OPRF protocol of [28] with a true OPRF, whenever the sets being compared have more than $2^{16} \approx 40,0000$ elements. Using the "MPC-friendly" PRF, LowMC [1], we start to see concrete efficiency improvements for sets of size $2^{12} \approx 6,000$. Note that these are very conservative estimates, since we are assuming the stash size, s, is minimal ($s = 2$), and we are ignoring the cost of s one-time OPRFs.
Figure 8 shows the overall reduction in communication when the BaRK protocol [28] is replaced with our protocol instantiated with LowMC. Figure 9 shows the overall reduction in communication (over [28]) when our protocol is instantiated using different OPRF instantiations. Since the OPRF cost does not increase with m, all three instantiations of our protocols approach the same asymptotic improvement (about 10%) over the protocol of [28].

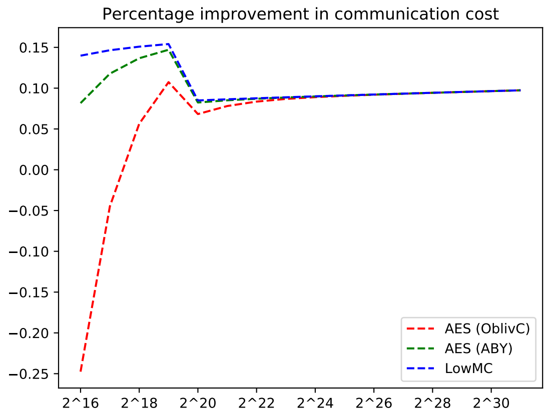

Figure 9: As m increases, and the stash size stays at 2, all three OPRF protocols tend towards a 10% improvement in communication cost over the BaRK protocol of [28]

10 Conclusion

PSI is one of the most basic and fundamental types of secure computation, and numerous diverse types of PSI protocols have been proposed and implemented. Existing PSI protocols are highly optimized and extremely efficient.

In this work, we show how simple, modular composition of existing PSI protocols leads to both asymptotic and concrete improvements in efficiency. Our main result is showing that combining the one-time OPRF protocols of [36, 34, 28] can be improved by replacing the stash-comparison step with an unbalanced PSI protocol [26].

We also show how combining a naive hashing-based PSI protocol with the Sort-Compare-Shuffle (SCS) protocol yields a generic (circuit-based) PSI protocol with better asymptotic efficiency over either solution. This solution is generic, can easily be adapted to support computing only the *secret-sharing* of the intersection set, and provides asymptotic efficiency comparable to the best honest-but-curious PSI protocols.

References

[1] Albrecht, M. R., Rechberger, C., Schneider, T., Tiessen, T., and Zohner, M. Ciphers for mpc and fhe. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2015), Springer, pp. 430–454.

[2] Amossen, R. R., and Pagh, R. A new data layout for set intersection on gpus. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (2011), IEEE, pp. 698–708.

[3] Arbitman, Y., Naor, M., and Segev, G. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on* (2010), IEEE, pp. 787–796.

[4] Chen, H., Laine, K., and Rindal, P. Fast private set intersection from homomorphic encryption. IACR ePrint 2017/299, 2017.

[5] Dachman-Soled, D., Malkin, T., Raykova, M., and Yung, M. Efficient robust private set intersection. In *Applied Cryptography and Network Security* (2009), Springer, pp. 125–142.

[6] De Cristofaro, E., and Tsudik, G. Practical private set intersection protocols with linear complexity. In *Financial Cryptography* (2010), vol. 10, Springer, pp. 143–159.

[7] De Cristofaro, E., and Tsudik, G. Experimenting with fast private set intersection. *Trust 7344* (2012), 55–73.

[8] Dong, C., Chen, L., and Wen, Z. When private set intersection meets big data: an efficient and scalable protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 789–800.

[9] Eppstein, D., Goodrich, M. T., Mitzenmacher, M., and Torres, M. R. 2-3 cuckoo filters for faster triangle listing and set intersection. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (2017), ACM, pp. 247–260.

[10] Even, S., Goldreich, O., and Lempel, A. A randomized protocol for signing contracts. *Communications of the ACM 28*, 6 (1985), 637–647.

[11] Freedman, M. J., Ishai, Y., Pinkas, B., and Reingold, O. Keyword search and oblivious pseudorandom functions. In *TCC* (2005), vol. 3378, Springer, pp. 303–324.

[12] Freedman, M. J., Nissim, K., and Pinkas, B. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques* (2004), Springer, pp. 1–19.

[13] Gao, P., and Wormald, N. C. Load balancing and orientability thresholds for random hypergraphs. In *Proceedings of the forty-second ACM symposium on Theory of computing* (2010), ACM, pp. 97–104.

[14] Goldreich, O. *Foundations of cryptography: volume 1*. Cambridge university press, 2001.

[15] Goldreich, O. *Foundations of cryptography: volume 2*. Cambridge university press, 2004.

[16] Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game. In *STOC* (1987), ACM, pp. 218–229.

[17] Hazay, C., and Lindell, Y. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *Journal of cryptology 23*, 3 (2010), 422–456.

[18] Hazay, C., and Nissim, K. Efficient set operations in the presence of malicious adversaries. In *Public Key Cryptography* (2010), vol. 6056, Springer, pp. 312–331.

[19] Huang, Y., Evans, D., and Katz, J. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS* (2012).

[20] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *Crypto* (2003), vol. 2729, Springer, pp. 145–161.

[21] Ishai, Y., Kushilevitz, E., Ostrovsky, R., and Sahai, A. Cryptography with constant computational overhead. In *Proceedings of the fortieth annual ACM symposium on Theory of computing* (2008), ACM, pp. 433–442.

[22] Jarecki, S., and Liu, X. Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In *TCC* (2009), vol. 5444, Springer, pp. 577–594.

[23] Jarecki, S., and Liu, X. Fast secure computation of set intersection. *Security and Cryptography for Networks* (2010), 418–435.

[24] Karoński, M., and Łuczak, T. The phase transition in a random hypergraph. *Journal of Computational and Applied Mathematics 142*, 1 (2002), 125–135.

[25] Kirsch, A., Mitzenmacher, M., and Wieder, U. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing 39*, 4 (2009), 1543–1561.

[26] Kiss, Á., Liu, J., Schneider, T., Asokan, N., and Pinkas, B. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies (PoPETs) 4* (2017), 97–117.

[27] Kissner, L., and Song, D. Privacy-preserving set operations. In *Crypto* (2005), vol. 3621, Springer, pp. 241–257.

[28] Kolesnikov, V., Kumaresan, R., Rosulek, M., and Trieu, N. Efficient batched oblivious PRF with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 818–829.

[29] Lambæk, M. Breaking and fixing private set intersection protocols. *IACR Cryptology ePrint Archive 2016* (2016), 665.

[30] Lelarge, M. A new approach to the orientation of random hypergraphs. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms* (2012), SIAM, pp. 251–264.

[31] Loh, P.-S., and Pagh, R. Thresholds for extreme orientability. *Algorithmica 69*, 3 (2014), 522–539.

[32] Many, D., Burkhart, M., and Dimitropoulos, X. Fast private set operations with SEPIA. Tech. Rep. 345, ETH Zurich, march 2012.

[33] Mitzenmacher, M. Some open questions related to cuckoo hashing. In *ESA* (2009), Springer, pp. 1–10.

[34] Pinkas, B., Schneider, T., Segev, G., and Zohner, M. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium* (2015), pp. 515–530.

[35] Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. In *Asiacrypt* (2009), vol. 9, Springer, pp. 250–267.

[36] Pinkas, B., Schneider, T., and Zohner, M. Faster private set intersection based on ot extension. In *USENIX Security Symposium* (2014), pp. 797–812.

[37] Rabin, M. O. How to exchange secrets with oblivious transfer. Tech. Rep. TR-81, Harvard University, 1981.

[38] Richa, A. W., Mitzenmacher, M., and Sitaraman, R. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization 9* (2001), 255–304.

[39] Rindal, P., and Rosulek, M. Improved private set intersection against malicious adversaries. In *EUROCRYPT* (2017), Springer, pp. 235–259.

A Bounds for static (offline) multiple-choice hashing

In our protocol Bob uses a multiple-choice hashing scheme to hash his m elements into n buckets. In general, each element will be hashed into k out of d possible locations. In previous work, the [36, 34, 28], Bob uses 1-out-of-k cuckoo hashing with a stash. Cuckoo hashing is designed for dynamic (online) hashing, whereas Bob is assumed to know his entire set at the time he makes the hash allocation.
Since the communication complexity of the resulting protocols depends on n (the number of hash buckets), we seek the minimum value of n that results in an acceptably low failure probability. Error bounds for multiple-choice hashing schemes have been well studied, and most results are obtained by analyzing the orientability of certain hypergraphs. In the following sections, we review some of the pertinant results.

A.1 Orienting hypergraphs

Given k hash functions, $h_i : \mathcal{U}[n]$, the problem of hashing m elements into n bins such that each bin contains at most b elements, and each element is hashed using d functions can be analyzed by translating the problem into one of orienting hypergraphs.
Formally, we have,

Definition 4 (Orientability of hypergraphs)**.** A k-uniform hypergraph $H = (V, E)$ is called (d, b)-orientable if there exists an assignment of each hyperedge $e \in E$ to exactly d of its vertices $v \in e$ such that no vertex is assigned more than b hyperedges.

Translating from our original context, hash-bucket will correspond to a vertex in the hypergraph (thus there are n vertices), and each element will correspond to an hyper-edge (thus there are m hyper-edges). The hyper-edge corresponding to an element $x \in \mathcal{U}$ will contain the k vertices corresponding to $h_1(x), \ldots, h_k(x)$. An (d, b) orientation of this hypergraph then corresponds to choosing a set of d hash buckets (vertices) for each element (hyper-edge) such that each vertex (bucket) is chosen at most b times.
The problem of orienting random hypergraphs is well-studied, and many deep results are known [24, 13, 30, 31].
[13] give asymptotic thresholds for (d, b)-orientability in terms of n and m, but their results only hold when b is "sufficiently large."
[2] show how a similar 2-out-of-3 hash structure can be used to compute set intersections in GPUs. There work focuses on the online setting, and they show the naive insertion algorithm has expected constant running time when $n = O(m)$.
[30] gives asymptotic thresholds for (d, b)-orientability in terms of n and m, for almost all values of d, b. The main result of [30] is the following: for any positive integers, k, d, b there is an explicit (although complicated) $c^* = c_{k,d,b}$ such that if $m > c^* n$ then the probability a random n, m, k hypergraph is (d, b)-orientable tends to 0 as $n \to \infty$, and if $m < c^* n$, this probability tends to 1.

k	d	b	$c_{k,d,b}$
3	2	1	.1666
5	3	1	.2119
7	4	1	.1747
9	5	1	.1453
11	6	1	.1236
13	7	1	.1074
15	8	1	.0948

Table 3: The threshold for orientability in random hypergraphs. If m is the number of elements, and n is the number of buckets, a random hashing scheme that hashes each element into d out of k buckets, where each bucket has size b will succeed with probability approaching one if and only if $m/n < c_{k,d,b}$. In particular, if we hash each element into 3 out of 5 buckets, then (with high probability) there will be no collisions as long as $n > 5m$.

k	d	b	$c_{k,d,b}$
2	1	1	.4999
3	1	1	.9179
4	1	1	.9768
5	1	1	.9924
6	1	1	.9973

Table 4: The threshold for 1-out-of-k orientability in random hypergraphs. If m is the number of elements, and n is the number of buckets, a random hashing scheme that hashes each element into d out of k buckets, where each bucket has size b will succeed with probability approaching one if and only if $m/n < c_{k,d,b}$. In particular, if we hash each element into 1 out of 3 buckets, then (with high probability) there will be no collisions as long as $n > 1.09 \cdot m$.

Theorem 1 (Theorem 1 [30])**.** For integers $k > d \geq 1$, and $b \geq 1$, then if ξ is the unique solution to

$$kd = \xi \frac{E\left[\max(d - \mathrm{Bin}(k, 1 - Q(\xi, b)), 0)\right]}{Q(\xi, b+1)\Pr\left[\mathrm{Bin}(k-1, 1 - Q(\xi, b)) < d\right]}$$

and

$$c_{k,d,b} = \frac{\xi}{k \Pr\left[\mathrm{Bin}(k-1, 1 - Q(\xi, b)) < \ell\right]}$$

where

$$Q(x, y) \stackrel{\text{def}}{=} e^{-x} \sum_{j=y}^{\infty} \frac{x^j}{j!}$$

and $\mathrm{Bin}(n,p)$ is the binomial distribution with $\Pr\left[\mathrm{Bin}(n,p) = k\right] = \binom{n}{k} p^k (1-p)^{n-k}$ then

$$\lim_{n\to\infty} \Pr\left[H_{n,m,k} \text{ is } (d,b)\text{-orientable }\right] = \begin{cases} 0 \text{ if } m > c_{k,d,b} n \\ 1 \text{ if } m < c_{k,d,b} n \end{cases}$$

In the case $d = k-1$ and $b = 1$, we have $c_{k,k-1,1} = \frac{1}{k(k-1)}$.

A.2 Finding an optimal allocation

Given $2k-1$ hash functions, $h_i : \mathcal{U} \to [n]$, and m, we would like to find an allocation such that each of the m values is hashed into k of its possible locations, and each bucket contains only a single value. As described above, this corresponds to finding a $(k,1)$ orientation on a $(2k-1)$-regular hypergraph (V, E).
If such an allocation exists, it can be found efficiently using a max-flow algorithm.

Theorem 2. Given $2k-1$ hash functions, $h_i : \mathcal{U} \to [n]$, and $\{x_i\}_{i=1}^m \subset \mathcal{U}$, it is possible to find an allocation $A : [m] \to \mathcal{P}([2k-1])$ that allocates each element k hash functions, satisfying

1. Each element is allocated exactly k hash functions, i.e., $|A(i)| = k$ for all $i \in [m]$.
2. Each bucket has at most one element, i.e., for all $i \neq i' \in [m]$, $\{h_j(x_i)\}_{j \in A(i)}$ and $\{h_j(x_{i'})\}_{j \in A(i')}$ are disjoint.

and the allocation can be found in time $O(nk)$.

Proof. We begin by translating the problem to finding a $(k,1)$-orientation in an $(2k-1)$-regular hypergraph $G_{\mathsf{hyper}} = (V_{\mathsf{hyper}}, H_{\mathsf{hyper}})$, with n vertices and m edges. As described above, each bucket corresponds to an vertex in the hypergraph, and each element $x \in \mathcal{U}$ corresponds to a hyperedge, containing the $2k-1$ vertices $\{h_1(x), \ldots, h_{2k-1}(x)\}$.
Finding a $(k,1)$-orientation of this hypergraph corresponds to finding an orientation, where each hyper-edge "points" to k of its vertices, and each vertex has in-degree at most 1.
This problem can be translated to a problem on a flow network as follows.
Consider a network with node set with nodes for each vertex and each edge in the hypergraph, thus the set of nodes is $H_{\mathsf{hyper}} \cup V_{\mathsf{hyper}} \cup \{s,t\}$. For each edge in the hypergraph, put a capacity 1 edge connected the node corresponding to that edge with the $2k-1$ nodes corresponding to the vertices in V_{hyper} that it contains. Add capacity k edges from the source s to each vertex corresponding to an edge in H_{hyper}, and capacity 1 edges from each vertex corresponding to a vertex in V_{hyper} to the sink, t.
Now, note that any $|H_{\mathsf{hyper}}|(k)$ flow from s to t in the flow network induces a $(k,1)$ orientation in the hypergraph G_{hyper}. Conversely, any $(k,1)$ orientation of the hypergraph corresponds exactly to an $|H_{\mathsf{hyper}}|(k)$ flow from s to t in the induced flow network.
Using the Ford-Fulkerson algorithm, the max flow can be computed in time $O((|V_{\mathsf{hyper}}| + |H_{\mathsf{hyper}}|)(k)) = O((m+n)k) = O(nk)$. □