Private Set Intersection

Bradley Cushing
New York University
Courant Institute of Mathematical Sciences
bsc6146@nyu.edu

This is a working draft (last updated on 5/9/2025) and summary of research which may contain incomplete information and incorrect or limited references to external sources. As a result, until this disclaimer is removed, nothing stated in this document should be considered as complete.

Abstract

Private Set Intersection (PSI) is a secure Multi-Party Computation (MPC) protocol that allows n parties to compute the intersection of n sets of objects, where at the end of the protocol each party learns the intersection of these sets but nothing more. This application of MPC has been an active area of research over the past 20 years where many variants of PSI using different primitives have been studied. This work attempts to introduce some of the different approaches to building PSI protocols, their variants, and the underlying primitives, giving the reader an understanding of many of the ways it can be achieved.

1 Introduction

In this paper we cover different protocols for PSI and the primitives used to build them. In Section 2 we discuss "Private Preference Matching", "PSI from Random Garbled Bloom Filters", "OT-Based PSI from PEQT", "Private Matching from Homomorphic Encryption", "Multi-party PSI from Symmetric-Key Techniques", "Updatable Private Set Intersection", and "Structure-Aware PSI and Fuzzy Matching".

In Appendix A we cover the primtives used the build each of the protocols in detail. These primitives include oblivious transfer (OT), random OT, OT extension, oblivious polynomial evaluation (OPE) built from additive homomorphic encryption, oblivious pesudorandom functions (OPRF), oblivious programmable pesudorandom functions (OPPRF), oblivious psudorandom generators (OPRG), Cuckoo hashing, and Bloom filters.

1.1 Motivation

The primary motivation for this survey is to explore in enough detail, some of the different techniques and primitives that are used to bulid PSI protocols with semi-honest security. While some of the ideas overlap a bit between each protocol, the main ideas behind them are mostly disjoint, and complement each other well. We assume no prior knowledge of PSI or any of the key primitives that are used to build it.

1.2 Definition

Two-party PSI is a protocol π, where party P_i has a set of size n and party P_j has a set of size m, where n doesn't necessarily equal m. We model this protocol [5] in terms of it's ideal functionality.

Functionality:
- Two parties, P_1 with a set size n, P_2 with a set size m.
- Wait for input X from party P_1 and Y from party P_2.
- Output $X \cap Y$ to both parties.

The goal of each of the protocols that we present in this paper is to realize this ideal functionality securely, through the use of cryptography. When a protocol implements a different functionality, like multi-party PSI, we state it explicitly in the same place the protocol is covered. We do the same for some slight variants like PSI aimed at finding the cardinality of the intersection or in the case of fuzzy search of rows in a database.

1.3 Adversary models

1.3.1 Semi-honest security

There are several ways to model security in MPC and specialized PSI protocols. We primarily focus on the semi-honest (i.e. honest-but-curious) security model where adversaries may collude with each other but must otherwise, honestly follow the rules of the protocol. We give light proofs of security in this model by means of indistinguishability and ideal functionality arguments.

We say a protocol π is secure in the semi-honest model if we can build a simulator s.t. the views of the adversaries participating in the protocol are the same as the views of the adversaries interacting with a simulator. More formally, $Views_S(\pi(\vec{x})) \equiv Sim(\vec{x_s}, f(\vec{x}))$ where the simulator is given the input of the adversaries and the output of the function f, computing the intersection. In a two-party protocol we only need to consider that one of the parties may be an adversary. In a multi-party protocol with n parties, we may have up to $n-1$ adversaries.

1.4 Generic MPC

Our work is focused on building more efficient, custom PSI protocols. We note that PSI can be built from generic MPC protocols including but not limited to Yao's Garbled Circuits, GMW, and BGW. While this is possible, less versatile, custom protocols that depend on the structure of the problem can achieve better communication and computational overhead.

1.5 Intuition

We first give the most basic technique for achieving PSI between two parties and explain where there is room for improvement. Consider two parties, P_1 with a set X, and P_2 with a set Y. P_1 computes the set $H_X = \{H(x)|x \in X\}$ using an agreed upon hash function H, and sends it to P_2. P_2 does the same, computing H_Y for Y, and and sends H_Y to P_1. Now each party can compare the values of H_X and H_Y independently, finding those that match, and learning the intersection.

At first glance this seems secure when H is modeled as a random oracle. P_1 learns nothing if $H(x) \notin H_Y$, and symmetrically P_2 learns nothing if $H(y) \notin H_X$. We note that it is secure as long as the input sets have high entropy [11], but in practice this is often not the case. Since any party can evaluate H non-interactively, this protocol is susceptible to a brute force dictionary attack. Nothing prevents party P_i from precomputing H_Z for any set Z, and learning the intersection for anything they would like.

2 Protocols

2.1 Private Preference Matching

A protocol for "Private Preference Matching" [9] was first introduced in 1999, based on the Diffie-Hellman key exchange protocol. This is an interactive protocol which allows us to evaluate a function between two parties to compare items in a set. We note the importance of interaction in any PSI protocol to avoid dictionary attacks. We give the basic construction and explore some variants.

Protocol. Consider two parties Alice and Bob who would like to learn the intersection of items in their respective sets. More formally, Alice has the list $x_1, \dots, x_n$ and Bob has the list $y_1, \dots, y_m$, where at the end of the protocol π, both Alice and Bob learn $x_i = y_j$ for any $i \leq n, j \leq m$ but nothing else.

Let $\mathbb{G}$ be a group and $H : \{0,1\}^* \to \mathbb{G}$ be a hash function, with all operations in mod p. Alice and Bob privately sample $a, b \overset{R}{\leftarrow} \mathbb{G}$, respectively. Alice sends $H(x_1)^a, \dots, H(x_n)^a$ to Bob. Bob sends $H(y_1)^b, \dots, H(y_m)^b$ to Alice. Bob sends $H(x_1)^{ab}, \dots, H(x_n)^{ab}$ to Alice. Alice sends $H(y_1)^{ba}, \dots, H(y_m)^{ba}$ to Bob. Both output x_i, y_j where $H(x_i)^{ab} = H(y_j)^{ba}$.

Note that Alice and Bob both have $H(x_i)^{ab}$ and $H(y_j)^{ba}$ for all $i \leq n, j \leq m$. If $H(x_i)^{ab} = H(y_j)^{ba}$, then $x_i = y_j$. Assuming order is retained in the sending of messages at each step of the protocol, both Alice and Bob know their own preimage x_i, y_i respectively. Therefore, they each learn and output the shared preimage of $H(\cdot)^{ab} = H(\cdot)^{ba}$, where each preimage is in the intersection.

Security. From the perspective of Alice, there are two cases where either $H(x_i)^{ab} = H(y_j)^{ba}$ or $H(x_i)^{ab} \neq H(y_j)^{ba}$. For the first case Alice would have received $H(y_j)^b$ from Bob where y_j is in the intersection. The simulator has $f(\vec{x}, \vec{y})$ and so it can sample $b \overset{R}{\leftarrow} \mathbb{G}$ and send $H(y_i)^b$ to Alice. For the $m - |f(\vec{x}, \vec{y})|$ values not in the intersection, the simulator samples $r_i \overset{R}{\leftarrow} \mathbb{G}$ for the i-th item not in the intersection and sends $H(r_i)^b$ which has the same distribution. If we model H as a random oracle confined to some group $\mathbb{G}$, by the one-more gap Diffie-Hellman assumption [3], the values not in the intersection look random to Alice. Therefore, we can simulate Alice's view and by a symmetric argument simulate Bob's the same way, so the protocol is secure.

Variants. By permuting the items at different steps we can also create variants of this protocol which compute something different, changing the output of f and what each party learns during the process. We can learn the cardinality of intersection if both parties randomly permute their sets before sending them in steps 2-5. The intuition is that since either party doesn't have access to the other's secret, anything they receive looks random and since the lists are permuted they can no longer rely on their original, local ordering.

2.2 PSI from Random Garbled Bloom Filters

It was shown that Bloom filters and garbled Bloom filters can be used together to create a two-party PSI protocol [4] in the semi-honest model for large sets. We explain an improved protocol below from 2013 using an extension called the random garbled Bloom filter (RGBF) and an oblivious pseudorandom generator (OPRG) [13], both covered in section 3, which allow us to reduce some of the necessary computation even further.

Protocol. The idea is for two parties to generate a shared view of the intersection by interactively building their own RGBF. Each RGBF can then be used to compute a random value for each entry in their respective input sets. These random values will match for any two distinct values that are present in both sets so can be used to compute the intersection.

Consider two parties P_1 and P_2 with equal sets of size n for simplicity. Both parties generate their own regular m-bit Bloom filters F_x and F_y, respectively. They then evaluate a m instances of an OPRG, P_1 using $F_x[i]$ and P_2 using $F_y[i]$ as input for the i-th instance. Each party uses the output $r_i \in \{0,1\}^\ell$ of the OPRG to build their own RGBF s.t. the i-th output maps to $R_x[i], R_y[i]$, respectively. Remember that if both input 1 for the same instance of the OPRG then their received output will be the same random string. Note that there will be some values where both $F_x[i] = F_y[i] = 0$ and the OPRG will output $\perp$ to both P_1 and P_2, so we can ignore all the computation for this case, making this construction more efficient.

It's now possible for P_1 to compute $m_{P_1}[j] = \bigoplus_{i=1}^{\kappa} R_x[h_i(x_j)]$ where h_i is the i-th hash function for the RGBF and x_j is item j in the input set X for some $j \leq n$. P_1 randomly permutes the ℓn-bit string m_{P_1} and sends it to P_2. Now P_2 is able to compute $m_{P_2}[j] = \bigoplus_{i=1}^{\kappa} R_y[h_i(y_j)]$ for each y_j in it's own input set Y for some $j \leq n$ and check if $m_{P_2}[j] \in m_{P_1}$. Correctness comes from the fact that whenever $F_x[i] = F_y[i] = 1$, then $R_x[i] = R_y[i]$ and so $m_{P_1}[i] = m_{P_2}[j]$ when $x_i = y_j$ for some $i, j \leq n$.

Security. When considering the security of P_2, we can observe that the view of P_1 is just the random set of values received when interacting with P_2 using the OPRG. In this case the simulator can just send random values to P_1 when it's input bit to the OPRG is 1 and otherwise send $\perp$.

For security of P_1, we consider the view of P_2 which is also these random values but additionally contains the output in the second phase. This output consists of the XOR values from the RGBF for each item in the input set X of P_1. These values look random to P_2 for any value $x_i \neq y_j$ for any $i, j \leq n$ or are the same as values in its own RBGF when $x_i = y_j$ and so they are both easy to simulate, the latter given the internal state of P_2.

2.3 OT-Based PSI from PEQT

There's an efficient PSI protocol [13] from 2014 based on oblivious transfer (OT), which comes directly from private set inclusion, itself built on a protocol for private equality testing (PEQT). The protocol is made even more efficient through the use of a stronger form of balanced hashing called Cuckoo hashing [12]. This technique allows us to limit the number of comparisons between input elements, resulting in a smaller number of OTs. We first explain how to build the simpler PEQT protocol, extend it to one for private set inclusion, and finally an efficient protocol for private set intersection.

PEQT. The goal of PEQT is to determine if $x = y$, given a single input x from P_1 and a single input y from P_2. P_1 and P_2 start the protocol by engaging in a random $\binom{2}{1}$ OT_ℓ^σ where $|x| = |y|$ are both σ-bit strings for simplicity. P_2 uses the σ bits of y as it's selection string in the OT protocol, receiving $s^i_{y[i]}$ from the randomly generated ℓ-bit strings (s^i_0, s^i_1) at the ith round of the protocol.

P_1 then computes and sends $m_{P_1} = \bigoplus_{i=1}^{\sigma} s^i_{x[i]}$ to P_2. P_2 compares m_{P_1} with it's own m_{P_2} which it can compute in the same way using x as selection bits for the random strings. This allows P_2 to determine $x = y$ iff $m_{P_2} = m_{P_1}$. Note that it's possible to use a different base-N representation of x and y for selection which can result in a more optimal $\binom{N}{1}$ OT_ℓ^σ protocol for some different choice of N.

Private set inclusion. We can extend PEQT to the where P_1 has an input set $X = \{x_1, \dots, x_n\}$ and P_2 has a single input y. We could naively run PEQT for each $x_i \in X$ for all $i \leq n$, but we can actually do better. Consider for some base-N that we can engage in $\binom{N}{1}$ $OT^t_{\ell n}$ where t times, we exchange random strings of size ℓn.

P_2 obtains $s^i_{y[i]}$ as before but where $s^i_{y[i]}[j]$ represents the j-th substring for all $j \leq n$. We now have a random value pertaining to the i-th bit of x_j if we're considering base-N where $N = 2$. Similar to before, P_1 can compute $m_{P_1}[j] = \bigoplus_{i=1}^t s^i_{x_j[i]}[j]$ and send m_{P_1} to P_2 where $|m_{P_1}| = \ell n$. P_2 can then compute $m_{P_2}[j] = \bigoplus_{i=1}^t s^i_{y[i]}$ and know $m_{P_2}[j] \in m_{P_1}$ iff $y \in X$. We can further optimize the OTs to that of PEQT by shrinking the length of the exchanged ℓn-bit strings down to ℓ and expand them using a PRG.

Private set intersection. To obtain $X \cap Y$ we can invoke the above private set inclusion protocol for each value of $y \in Y$. Correctness and security follow directly from the PEQT protocol. Note that the protocol becomes inefficient as n grows, increasing the amount of communication due to increasing number of OTs. We can reduce the number of OTs by limiting the number comparisions if we use of Cuckoo hashing.

Security. In the case of PEQT, the security of P_2 comes from the fact that P_1 only learns the random values generated in the OTs so we can simulate the view of P_1 by randomly sampling these values. Considering the security of P_1, the view of P_2 is the set of random values output from each OT corresponding to it's σ selections using the value y along with the value m_{P_1} which is the XOR of all ℓ-bit strings using the x of P_2 for selection. Therefore, P_2 only learns x when $x = y$ and $m_{P_1} = m_{P_2}$, otherwise m_{P_1} looks random. To simulate the view of P_2 we can either send a random value when $x \neq y$ or the exact value of m_{P_2} when $x = y$. The same simulation argument extends to private set inclusion and subsequently the full protocol.

2.4 Private Matching from Homomorphic Encryption

We can obtain two-party and multi-party PSI from homomorphic encryption [7] like Pallier, which allows for addition and multiplication by a constant. This idea from 2004 relies on oblivious polynomial evaluation (OPE), where a receiver R can encrypt the coefficients of a polynomial P, which a sender S can evaluate on a private input x, and where only R can recover the result. Note that R doesn't learn the input of S and S doesn't learn the actual polynomial P.

Protocol. The protocol starts with the client C choosing a homomorphic encryption scheme and publishing it's public key and public parameters. C then computes a polynomial P s.t. the roots of the polynomial are the values of it's input set $Y = \{y_0, \dots, y_n\}$. This can be done easily by computing

$P(y) = (x_1 - y)\dots(x_n - y) = \sum_{i=0}^{n} \alpha_i y^i$. C then sends the homomorphic encryption of the coefficients of P to the server S.

S evaluates the polynomial P on it's own input $z_i \in Z = \{z_0, \dots, z_m\}$ for all $i \leq m$, multiplies each result by a random number, and adds it to an encryption of z_i. This is equivalent to S computing $Enc(r_i \cdot P(z_i) + z_i)$ explained above. C then permutes all computed values and sends them to S who can decrypt and compare each value to the values in it's own set Y. Note that since any y_j for some $j \leq n$ is a root of the polynomial P, then $P(y_j) = 0$, and so $Enc(r_i \cdot P(z_i) + z_i) = Enc(z_i) = Enc(y_j)$ iff $z_i = y_j$ for some $i \leq m$.

The dominating cost in this scheme is the number of exponentiations in the homomorphic encryption scheme. We can reduce this number by reducing the domain of the inputs and by using Horner's rule when evaluating the polynomial to remove large exponents. Computing high-degree polynomials by C is also expensive but this can be reduced by using balanced hashing or Cuckoo hashing. Balanced hashing assigns each input item to one of a number of bins B, where each bin has no more than M items. We can build low-degree polynomials for a limited number of input items per bin and evaluate those instead. We also make sure each bin has M items exactly by adding an additional number of zero roots to hide the number of items per bin.

Security. The privacy of C comes from the fact that S only sees semantically-secure, homomorphic encryptions of the coefficients of P. Therefore, to simulate the view of S we can randomly sample any coefficients which we encrypt and send to S.

The Privacy of S comes from the fact that C only receives "meaningful" encryptions when $z_i = y_j$ for some $i \leq m$ and $j \leq n$, and otherwise encryptions of random values. Given values in the intersection, the simulator can just encrypt these values and send them to S. For values not in the intersection, the simulator can encrypt a randomly sampled value which is distrubted the same, and send that to S.

Variants. We define fuzzy search to be where C would like to obtain a row of attributes from a database held by S. Consider where C has a set of attributes Z and S has a set of attributes Y for some row in the database. The protocol is identical to before except that S will compute $Enc(r_i \cdot P(z_i) + s_i)$ where s_i is the i-th secret share of Y for $i \leq t$ in a t-threshold secret sharing scheme. If the number of matches is $\leq t$ then the shares are enough for C to recover Y.

2.5 Multi-party PSI from Symmetric-Key Techniques

Kolesnikov et. al [10] introduced a multi-party PSI protocol in 2017 for $n \geq 2$ by introducing the notion of a new primitive, programmable oblivious pseduorandom funtcion (OPPRF). The security we want to achieve is that each party P_i learns the intersection $\bigcap_{=1}^{n} X_i$ at the end of the protocol π but nothing else.

Functionality:

- n parties, P_i to P_n all with a set size m.
- Wait for input $X_i = \{x_i^1, \ldots, x_i^m\}$ from each party P_i.
- Output $\bigcap_{i=1}^n X_i$ to all parties.

Two-party protocol. The idea is that each party can use an OPPRF to program a set of shares that addively XOR to zero for each value in their input set. If each party evaluates such an OPPRF with each other on the same input value, then at the end of the protocol when these shares are collected, all shares for that value across all parties will also XOR to zero.

There are two phases of the protocol, a conditional zero-sharing phase and conditional reconstruction phase. We consider what happens in the two-party case first to make things simpler. In conditional zero-sharing, party P_1 creates a mapping to shares $S_1(x_k^1) = s_k$ where $x_k^1 \in X_1$ is some k-th item in the set X_1 of P_1 and s_k is a random string. These shares have the property s.t. if $x \in \bigcap_{i=1}^n X_i$, in this case $x \in X_1 \cap X_2$, the corresponding shares XOR to zero. P_1 then programs an OPPRF s.t when queried by P_2 on x_k^2, it outputs some $s'_k = S_2(x_k^2)$. If the parties share $x_k^1 = x_k^2$ then, both parties share the same mapping and we have that $S_1(x_k^1) = S_2(x_k^2) = s_k$.

In conditional reconstruction, party P_1 acts as a special "dealer" to collect all the shares that pertain to it's input set X_1 from the other party P_2. Party P_2 programs an OPPRF again, s.t. when queried on x_k^1 it will output the same value $S_2(x_k^1) = S_1(x_k^1) = s_k$ when $x_k^1 \in X_2$. Otherwise it will output some value s'_k which is mapped randomly. If $x_k^1 = x_k^2$, then $S_2(x_k^1) \oplus S_1(x_k^1) = s_k \oplus s_k = 0$ and so P_1 knows x_k^1 is in the intersection.

Multi-party protocol. We can see how the above two-party example can be expanded to $n > 2$ parties. In the conditional zero-sharing phase, each sender P_i creates n shares $[s_k^i]$ for each $x_k^i \in X_i$ and programs an OPPRF that every other P_j party will evaluate as the receiver. These randomly generated shares have the additive property where $\bigoplus_{j=1}^n s_k^{i,j} = 0$ for all shares of some x_k^i sent from P_i to P_j. When the "dealer" collects all shares from the other $n-1$ parties for a given input, and this input was shared between them all, $\bigoplus_{i=1}^n [s_k^i] = 0$ due the additive XOR property.

It's important to note that OPPRF constructions, which themselves rely on an efficient OPRF, are efficient only when the number of programmed points is small. This limitation is overcome by using a variant of Cuckoo hashing which has no stash and ensures one item per bin with high probability. This allows us to reduce the number of comparisons made between items and limit the number of programmed points to $t = 1$.

Security. We consider two groups, the adversaries A and honest parties H, and two general cases. In the first case, all parties in A have x but not all parties in H have x. When $|H| = 1$, trivially the adversaries can learn the value of x given the output from the protocol. When $|H| > 1$, some honest party P_i has not programmed the value of x in their OPPRF. Security of the OPPRF ensures that any party in A learns nothing about which party this is. This holds after the first phase or the second phase in the role of the “dealer”.

In the second case, not all parties in A have x. Since some adversary in A doesn't have x, each honest party P_i will have a uniformly distributed $S_i(x)$ value for x due to that party. Then in the conditional reconstruction phase, values sent to the “dealer” are either programmed on x or not. Security of the OPPRF ensures that in both phases, the adversary can't distinguish between shares programmed or not programmed on x.

2.6 Structure-Aware PSI and Fuzzy Matching

A new paradigm called structure-aware PSI [8] was first introduced in 2022, reducing the problem to that of developing an efficient function secret-sharing (FSS) scheme. In this setting we want to learn the intersection of two sets $A \cap B$, where Alice has a structured set A, and Bob has an unstructured set B.

That paper introduces the notion of weak Bolean Function Secret Sharing (bFSS) which is a weaker notion of FSS, allowing for a more efficient protocol that scales with the description of the structured set. We cover the high-level details of the protocol assuming the properties of weak bFSS indicator functions for a family of sets.

Protocol. The general idea is that we can instantiate PSI using bFSS, where f is an indicator function for set membership. We denote the size of a share as σ which dominates the communication cost and is porportional to the description size d of the structured set. We note that the more structure we have, the smaller the description relative to the cardinality of the structured set.

Alice will first use the bFSS to compute security parameter κ independent sharings of her structured set A as $(k_0^{(i)}, k_1^{(i)}) \leftarrow Share(A)$. Bob then chooses a random string $s \leftarrow \{0,1\}^\kappa$ where each bit of s is used as a selection bit in κ instances of OT, where Bob would like to learn each share $k_{s_i}^{(i)}$ for all i where $1 \leq i \leq \kappa$. Bob defines a function $F(b) = H(Eval(k_{s_1}^{(1)}, b), \ldots, Eval(k_{s_\kappa}^{(\kappa)}, b))$, computes $F(b)$ for all $b \in B$, and sends all such $F(b)$ values to Alice. If $b = a$, then by correctness of the bFSS, we have that $Eval(k_*^{(i)}, b) = Eval(k_*^{(i)}, a)$ for any share $k_*^{(i)}$, where $k_*^{(i)} = k_0^{(i)} = k_1^{(i)}$ for all i. Alice can then compute $F(a) = H(Eval(k_0^{(1)}, a), \ldots, Eval(k_0^{(\kappa)}, a))$ for all $a \in A$ and know that $F(a) = F(b)$ if and only if $a = b$.

Security. The security of Alice comes from the fact that Bob only sees a single FSS share per OT instance. By security of the FSS, a single share leaks nothing about the structured set A. The security of Bob comes from the fact that Alice doesn't know Bob's selection string s. Even though Alice has all the shares, by security of the FSS, if some $b \notin A$, then $Eval(k_*^{(i)}, b) \neq Eval(k_*^{(i)}, a)$ for all $a \in A$, and Alice would have to guess s of which the the probability is $negl(\kappa)$.

Application. The primary and practical application given as an example is ridesharing, where we want to privately match a person with available cars within some distance δ threshold. Using structure-aware PSI for structured sets, we can represent this set geometrically as a union of ℓ_∞ balls. Constructions for more complex distance metrics like ℓ_1 and ℓ_2 are also possible.

2.7 Updatable Private Set Intersection

Another paradigm is that of updatable PSI (UPSI), where parties may add or remove items from their sets over time, while maintaining an intersection invariant. Newer work [1] defines PSI in two restricted models, that of UPSI with addition and UPSI with weak deletion. UPSI with addition allows each party only to add items to their sets while weak deletion allows adding of items in addition to deleting old items after t days. We focus on the simplest form, UPSI with addition, in a two-sided protocol.

Functionality:

- Two parties, P_1 with $X = \emptyset$, P_2 with $Y = \emptyset$.
- Wait for input X_d of size N_d from party P_1 where $X_d \cap X = \emptyset$.
- Wait for input Y_d of size N_d from party P_2 where $Y_d \cap Y = \emptyset$.
- On input from both, updates $X = X \cup X_d$, $Y = Y \cup Y_d$.
- Computes $I_d = X \cap Y$ and ouputs I_d to both parties.

The size of the update set on any day d is denoted N_d. This definition assumes for simplicity that items in the update are never duplicates and that both parties add the same number of items on each day. We note that naively we can achieve this functionality by rerunning any two-sided PSI protocol anytime there's an update. The goal of any UPSI protocol is therefore to be more efficient than this while maintining the same security.

References

[1] Saikrishna Badrinarayanan, Peihan Miao, and Tiancheng Xie. Updatable private set intersection. Cryptology ePrint Archive, Paper 2021/1349, 2021.

[2] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT (2)*, pages 337–367. Springer, 2015.

[3] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. SoK: Oblivious pseudorandom functions. Cryptology ePrint Archive, Paper 2022/302, 2022.

[4] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 789–800, New York, NY, USA, 2013. Association for Computing Machinery.

[5] David Evans, Vladimir Kolesnikov, and Mike Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation*, pages 26–63. NOW Publishers, 2018.

[6] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 303–324. Springer, 2005.

[7] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 1–19, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[8] Gayathri Garimella, Mike Rosulek, and Jaspal Singh. Structure-aware private set intersection, with applications to fuzzy matching. Cryptology ePrint Archive, Paper 2022/1011, 2022.

[9] Bernardo A. Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, page 78–86, New York, NY, USA, 1999. Association for Computing Machinery.

[10] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1257–1272, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Marcin Nagy, Emiliano De Cristofaro, Alexandra Dmitrienko, N. Asokan, and Ahmad-Reza Sadeghi. Do i know you? efficient and privacy-preserving common friend-finder protocols and applications. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, page 159–168, New York, NY, USA, 2013. Association for Computing Machinery.

[12] R. Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51:122–144, 2001.

[13] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. Cryptology ePrint Archive, Paper 2014/447, 2014.

A Primitives

A.1 Oblivious Transfer

We start by explaining oblivious transfer (OT) [5] because of it's usefulness in building protocols and other primitives. We show an OPRF, OPPRF, and OPRG construction later in this section which all rely on OT and a more effcient form of batching called OT extension.

A.1.1 1-out-of-2 OT

The simplest form of OT involves two parties, a sender S and receiver R, where R can request a single secret from S who has two secrets (s_1, s_2). For a private choice of $b \in \{0,1\}$ by R, we require that R receives s_b from S where R doesn't learn s_{b-1} and S learns nothing about b.

Functionality:

- Two parties: Sender S and Receiver R.
- S has two secrets, $x_0, x_1 \in \{0,1\}^n$.
- R has a selection bit, $b \in \{0,1\}$.
- R receives x_b, S receives $\perp$.

Protocol. We can implement the above functionality using public-key cryptography, where we it's poossible to randomly sample public keys without a secret key. First, R generates a public-private key pair sk, pk and samples a random public key (without a secret key), pk', from the public key space. If $b = 0$, R sends the pair (pk, pk') to S, otherwise $(b = 1)$ and R sends the pair (pk', pk). S receives (pk_0, pk_1) and sends back to R the two encryptions $e_0 = Enc_{pk_0}(x_0), e_1 = Enc_{pk_1}(x_1)$. Finally, R receives e_0, e_1 and decrypts the ciphertext e_b using sk. Note that R is unable to decrypt the second ciphertext as it does not have the corresponding secret key.

Security. The privacy of R is determined by the view of S who only sees two public keys. The simulator can randomly sample two public keys which it sends to S. Since S doesn't have a secret key, these public keys look the same as those in the real protocol.

The privacy of S is determined by the view of R who sees two encryptions e_0, e_1, where it can decrypt only one of them with pk. The simulator can sample sk, pk and pk', encrypting the value of x_b with pk, and 0 with pk'. R can use sk to decrypt $e_b = Enc_{pk}(x_b)$ and by the security of the encryption scheme, $e_{b-1} = Enc_{pk'}(0) \approx Enc_{pk'}(x_{b-1})$.

A.1.2 Random OT

Functionality:

- Two parties: Sender S and Receiver R.
- S generates two secrets, $x_0, x_1 \overset{R}{\leftarrow} \{0,1\}^n$ uniformly.
- R has a selection bit, $b \in \{0,1\}$.
- R receives x_b, S receives $\perp$.

Random OT is the same as the basic OT protocol above, except that the secrets x_0, x_1 are randomly sampled during the protocol. This small change allows us to reduce the amount of communication needed between S to R in the last round. Some of the PSI protocols we review also use the protocol-time randomness generation property to reduce some of the necessary computation.

A.1.3 OT extension

OT can be extended [5] from 1-out-of-2 OT to 1-out-of-n OT, 1-out-of-2^ℓ OT, and 1-out-of-∞ OT. If we have a protocol which requires many OTs, then we must efficiently generate them for the protocol to be efficient. OT extension provides a way to generate some large number of m OTs using only some constant number of k base OTs. These protocols only require a constant number of expensive, public-key operations, and leverage more efficient, symmetric-key operations instead.

A.2 OPE

Oblivious polynomial evaluation (OPE) allows for two parties to interactively evaluate a polynomial P. P_1 has some private polynomial P and P_2 has a some private input x. We want that P_1 learns the result $P(x)$ without learning x and that P_2 doesn't learn the polynomial P.

Functionality:
- Two parties, P_1 and P_2.
- P_1 holds a a polynomial P; P_2 holds an evaluation point x.
- P_1 outputs $P(x)$; P_2 outputs nothing.

Protocol. One method of building OPE uses homomorphic encryption by having P_1 encrypt the coefficients of P representing some new polynomial P', which is then sent to P_2. P_2 can then evaluate the polynomial P' where for the output on any point y we have $P'(y) = Enc(P(y))$. P_2 sends $Enc(P(y))$ to P_1 who then decrypts to obtain $Dec(Enc(P(y))) = P(y)$.

A.3 OPRF

An oblivious pseudorandom function (OPRF) [6] and more specifically the notion of a strongly-private OPRF, allows for two parties to compute a psuedorandom function (PRF), where neither party learn each others input and only one party learns the result.

Functionality:
- Two parties, P_1 and P_2.
- P_1 holds an evaluation point w; P_2 holds a key r.
- P_1 outputs $f_r(w)$; P_2 outputs nothing.

Protocol. A simple scheme [3] is based on Diffie-Hellman key exchange, which relies on blind exponentiation to securely implement the ideal functionality. Alice privately samples $a \overset{R}{\leftarrow} \mathbb{G}$ and sends $H(w)^a$ to Bob for some private evalutation point w. Bob privately samples a key $r \overset{R}{\leftarrow} \mathbb{G}$ and sends $H(w)^{ar}$ back to Alice. Finally, Alice computes $H(w)^{ar\cdot(1/a)} = H(w)^r = f_r(w)$.

Security. When H is modeled as a random function whose output is in the range of the group $\mathbb{G}$, this construction is secure under the one-more gap Diffie-Hellman assumption. Alice can't determine $H(w)$ given $H(w)^a$ and Bob can't determine r given $H(w)^{ar}$.

A.3.1 OPPRF

An oblivious programmable pseudorandom function (OPPRF) [10] is the same as an OPRF but with one additional property. It allows the sender S to initially provide a set of points P which can be programmed and for R to make up to t queries. When the OPPRF is evaluated by the receiver R on input x_i for point p_i, it should output the programmed y_i.

Functionality:

- Two parties: Sender S and Receiver R.
- S has a set of points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ and $(k, hint)$.
- R has a set of queries $(q_1, \ldots, q_t)$ and $F(k, q)$ for $q \in Q$.
- R receives $(hint, F(k, hint, q_1), \ldots, F(k, hint, q_t))$.

PPRF. To implement the ideal functionality above, we first consider the notion of a programmable psuedorandom function (PPRF). We define correctness such that if a point $p_i = (x_i, y_i)$ is programmed by the S, if the receiver R queries the PPRF with input x_i, then it should always output the programmed y_i.

There are a few different ways to instantiate a PPRF, including but not limited to polynomials and garbled Bloom filters. The intuition is that we can use either to store a mapping $x \mapsto y$. If y is chosen uniformly then it should be indistinguishable from the random output of a PRF. We show the construction for a PPRF $\hat{F}$ from a basic PRF F and polynomials.

To build $\hat{F}$, the sender S generates a set of points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ where y_i is chosen uniformly, chooses a random key k for F, and interpolates a degree $n - 1$ polynomial l over the points $\{(x_1, y_1 \oplus F(k, x_1)), \ldots, (x_n, y_n \oplus F(k, x_n))\}$. We define the $hint$ to be the coefficients of L and q some input query where $\hat{F}(k, hint, q) = F(k, q) \oplus l(q)$. Correctness follows from the fact that $F(k, q) \oplus l(q) = F(k, q) \oplus y_i \oplus F(k, q) = y_i$.

Security. Security of the PPRF states that R should not be able to tell what the programmed points are, given the $hint$ and t outputs. More formally, for the input sets $|X_1| = |X_2| = n$ and $|Q| = t$ queries we have that $|\Pr[A^{PPRF}(X_1, Q, \kappa) = 1] - \Pr[A^{PPRF}(X_2, Q, \kappa) = 1]| = negl(\kappa)$. We make the distinction that each programmed y_i is uniformly random while non-programmed points are pseudorandom.

OPPRF. We can naturally state an OPPRF protocol for $\hat{F}$ if we have an OPRF protocol for F. Running the protocol for F, S obtains k and can compute l while R can compute $\hat{F}(k, hint, q) = F(k, q) \oplus p(q)$ for some query q. Correctness and security of the OPPRF come directly from the PPRF.

A.4 OPRG

An oblivious pseudorandom generator (OPRG) [13] allows two parties to conditionally generate shared randomness. In the end of the protocol, the OPRG will output a random string to either one, both, or none of the parties depending on two selection bits, one from each party.

Functionality:

- Two parties, P_1 and P_2.
- Party P_i has a selection bit b_i.
- Generates a random string s.
- Outputs s to P_i if $b_i = 1$, otherwise $\perp$.

Importantly, party P_i won't learn if party P_{i-1} received the random string s or not. We mention that this primitive can be built from random OT extension but don't give a construction. We note that this primtive gives us the useful property that when both $b_1, b_2 = 0$, then no party will receive any output, and so we can ignore computation in that case. This allows us to improve efficiency of PSI protocols using this primitive.

A.5 Cuckoo hashing

Dynamic data structures like Cuckoo hashing [12] give us a single dictionary with worst case constant lookup time and efficient space. Many of the protocols we cover rely on Cuckoo hashing to achieve efficiency in OPRF evaluations by limiting item comparisons.

Cuckoo hashing provides semi-predictable locations in a distributed environment. In the context of PSI, we often want to limit comparisons between items to a small set of relevant inputs. If we can target items by some remote lookup location, we can limit the number of locations where we need to look for a given input, leading to a more efficient protocol. Leveraging oblivious primmitives we've already covered, this can be done without leaking information.

Construction. Consider the generic form where we have b bins and a set of k hash functions $h_1, \ldots, h_k$ where $h_i : \{0,1\}^* \rightarrow [1, b]$ maps an item x to some bin. Given some item x, we attempt to place item x in bin $h_1(x)$. If this bin is empty, then we place it there and we're done, otherwise we evict the current value y before placing it. Now, we attempt to place item y using $h_i(y) \neq h_1(x)$ where $i \in \{1, \ldots, k\}$. We keep repeating the above until we have no more collisions or until some finite number of relocations is reached, at which point we append the item to a special bin s which we call the stash. We note that stash can have any number of items in any order depending on the number of collisions.

Depending on the number of bins b and hash functions h, our data structure will have different properties. With a large number of items, small number of hash functions, and small number of bins, it's likely that more items end up in the stash. Increasing the number of hash functions and bins yields more choices for mappings which can mean less items in the stash. In the extreme case we can create a data structure without a stash with high probability.

Variants. Cuckoo hashing is a special case of a more general data structure called balanced hashing. In balanced hashing it's possible to have more than one item per bin given some threshold with high probability. Cuckoo hashing is often used in tandem with simple hashing, where one party maps each input x to every possible bin b, ensuring matching items between parties will always be compared.

A.6 Bloom filters

Another useful data structure is the Bloom filter (BF) which gives a way to determine set membership using only a single m string for storage. For a Bloom filter F, let S be a set where $|S| = n$, F be an m-bit string, and $h_1, \dots, h_\kappa$ be κ independent hash functions where $h_i : \{0,1\}^* \to [1, m]$ for $1 \leq i \leq m$.

Bloom filter. We first initialize the BF where $F = 0^m$. To insert an item y, set $F[h_i(y)] = 1$ for all $i \leq \kappa$. To check the presence of an item, check $F[h_i(y)] = 1$ for all $i \leq \kappa$ and if so, then y is in the Bloom filter with high probability, otherwise it is not. We set the security parameter κ s.t. the probability of a false positive is $\epsilon = 2^{-\kappa}$.

Garbled Bloom filter. The garbled Bloom filter (GBF) [4] is a more powerful extension of the previous idea. For some GBF G, instead of setting single bits to 1 in an m-bit string, we store generate and store random ℓ-bit strings s.t. if y is in the GBF then $\bigoplus_{i=1}^{\kappa} G[h_i(y)] = y$, otherwise it is not.

Similar to before, we initialize G with "uninitialized" values for all $i \leq \kappa$. To insert an item y, set each unset value $G[h_i(y)]$ s.t we maintain the invariant $\bigoplus G_{i=1}^{\kappa}[h_i(y)] = y$. If some value $G[h_i(y)]$ has already been set, we only set the remaining values such that the invariant holds. We note that with high probability, $G[h_i(y)]$ is not yet set for some $i \leq \kappa$. After all values have been added to the GBF, we add random values in all of the remaining slots. As with basic Bloom filters, there is some negligible probability of false positives.

Random garbled Bloom filter. There is one more extended notion called the random garbled Bloom filter (RGBF). This notion is identical to that of garbled Bloom filters, except with a change to the invariant $\bigoplus R_{i=1}^{\kappa}[y] = \mathcal{U}_\ell$. Now we have that the result of the XOR operation can be any random value.

Intuitively, if two parties are able to each generate an RGBF interactively, s.t some of it's entries are shared between them, then both parties can use it to determine if they share a value y. Protocols built using this technique gain efficiency from the fact they can leverage random OT extension which itself is more efficient.

A.7 Function Secret Sharing

Two-party Function Secret Sharing (FSS) was first introduced by Boyle, Gilboa, and Ishai [2] in 2015. It allows two parties to distribute shares (f_1, f_2) of a function f, which individually hide the function f, but where $f(x) = f_1(x) \oplus f_2(x)$ for all inputs x. Security of FSS says that we can simulate each individual share with respect to the security parameter κ. We give the more formal definition for FSS syntax as stated in the appendix of GRS22 [8].

FSS. A two-party function secret sharing scheme (FSS) for a class of funtions F with input domain $\{0,1\}^n$ and co-domain $\{0,1\}^m$ consists of a pair of algoritms $(Share, Eval)$ and a security parameter κ.

- $(k_0, k_1) \leftarrow Share(1^\kappa, \hat{f})$: The randomized share function takes as input the security parameter κ and the function description $\hat{f}$ for some function $f \in F$, and it outputs two keys, representing shares of the function f.
- $y_p \leftarrow Eval(1^\kappa, idx, k_{idx}, x)$: The deterministic evaluation function takes as input the security parameter, party index $idx \in \{0,1\}$, the corresponding FSS key k_{idx}, and the input $x \in \{0,1\}^n$, and it outputs $y_{idx} \in \{0,1\}^m$.

bFSS. Boolean Function Secret Sharing (bFSS) is a special case of FSS, where $f \in F$ is an indicator function for set membership. The indicator function evaluates to 0 when the input x is included in the set, and 1 otherwise. bFSS comes in two flavors, weak (p,k)-bFSS and strong bFSS (i.e. $(1,1)$-bFSS), where p is the false positive error probability and k is the share output length. We note that weak (p,k)-bFSS can be built and used to create more efficient structure-aware PSI schemes.