Survey of Private Set Intersection Protocols & their Applicability to Private Contact Discovery

Wyatt Howe U01809769
CS 568 Applied Cryptography, Boston University
whowe@bu.edu

Abstract

Undergraduate-level explanation of several ways to build a private set intersection protocol and contrast benefits in security, efficiency as well as explaining the underlying primitives used for each type.

1 Introduction

As our society moves slowly towards more privacy oriented technology, companies are having to take into consideration not only new laws and regulations of what data they are allowed to store, but what they owe to their users, as the many people are becoming more cautious about their private data. This means simple things like names and phone numbers. When users join a new messaging service (if they care about privacy) they will want to be able to control what contact info is accessible to others, and that the number of their own personal contacts they share to the service is at a minimum. This report focuses on one particular case of protecting data in use: protecting data in set intersection.

1.1 Set Intersection

Many data sharing scenarios have an objective algorithmic reducible to set intersection. Most basic problem is set as follows: Two or more parties each have a set of items, and they wish to find out which items they both have in common. Publicly this process is very simple just look at the other's set or sets and if you see anything familiar, it's a match. But while, as it many times is, this joint process trades efficiency for privacy, the parties lose all trace of privacy as they have all seen the contents of all other sets.

1.2 Set Intersection, Privately

Private Set Intersection (PSI) solves the same problem, but with a different approach—now privacy is the most important, and efficiency comes second only after all secrets that can remain protected, are. The new problem is this: Two parties each have a set of items which they wish to remain secret from the other. Each party is okay revealing part of its set, so long as it is not unique overall. Reveal $S \cap T$ but not $S \cap T'$ or $S' \cap T$. This is the two-party case. We will not further discuss a generalized version, but note [4]. Sometimes these requirements are tightened further to permit only one party of learning the result.

1.3 Applications

PSI has a number of useful applications where parties may mutually benefit on comparing potentially jointly-held yet sensitive information. These range from marketing techniques like measuring advertising success rates thought PSI [3] to enriching social media applications to improve user connectivity through private contact discovery. Private contact discovery in particular allows users of a service to find out which of their outside contacts (from their address book, typically with phone numbers) are also present on the same service. The assumption being that the service is more valuable when more of the user's contacts are using it as well.

2 Preliminaries

In order to understand how one might approach a PSI-type problem and begin design an acceptable protocol to solve it, we must first consider the following cryptographic primitives (building blocks) in our possession.

2.1 One-way functions

Using a hash function like SHA-256 can be a good option when trying to send a representative piece of data, or when trying to mask correlations from a set of values. Of course setting collisions or compression worries aside, this tool isn't completely misuse resistant and we'll look at an example of this later in §3.

2.2 Oblivious Transfer

1-out-of-2 Oblivious Transfer, $\binom{1}{2}$-OT, in its simplest form is allowing for a party P_1 with messages M_1 and M_2 to send M_c to a second party P_2 with select bit $c \in \{0, 1\}$ without P_1 ever learning anything about c and without P_2 ever being about to obtain both M_c *and* M_{1-c}. A notable generalization is $\binom{1}{n}$-OT where P_1 has n messages and $c \in \{1, ..., n\}$. Given an the ability to pick 1-in-2, we also have the ability to pick 1-in-n.

2.3 Pseudo-random Functions

[2] For this report consider pseudo-random function (PRF) to mean a deterministic function $f_k(x)$ keyed by a key $k \in \{0, 1\}^m$ with $x \in \{0, 1\}^n$ and indistinguishable from a truly random function $\pi : \{0, 1\}^n \to \{0, 1\}^n$ given π and f_k.

2.4 Oblivious Pseudo-random Functions

Oblivious pseudo-random functions (OPRFs) are the secure computation equivalent of pseudo-random functions. Given a PRF, $f_k(x) = y$ and two parties, P_1 who has a random key k, and P_2 who has an element x of the domain, the corresponding OPRF would return the output y to P_2 and leave P_1 to lean nothing [6].

3 Overview of PSI Strategies

This section hopes to give an high-level overview of some of the more common techniques used in practice to build a PSI scheme and discuss a little about each.

3.1 Hash Tables

Hash H every element and compare the hash digests out in the open.

$$X = \{x_i = H(s_i) : s_i \in S\}$$

$$Y = \{y_i = H(t_i) : t_i \in T\}$$

$$S \cap T \;\cong\; X \cap Y$$

If two elements fall into the same 'bin', they must be the same, right?

3.1.1 Discussion

Unfortunately this is idea overlooks a core property of all hash functions: determinism. There's no key, nonce, changing evaluation from each time on. Combined with the fact that many unique user attributes, such as phone numbers and names, have reasonably low entropy, there's nothing stopping a malicious party from manually (brute force / with rainbow tables) inverting your hash outputs.

3.2 Secret Equality using Garbled Circuits

Garbled circuits are a form of secure computation which lets parties compute a function securely (private inputs, public output) so long as the function is represented in circuit form with Boolean logic. Using a circuit that represents $f(x, y) \to x = y$ would allow any parties to compare pairwise and find the intersection this way.

3.2.1 Discussion

The author has not researched PSI circuit protocols in much detail, but is happy to see a simple solution with easy to check proof of security. Clearly if the protocol for garbled circuits is secure, the outputs will not reveal anything more than "Yes, there is a match", or "No, there isn't" from which is already the output we are okay with revealing as defined in §1.2. Unfortunately the run time of this solution is proportional to the size of both sets, and in the case where both are equal sizes, the number of comparisons is quadratic. Note that other circuit-based methods do exist. Ref [6] gives an example of how to do this by garbling an PRF to make an OPRF.

3.3 Oblivious PRF via Randomized OT

Instead of using garbled circuits to create and run an OPRF, this section describes how to build one from OT. Once the OPRF is all set up the two parties can run §5 from [5] as such:

$$f_k(s_i) = H(\bigoplus_{j=1}^{|S|} R^i_{s_i[j]})$$

$$X = \{f_k(s_i) : s_i \in S\}$$

$$Y = \{f_k(t_i) : t_i \in T\}$$

First P_1 computes θ random strings for each element of the longer set (so that $|R| = max(|S|, |T|)$), and sores it in R. R^i are random strings for s_i, and $R^i_{s_i[j]}$ is the random string selected by $s_i[j]$ (the jth bit of s_i) by preforming 1-in-2-OT where P_1 has R^i and P_2 has $c = s_i[j]$ as its select bit. After

transferring the random string for each bit of s_i, P_2 then XORs them all together and takes saves hash of this XOR product. P_2 repeats this $\forall s_i \in S$ to get X. P_1 follows the same process but, because P_1 created R it is already known and can compute Y without any need for more OT. The hash prevents a malicious P_2 from finding a correlation between XOR products. The intersection between X and Y matches that of S and T.

3.3.1 Discussion

This OT-based PSI protocol will not have as good of a communication cost as a garbled circuit-based solution might, as evident my the large number of OT uses (although I speculate their costs would overlap for small inputs). We do not give a formal proof of complexity. The security of the protocol follows from the security of OT and depends on P_1 being able to generate many random strings. Parties could improve the efficiency of the random OT by a factor of $|S|$ by using a single base/seed OT then "extended" as described popularly in [1, 5, 6].

References

[1] T. Chou and C. Orlandi. The simplest protocol for oblivious transfer. In *International Conference on Cryptology and Information Security in Latin America*, pages 40–58. Springer, 2015.

[2] O. Goldreich, S. Goldwasser, and S. Micali. How to construct randolli functions. In *Foundations of Computer Science, 1984. 25th Annual Symposium on*, pages 464–479. IEEE, 1984.

[3] M. Ion, B. Kreuter, E. Nergiz, S. Patel, S. Saxena, K. Seth, D. Shanahan, and M. Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. *IACR Cryptology ePrint Archive*, 2017:738, 2017.

[4] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1257–1272, 2017.

[5] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on {OT} extension. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 797–812, 2014.

[6] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on ot extension. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–35, 2018.

4 Pseudocode for Completed Version of §3.3

```
let l = HASH_LENGTH
let n = 16     // 1 in 16 OT and number base

// Server PSI code
server_psi(users)
    let size = await get('size')
    let random = Array(l*n).map(random_number)

    let y = []
    for i in [0, 1, ..., users.length)
        let hash = H(users[i])
        let s = []
        for j in [0, n, ..., random.length)  // loop l times
            let c = hash[j/n]  // selection index
            s[j/n] = random[j:j+n][c]

        y[i] = s[0]
        for j in [1, 2, ..., l)
            y[i] = y[i] xor s[j]

    y = y.map(H)
    give('y', y)

    for i in  [0, 1, ..., size)
        for j in [0, n, ..., random.length)  // loop l times
            OT.send(s_i[j/n], random.slice(j, j+n))


// Client PSI code
client_psi(contacts)
    give('size', contacts.length)

    let discovered = []
    let y = await get('y')

    for i in [0, 1, ..., hashes.length)
        let hash = H(contacts[i])
        let promise_s = []
        for j in [0, 1, ..., l)
            let c = hash[j]  // selection index
            s_i[j] = OT.receive(s_i[j], c, n)

        let x = s_i[0]
        for j in [1, 2, ..., l)
            x = x xor s_i[j]
        x = H(x)

        if x in y
            discovered.push(contacts[i])

    return discovered
```

4.1 JavaScript Demo

`https://github.com/wyatt-howe/private-contact-discovery-demo`
There is a functioning demo written in JavaScript at the link above. It features a dummy server (simulated in JavaScript), which you can register clients to to populate its internal set of users. There is a function similar to the ones in pseudocode above that will perform private contact discovery using the OT-based PSI method from section 3. Because of the line limit, I did *not* import any libraries. Instead, I wrote several dummy helper functions to simulate calling real cryptographically secure primitives. Do not focus on those, they are insecure purposely, in order to support my `server_psi` and `client_psi` methods which is the focus of my report, along with oblivious transfer. For OT I implemented the 1-in-2-OT protocol here[1]. My code does make use of 1-in-**n**-OT as an optimization, but that part itself is only secure against semi-honest adversaries because I decided it wasn't worth the time to implement any of the malicious 1-in-**n**-OT I read about, and instead I focused on the high-level PSI scheme. My discussion and analysis in §3 of OT-based PSI holds for my JavaScript implementation.