Implementation of Oblivious Bloom Intersection in Private Set Intersection Protocol (PSI)

L. Ertaul[1], A. M. Mehta[2], and T. K. Wu[2]
[1]Math & Computer Science, California State University, East Bay, Hayward, CA, USA
[2]Math & Computer Science, California State University, East Bay, Hayward, CA, USA

Abstract - *Today we are in the era of Big Data. The design of privacy preserving protocols in data processing is really challenging as the amount of data grows largely and complex. How to preserve privacy while meeting the requirements of speed and throughput have become critical criteria in the design. In this paper, we implement a practical use of Private Set Intersection (PSI) Protocol based on the new approach of oblivious Bloom intersection. The high scalability is achieved with parallel operations. We implemented the basic protocol and utilized Google Contact API to directly access the private contact information from two different Google accounts. The intersection of contact information could be found without disclosing any other private information from each account. We reported the result of the performance with respected to the number of contacts for different security levels. We only computed the intersections of two sets up to 25,000 contacts.*

Keywords: Privacy, Private Set Intersections, Privacy Preserving Protocols.

1 Introduction

Recent controversies about the leakage of documents revealing how big data can be fatal even though it creates tremendous opportunities for the world in field of medical research and national security [26]. Privacy issues and collection of consumer information have also been hot topics in the political circles around the world like the Prism program of the National Security Agency (NSA) under the guise of anti-terrorism [27]. Everyone has the right to privacy, but in the case of big data computation it's necessary to maintain data protection and privacy so that it cannot be misused. Using someone's information without their consent is unethical and we need high security. But if Big Data analytics leads to a terrorist suspect then in this case security of the society is counted much higher than an individual's security and privacy.

According to a study by Wikibon [28], shows that the market for Big Data will reach $50 billion mark in the next 5 years. According to results shown in Figure 1, in 2012 Big Data stood at just over a $5 billion in terms of services, hardware and software revenue. The awareness and the interest in Big Data have increased in the recent years. The power and the capability of Big Data to improve the efficiency of operations together with its influence in technological developments and services make Big Data's CAGR increase 58% from now and 2016.

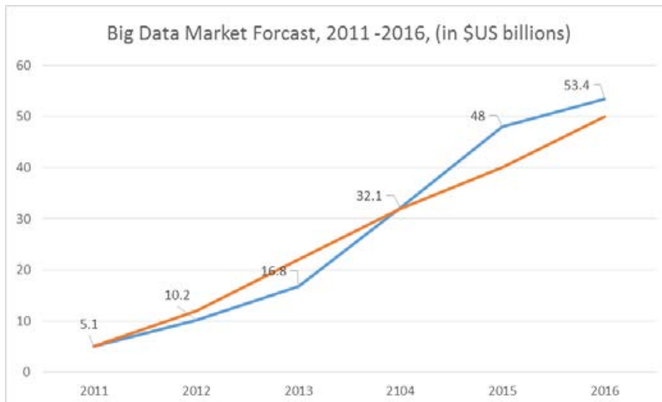

Figure 1 Big Data market Share

Privacy is often addressed as how the information in the application is kept secured and it's an essential issue with big data applications. Everyone has the right to be free from disturbances and intrusion in their respective personal life and also they are subject to right to privacy. Policy makers have therefore started addressing the most fundamental privacy laws, also "personally identifiable information" and role of consent were reviewed.

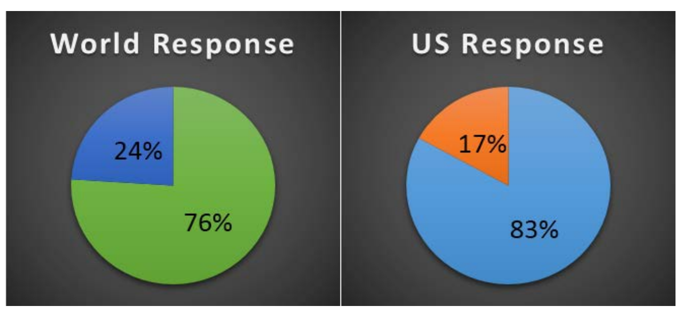

Figure 2 Privacy is the top preference according to World.

Figure 2 shows that trust plays a huge role for the success of Big Data. The survey was carried by Boston Consulting Group (BCG). The result of this survey shows that privacy is the most important preference for Big Data. Top issue according to 76% of consumers feel that the privacy is top issue with Big Data, but in the US 83% of the consumers feel the same. Big data allows organizations to boost their chances for success by enhancing customer service, manufacturing and other technological aspects. Privacy will create a trust which will help these organizations to benefit themselves and the consumers with Big Data capabilities.

In this paper, we first discuss on the problem of Private Set Intersection (PSI). The scenario is this. There are two parties, a client and a server, who want to compute and find out the intersection of their private inputs. At the end, client learns the intersection and the server learns nothing. The value in this study is that there are many practical applications, such as homeland security, two different law enforcement entities who want to compare their respective databases of suspects [8], detection of online game cheating [21], and find tax evaders [14]. To solve this kind of problem, many proposed PSI protocols are proposed, such as [3, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. However, the performance becomes an issue and unacceptable as the required security parameter and the size of the input data are getting big. Based on the result in [3], we found out Changyu Dong's protocol with oblivious Bloom intersection has the best performance comparing with other existing protocols, RSA-OPRF-based protocol by De Cristofaro et al [8] and the garbled circuit protocol by Huang et al [9]. The computational time of two million-element sets with 80 bit security for Dong's protocol needs only 41 seconds while De Cristofaro's protocol needs 10.6 minutes and Huang's protocols needs 27 hours [3].

Next, we implement the basic protocol, proposed by Dong, using the approach of oblivious Bloom intersection with actual private information from Google Contact. The reason we chose this protocol over other existing protocols is not only due to its efficiency and scalability, but also its simple operations. The computational, memory, and communication complexities are all linear in the size of the input sets [3]. Two Google Accounts are created, one as a server and the other one as a client. We first uploaded 25,000 contacts to each account and jointly compute the intersection of their private contact lists. At the end, client learns the intersection and the server learns nothing. The result shows that our implementation can compute the intersection of two 25,000 element sets from both Google Account efficiently.

The rest of the paper is organized as follows: In section 2, we present the definition of the key components of the basic protocol. In section 3, we will discuss the implementation of the basic protocol. In section 4, we evaluate the result.

2 The Basic Protocol

In this section, we review the flow and algorithms used in the basic protocol of PSI. The concept is actually simple. First, the client encodes its set C by computing a Bloom Filter (BF_c) and server encodes its set S by computing a Garbled Bloom Filter (GBF_s). By running an oblivious transfer (OT) protocol, the client receives a Garbled Bloom Filter representing the intersection while server learns nothing. At the end, the client uses it to query and obtain the intersection. Figure 3 illustrates the basic PSI protocol.

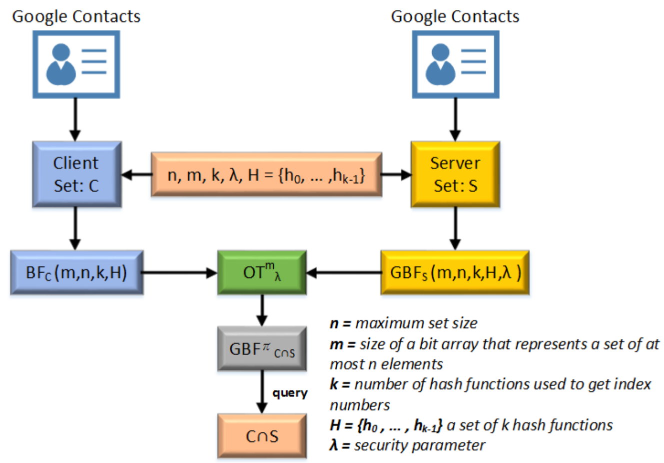

Figure 3: The basic PSI protocol

A. *Bloom Filter*

A Bloom Filter [1], designed by Burton H. Bloom in 1970, is probabilistic data structure that is used to test whether an element is present in a set in a rapid and memory-efficient way. A Bloom Filter has a base data structure of bit vector, an array of m bits that presents a set of S with n elements at most. A Bloom Filter uses a set of k independent hash functions H = $\{h_0, \ldots, h_{k-1}\}$. For each hash function h_i, the elements get mapped and uniformly distributed to the index numbers in the range of [0, m-1]. In this paper, we use $BF(m, n, k, H)$ to denote a Bloom Filter with the parameters of (m, n, k, H), use BF_s to denote the set S encoded by Bloom Filter, use $BF_s[i]$ to denote the bit at the index i in BF_s.

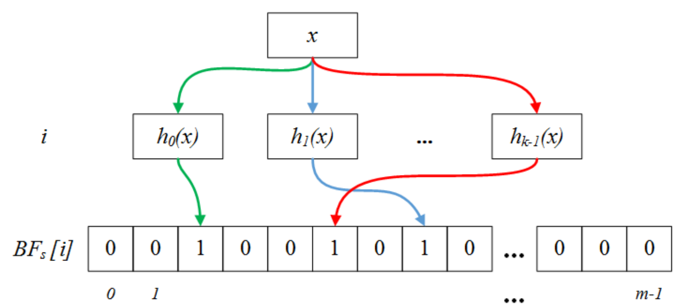

Figure 4: Add an element x to Bloom Filter

To create a Bloom Filter, as shown in Figure 4, for a set of S, all m bits in the array are first initialized to 0. Each element x that belongs to the set S is inserted into the filter by hashing x with k hash functions to get k index numbers and then setting all the bits at these indexes to 1, i.e. set $BF_s[h_i(x)] = 1$, where $0 \leq i \leq k$-1. We can also verify whether an element y is in the set S by hashing y with k hash functions to get k indexes and checking these indexes in the filter. If any of the bits at these index locations is 0, y is not in the set S. Otherwise, there is a probability that y is present in set S. Bloom Filter never yields a false negative due to the nature of hash functions being deterministic. However, it is possible to have false positive, which means y is not actually in set S while all $BF_s[h_i(x)]$ are set to be 1.

According to [2], the probability of a bit is still 0 in the Bloom Filter is

$$p' = (1 - 1/m)^{kn}$$

The probability of a certain bit is set to 1 is

$$p = 1 - p' = 1 - (1 - 1/m)^{kn} .$$

And the upper bound of the false positive probability is:

$$\epsilon = p^k \times (1 + O\left(\frac{k}{p}\sqrt{\frac{\ln(m) - k \times ln(p)}{m}}\right)) \quad (1)$$

which is negligible in k.

To be practical, it is necessary to build a Bloom Filter with a false positive probability that is capped. Based on [3], the efficiency of a Bloom Filter depends on the parameters of m and k. In our case, we assume that optimal m is used, which is $knlog_2e$ [3].

B. *Oblivious Transfer*

Oblivious Transfer (OT) [4] is a protocol that allows a sender to send part of its input to a receiver that protects both parties. The sender does not know which part of its input the receiver receives while the receiver does not know any information about other part of sender's input. A scenario that best explains the protocol is in the following: a server has a list of n strings $x_1 \ldots x_n$ and a client wants to learn x_i. The client does not want the server to know i and the server does not want the client knows x_j where j is not equal to i. The process of the server should transfer x_i to the client without knowing i is called oblivious transfer.

The operation of Oblivious Transfer protocols are actually costly and can be the bottleneck of efficiency in the design. However, Beaver has shown a solution to keep the oblivious transfer calls minimal [5]. In addition, efficient OT extensions were proposed in [6]. In our implementation, we kept the number of Oblivious Transfer calls at minimal.

C. *Google Contact API*

The Google Contact API v3 [7] allows client applications to request service and access to a user's contacts. These contacts are stored in user's Google account. However, the user account is limited to a maximum of 25,000 personal contacts and 128KB per contact [25].The requests to these private user data must be authorized by an authenticated user before the access is granted. Google uses OAuth 2.0 for this authorization process. By specifying the scope information and user's credential in the application, we can retrieve the contact list from the user's Google Account. The details of how to use the APIs are available at Google developers' website and Google's OAuth 2.0 Documentation [7].

D. *Garbled Bloom Filter*

A Garbled Bloom Filter [3], introduced by Dong, is a garbled version of a standard Bloom Filter. Essentially, there is no difference between a Garbled Bloom Filter and a Bloom Filter from high level point of view. In the creation of these filters, k uniform and independent hash functions are used to map each element into k index numbers. The corresponding array locations are set or checked for adding or querying an element respectively. What makes a Garbled Bloom Filter different than a standard Bloom Filter is the underlying data structure. To be specific, a Garbled Bloom Filter uses an array of λ-bit strings, where λ is a security parameter, and a standard Bloom Filter uses an array of bits.

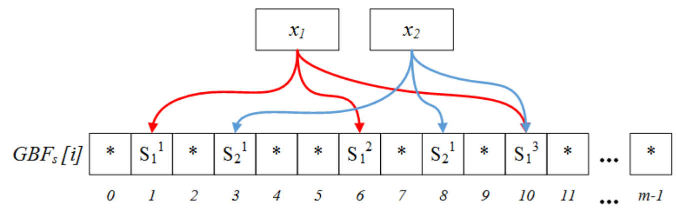

Figure 5: Add elements to Garbled Bloom Filter

Algorithms 1 and 2 [3] in the following are the pseudo codes for adding a set S into a Garbled Bloom Filter and for querying an element respectively.

<u>**Algorithm 1: BuildGBF(*S, n, m, k, H, λ*)E.**</u> **input**: a set *S, n, m, k, λ, H* = {*h*0, ...*h*k−1}
output: a *GBFs*(*m, n, k, H, λ*)
1 *GBFs* = new m-element array of bit strings;
2 **for** *i= 0* **to** m− 1 **do**
3 *GBFs[i]=NULL;*
4 **end**
5 **for** each x ∈ S **do**
6 emptySlot = −1, finalShare= x;
7 **for** *i=0* **to** *k-1* **do**
8 *j = hi(x);*
9 **if** *GBFs[j]==NULL* **then**
10 **if** *emptySlot* ==−1 then
11 emptySlot=*j*;
12 **else**
13 $GBFs[j] \overset{r}{\leftarrow} \{0,1\}^{\lambda}$;
14 finalShare=finalShare⨁*GBFs[j];*
15 **end**
16 **else**
17 finalShare=finalShare⨁*GBFs[j]*;
18 **end**
19 **end**
20 *GBFs[emptySlot]*=finalShare;
21 **end**
22 **for** *i= 0* **to** m− 1 **do**
23 **if** *GBFs[i]==NULL* **then**
24 $GBFs[i] \overset{r}{\leftarrow} \{0,1\}^{\lambda}$;
25 **end**
26 **end**

In Algorithm 1, first an empty Garbled Bloom Filter is created and initialized to NULL (line1-4). To add an element $x \in S$ into a Garbled Bloom Filter, the element gets spitted into k λ-bit shares using XOR-based Shamir's secret sharing scheme [20] and the shares gets stored in $GBF_s[h_i(x)]$ (line5-21). In this process, it might be possible that $j = h_i(x)$ has been occupied by a previously added element. For this scenario, the existing share stored at $GBF_s[j]$ is reused (line16-18) as shown in the Figure 5. The 3 shares of x_1, s_1^1, s_1^2, s_1^3 are added to the GBF_s first. Then the 3 shares of x_2 get added next. However, $GBF_s[10]$ has been occupied by s_1^3.

To prevent x_1 from becoming unrecoverable due to the replacement of s_1^3 with another string, it is reasonable to reuse the string s_1^3 as a share of x_2, where $x_2 = s_2^1 \oplus s_2^2 \oplus s_1^3$. After all the elements in S are added, the locations in filter that are still NULL will be filled with randomly generated λ-bit strings. According to [3], the reuse of shares will not cause security problems, and the probability of getting all shares of an element that is not in the intersection in this protocol is negligible. The detailed proofs and analysis are presented in [3].

Algorithm 2: QueryGBF(*GBFs, x, k, H*)

input : a GBF_S, an element $x, k, H = \{h_0, ...h_{k-1}\}$

output: True if $x \in S$, False otherwise

recovered = $\{0\}^\lambda$;
for $i=0$ **to** $k-1$ **do**
  $j = h_i(x)$;
  recovered = *recovered* $\oplus$ $GBF_S[j]$;
end
if *recovered* == x **then**
  return True;
else
  return False;
end

E. Produce an Intersection GBF

The idea of how to produce an intersection of Garbled Bloom Filter is based on performing the logic AND operation on two Bloom Filters. The resulting bits copied to a new filter that are set to 1 will be the intersection. The Algorithm 3 [3] in the following is the pseudo code used to build the intersection of Garbled Bloom Filter.

Algorithm 3: GBFIntersection(*GBFs, BFc, m*)

input: a $GBF_S(m, n, k, H, \lambda)$, a $BF_C(m, n, k, H)$, m

output: a $GBF_{C\cap S}(m, n, k, H, \lambda)$

$GBF_{C\cap S}$= new m-element array of bit strings;
for $i=0$ **to** $m-1$ **do**
  if $BF_C[i] == 1$ **then**
    $GBF_{C\cap S}[i] = GBF_S[i]$;
  else
    $GBF_{C\cap S}[i] \xleftarrow{r} \{0,1\}^\lambda$;
  end
end

If an element x is in $C \cap S$, we know that $BF_C[i]$ must be a 1 bit and $GBF_s[i]$ must be a share of x for each location i it hashes to. By running this algorithm, all elements in $C \cap S$ are preserved in a new Garbled Bloom Filter. The resulted intersection $C \cap S$ is called Oblivious Bloom Intersection as shown in Figure 3. The detailed proofs and analysis are presented in [3].

3 Implementation

Based on the result presented in [3], the approach of oblivious Bloom intersection is very promising and more scalable and efficient than other existing PSI protocols. Our initial plan is to implement the protocol on mobile phones for practical use. However, the computation requires large amount of memory resources. Due to the fact of limited resources mobile phones have, we decided to implement on laptops.

We have implemented the basic PSI protocol of Oblivious Bloom Intersection in conjunction with Google Contact API in Java. Currently the hash function we used to build and query Bloom Filters and Garbled Bloom Filters is SHA1 [22, 23, 24]. We registered two Google Accounts, one is used as client and the other one is as server. For the initial account setup, we uploaded 25,000 randomly generated contacts with phone numbers to each account and intentionally made 15 contacts commonly exist in both accounts. The purpose is to be able to verify result later. To access the contact information from Google Account, we use Google Contact API v3 libraries to call the Contact Service.

The detailed specification of the implementation is shown in the following table.

Table 1: Specification of Implementation

Platform	Intel® i5 Quad-Core 2.5Gz, 16GB RAM
Operating System	Windows 7
Programming Language	Java
Runtime Environment	JRE 7
Network Model	TCP/IP Client/Server Model
IDE	Eclipse
Crypto Library	Java.Security
Hash Algorithm	SHA-1
Key Size and Security Parameter	80, 128 bit
Mode	Single Threaded, Parallel Mode
Input Set	Google Contacts: two 25,000-element sets

4 Results and Evaluation

In this section, we show the performance result of our implementation with Google Contacts. Both client and server programs run on the same laptop with an Intel® i5 quad-core 2.5Gz, 16GB RAM, Windows 7 platform and are developed in Eclipse IDE with JDK 1.7.0.45. In our implementation, we set $k = \lambda$ to keep the false probability of a Bloom Filter to be as low as $2^{-\lambda}$ and set m to be optimal value $knlog_2e$. For example, at 80 bit security $k = \lambda = 80$, when n = 25,000, m = 2,885,390. We measured the total running time of the protocol that starts from the client sending request and ends when client output the intersection. The time of fetching the contacts from the Google Accounts and the time of setting up sockets are excluded.

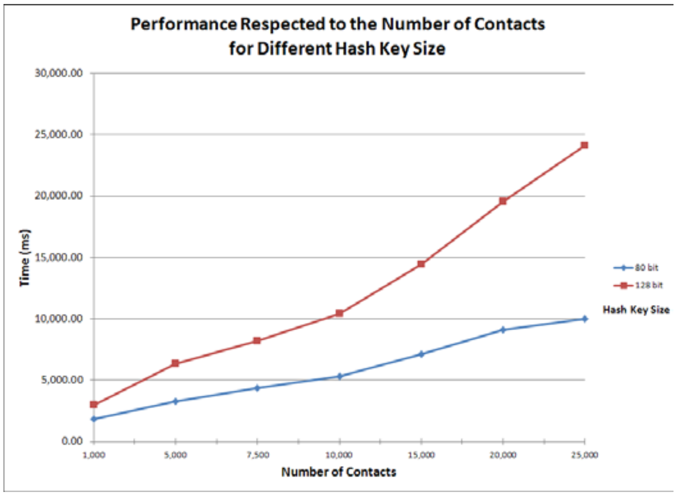

Figure 6: Performance of basic protocol respected to the number of contacts for different security key size

A. *Performance*

First, we show the performance in single threaded mode. We vary the size of contacts (n) from 1,000 to 25,000 and the security ($k = \lambda$) from 80 to 128 bit. The result is shown in Figure 6. As we can see, the running time increases almost linearly as the number of contacts increases at each level of security. For 25,000 contacts, it takes 10 seconds with 80 bit security and 24 seconds for 128 bit security.

Next, we show the comparison of performance between single-threaded and multi-threaded modes. We keep the key size to be 80 bit and vary the size of contacts (n) from 10,000 to 25,000. The result is shown in the Figure 7. The total running time in multi-threaded mode is significantly less than in single-threaded mode as the number of contacts increases. For 80 bit security and 25,000 contacts, it takes 10 seconds in single-threaded mode while it only takes 6.3 seconds in multi-threaded mode.

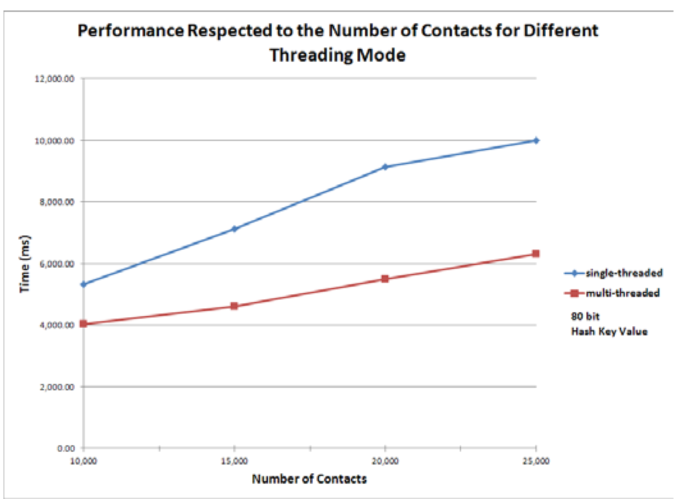

Figure 7: Performance of basic protocol respected to the number of contacts for different threading modes

In comparison to De Cristofarro's RSA-OPRF protocol and Huang's Sort-Compare-Shuffle with Waksman Network protocol that are previously the fastest PSI protocols, Dong [3] showed that the approach of oblivious Bloom intersection is in orders of magnitude faster than these protocols.

B. *Screenshot from Implementation*

The Figures 8 and 9 demonstrate the user interface of our implementation in Oblivious Bloom Intersection. The interaction between client and server can be easily observed. Here is the process of computing the intersection:

1. The server and client connect to its corresponding Google Account we set up initially and get initialized to run in the environment.
2. The server will generate the symmetric key and send to the client.
3. The client and server will each encode their data set to Bloom Filter and Garbled Bloom Filter respectively.
4. The client and server then perform oblivious transfer and server will generate a new Garbled Bloom Filter for intersection for the client
5. The client will use the new GBF to query and compute the intersection
6. At the end, we allow client to send the set back to the server for verification purpose.

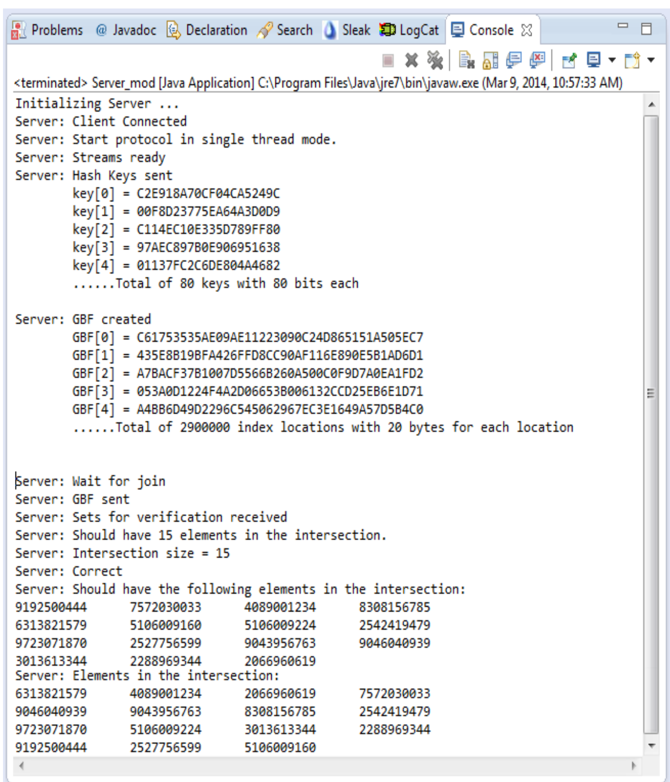

Figure 8: Interactive Server Interface

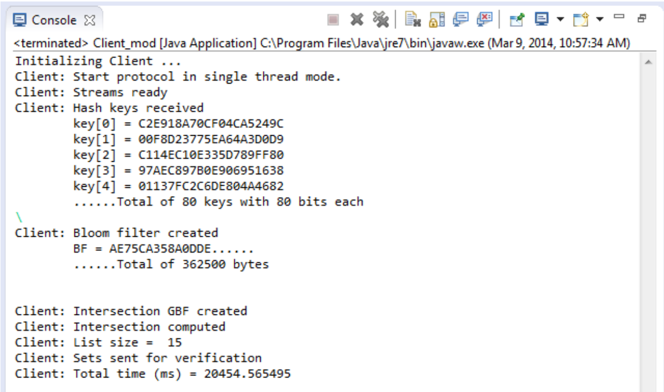

Figure 9: Interactive Client Interface

5 Conclusions

In this paper, we presented the practical use of a highly efficient and scalable PSI protocol based on the approach of oblivious Bloom intersection by implementing it in conjunction with Google Contacts. We also showed how this protocol can be easily integrated with cloud services like Google accounts to get contact information to be used as the input for both the client and the server. As explained by Dong, this protocol mainly depends on efficient symmetric key operations and these operations can be easily run in parallel. What makes the approach of oblivious Bloom intersection different than other protocols is mainly from its underlying data structure while other protocols are based on improving previous work with better algorithm. Its high performance is pretty encouraging and promising. In addition, it is suitable for large scale privacy preserving data processing. We hope that more applications can be developed with this protocol to provide secure and fast data processing

6 References

[1] B. H. Bloom. *Space/time trade-offs in hash coding with allowable errors.* Commun. ACM, 13(7):422–426, 1970.

[2] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid, and Y. Tang. *On the false-positive rate of bloom filters.* Inf. Process. Lett., 108(4):210–213, 2008.

[3] Changyu Dong, Liqun Chen, Zikai Wen, *When Private Set Intersection Meets Big Data: An Efficient and Scalable Protocol*, Page 4-15, 17-22. 2013

[4] M.O. Rabin. *How to exchange secrets by oblivious transfer*. Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[5] D. Beaver. *Correlated pseudorandomness and the complexity of private computations*. In STOC, pages 479–488, 1996.

[6] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. *Extending oblivious transfers efficiently.* In CRYPTO, pages 145–161, 2003.

[7] Google Contact API v3, Google.com. Retrieved Feb 18, 2014, from https://developers.google.com/google-apps/contacts/v3/

[8] E. D. Cristofaro and G. Tsudik. *Practical private set intersection protocols with linear complexity.* In Financial Cryptography, pages 143–159, 2010.

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

[10] M. J. Freedman, K. Nissim, and B. Pinkas. *Efficient private matching and set intersection.* In EUROCRYPT, pages 1–19, 2004.

[11] L. Kissner and D. X. Song. *Privacy-preserving set operations.* In CRYPTO, pages 241–257, 2005.

[12] J. Camenisch and G. M. Zaverucha. *Private intersection of certified sets.* In Financial Cryptography, pages 108–127, 2009.

[13] C. Hazay and Y. Lindell. *Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries.* In TCC, pages 155–175, 2008.

[14] E. D. Cristofaro, J. Kim, and G. Tsudik. *Linear-complexity private set intersection protocols secure in malicious model.* In ASIACRYPT, pages 213–231, 2010.

[15] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. *Efficient robust private set intersection*. In ACNS, pages 125–142, 2009.

[16] C. Hazay and K. Nissim. *Efficient set operations in the presence of malicious adversaries.* In Public Key Cryptography, pages 312–331, 2010.

[17] S. Jarecki and X. Liu. Fast secure computation of set intersection. In SCN, pages 418–435, 2010.

[18] S. Jarecki and X. Liu. *Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection.* In TCC, pages 577–594, 2009.

[19] G. Ateniese, E. D. Cristofaro, and G. Tsudik. *(if) size matters: Size-hiding private set intersection.* In Public Key Cryptography, pages 156–173, 2011.

[20] A. Shamir. *How to share a secret.* Commun. ACM, 22(11):612–613, 1979.

[21] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. *Openconflict: Preventing real time map hacks in online games*. In *IEEE Symposium on Security and Privacy*, pages 506–520, 2011.

[22] Florent Chabaud, Antoine Joux: *Differential Collisions in SHA-0*. CRYPTO 1998. pp56–71

[23] Eli Biham, Rafi Chen, *Near-Collisions of SHA-0, Cryptology ePrint Archive*, Report 2004/146, 2004 (appeared on CRYPTO 2004), IACR.org

[24] Xiaoyun Wang, Hongbo Yu and Yiqun Lisa Yin, *Efficient Collision Search Attacks on SHA-0*, CRYPTO 2005

[25] Google App Account Support, Google.com. Retrieved Feb 18, 2014, from https://support.google.com/a/answer/1146409?hl=en

[26] Rudarakanchana, Nat. "*Big Data: Cat-And-Mouse Escalates On Privacy Concerns*, As NRF Retail Conference Looms." International Business Times. Http://www.ibtimes.com/, 09 Jan. 2014. Web. 23 Feb. 2014.
[27] Bloomberg, Jason. "Big Data Governance: 5 Lessons Learned From PRISM." *Big Data Governance: 5 Lessons Learned From PRISM*. Http://www.baselinemag.com, 08 July 2013. Web. 23 Feb. 2014.
[28] Kelly, Jeff. "Big Data Market Size And Vendor Revenues - Wikibon." *Big Data Market Size And Vendor Revenues - Wikibon*. Wikibon, n.d. Web. 23 Feb. 2014.