When Private Set Intersection Meets Big Data: An Efficient and Scalable Protocol

Changyu Dong
Dept. of Computer and
Information Sciences
University of Strathclyde
Glasgow, UK
changyu.dong@strath.ac.uk

Liqun Chen
Hewlett-Packard Laboratories
Bristol, UK
liqun.chen@hp.com

Zikai Wen
Dept. of Computer and
Information Sciences
University of Strathclyde
Glasgow, UK
wjb12186@uni.strath.ac.uk

ABSTRACT

Large scale data processing brings new challenges to the design of privacy-preserving protocols: how to meet the increasing requirements of speed and throughput of modern applications, and how to scale up smoothly when data being protected is big. Efficiency and scalability become critical criteria for privacy preserving protocols in the age of Big Data. In this paper, we present a new Private Set Intersection (PSI) protocol that is extremely efficient and highly scalable compared with existing protocols. The protocol is based on a novel approach that we call *oblivious Bloom intersection*. It has linear complexity and relies mostly on efficient symmetric key operations. It has high scalability due to the fact that most operations can be parallelized easily. The protocol has two versions: a basic protocol and an enhanced protocol, the security of the two variants is analyzed and proved in the semi-honest model and the malicious model respectively. A prototype of the basic protocol has been built. We report the result of performance evaluation and compare it against the two previously fastest PSI protocols. Our protocol is orders of magnitude faster than these two protocols. To compute the intersection of two million-element sets, our protocol needs only 41 seconds (80-bit security) and 339 seconds (256-bit security) on moderate hardware in parallel mode.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Cryptographic controls*

Keywords

Private Set Intersection; Bloom Filters

1. INTRODUCTION

In many countries, protecting data privacy is no longer optional but a legal obligation. Legislation includes various US privacy laws (HIPAA, COPPA, GLB, FRC, etc.), European Union Data Protection Directive, and more specific national privacy regulations. It is a challenging task for organizations because they have to protect data in use and transmission. To this end, many security solutions have been proposed to enable privacy-preserving data processing. The amount of data to be processed and protected becomes increasingly large. For example, geneticists need to search 3 billion base pairs in personal genome to find genetic disorders that might cause diabetes or cancers, epidemiologists need to link multiple medical databases that contain millions of patients' records to identify risk factors for diseases, and online retailers want to correlate petabytes of their transaction records with customers' social network activities, hoping to increase customer satisfaction. Any privacy-preserving data processing service is not cost free and this has brought us new challenges: how to meet the increasing requirements of speed and throughput of modern applications, and how to scale up smoothly when data being protected is big? With the prevalence of large scale data processing, efficiency and scalability become critical criteria for designing a privacy-preserving protocol in the age of "Big Data".

The subject of study in this paper is the Private Set Intersection (PSI) problem. Namely, two parties, a client and a server, want to jointly compute the intersection of their private input sets in a manner that at the end the client learns the intersection and the server learns nothing. The PSI problem has been extensively studied for two reasons, firstly set intersection is a foundational primitive and secondly it has many practical applications. For example, PSI has been proposed as a building block in applications such as privacy preserving data mining [4], human genome research [6], homeland security [16], Botnet detection [33], social networks [32], location sharing [35] and cheater detection in online games [11]. Many PSI protocols have been proposed, e.g. [21, 30, 23, 13, 24, 27, 12, 16, 15, 28, 5, 25]. PSI protocols are often criticized as being impractical because the performance becomes unacceptable when the input size or the security parameter becomes large, and it is difficult to improve the performance by just adding hardware proportionally.

The criticism is not unfounded. Currently two protocols claim to be the fastest PSI protocol: the RSA-OPRF-based protocol by De Cristofaro et al [16, 17] and the garbled circuit protocol by Huang et al [25]. Both protocols have a highly optimized implementation. We obtained the source code from the authors of these two protocols and tested the performance. To compute the intersection of two 1,048,576-element (2^{20}) sets, De Cristofaro's protocol needs 10.6 minutes at 80-bit security, but requires a much longer time at 256-bit security. We estimate the time to be approximately 131 hours from tests with smaller sets. The tests with million-element sets on Huang's protocol were unsuccessful because the Java Virtual Machine ran out of memory on the client computer that has 16 GB RAM. From tests with smaller sets, we estimate that Huang's protocol requires 27 hours and 51 hours respectively to compute the intersection at 80-bit and 256-bit security. Clearly to use PSI in real world applications, we need more practical protocols.

Contributions We present a new PSI protocol that is much more

.

efficient than all the already existing PSI protocols. The protocol is designed based on a novel two-party computation approach, which makes use of a new variant of Bloom filters that we call *garbled Bloom filters*, and we refer the new approach as *oblivious Bloom intersection*. The ideas of garbled Bloom filters and oblivious Bloom intersection are general and have their own interests.

Our PSI protocol has two versions: a basic protocol, security of which can be proved in the semi-honest model, and an enhanced protocol, security of which can be proved in the malicious model. The basic protocol has linear complexity (with a small constant factor) and relies mostly on symmetric key operations. It is fast even with large input sets, and when the security parameter increases, the performance degrades gracefully. Test results show it is orders of magnitude faster than the previous best protocols. The enhanced protocol is an extension of the basic protocol, that only increases the cost by a factor proportional to the security parameter.

Apart from efficiency, another big advantage of the protocol is scalability: the computational, memory and communication complexities are all linear in the size of the input sets. More attractively, most operations in the protocol can be performed in the SPMD (single program, multiple data) fashion, which means little effort is required to separate the computation into a number of parallel tasks. Therefore it can fully take the advantage of parallel processing capacity provided by current multi-core CPUs, GPGPUs (General-purpose graphics processing unit) and cloud computing. As a result, the protocol is particularly suitable for Big Data oriented applications that have to process data in a parallelized and/or distributed way.

We have implemented a proof of concept prototype of the basic protocol. To compute the intersection of two million-element sets, it needs only 41 seconds (80-bit) and 5.65 minutes (256-bit) on two moderate computers in parallel mode.

Organization The paper is organized as follows: in section 2, we review the related work, in section 3 we introduce the notation and building blocks; in section 4, we present the garbled Bloom filter data structure, the semi-honest protocol, analyze the security and provide a simulation-based proof; in section 5 we show how to extend the basic protocol to achieve security against malicious adversaries; in section 6 we show a prototype of the basic protocol and the performance evaluation result; in section 7, we conclude the paper.

2. RELATED WORK

The concept and first protocol of Private Set Intersection were introduced by Freedman et al in [21]. Their protocol is based on oblivious polynomial evaluation. Along this line, Kissner and Song [30] proposed protocols in multiparty settings, Dachman-Soled et al [13], and Hazay and Nissim [24] proposed protocols which are more efficient in the presence of malicious adversaries. Hazey and Lindell [23] proposed another approach for PSI which is based on oblivious pseudorandom function (OPRF) evaluation. This approach is further improved by Jarecki and Liu [27, 28] and De Cristofaro et al [16, 15]. There are also a number of variants of PSI protocols, which aim to achieve more features than the original PSI concept. Camenisch and Zaverucha [12] proposed a PSI protocol which requires the input sets to be signed and certified by a trusted party, Ateniese et al [5] proposed a PSI protocol that also hides the size of the client's input set. Among the above protocols, the most efficient protocol is the protocol by De Cristofaro et al [16, 15]. It has linear complexity and requires $O(n)$ public key operations, where n is the size of the set. The performance of this protocol is affected significantly by n and the security parameter. Recently, Huang et al [25] presented a semi-honest PSI protocol based on garble circuits. This protocol requires $O(nlogn)$ symmetric key operations and a small number of public key operations. The authors demonstrated that in certain cases this protocol is significantly more efficient than the previous PSI protocols. At low security settings, De Cristofaro's protocol [16] is the fastest but at high security settings, Huang's protocol [25] is more efficient.

Recently a few PSI protocols based on Bloom filters were proposed. In [31], the parties AND their Bloom filters by a secure multiplication protocol and each party obtains an intersection Bloom filter. They then query the resulting Bloom filter to obtain the intersection. However the protocol is not secure because the intersection Bloom filter leaks information about other party's sets. In [29], Bloom filters are used in conjunction with the Goldwasser Micali homomorphic encryption.The semi-honest version of the protocol requires kn hash operations and $(k \log_2 e + kl + k + 2l)n$ modular multiplications, where k and l are parameters controlling false positive and e is the base of natural logarithms. Our basic protocol requires $2(k + k \log_2 e)n$ hash operations and a few hundred public key operations (independent to n). The total number of operations in our basic protocol is much less than the protocol in [29]. Given that a modular multiplication is faster than a public key operation but slower than a hash operation, for large input sets (i.e. a large value of n), the PSI scheme in [29] would be slower than our basic protocol. The protocol also has a higher communication overhead than ours, as each bit in the Bloom filter and the encrypted elements has to be expanded to a group element. The version secure in the malicious model requires a trusted party to certify the client's set, thus is hard to compare fairly with our enhanced protocol.

3. PRELIMINARIES

3.1 Notation

A function $\mu(\cdot)$ is *negligible in* n, or just *negligible*, if for every positive polynomial $p(\cdot)$ and any sufficiently large n it holds that $\mu(n) \leq 1/p(n)$. A *probability ensemble* indexed by I is a sequence of random variables indexed by a countable index set I. Namely, $X = \{X_i\}_{i \in I}$ where each X_i is a random variable. Two distribution ensembles $X = \{X_n\}_{n \in \mathbb{N}}$ and $Y = \{Y_n\}_{n \in \mathbb{N}}$ are *computationally indistinguishable*, denoted by $X \overset{c}{\equiv} Y$ if for every probabilistic polynomial-time (PPT) algorithm D, there exists a negligible function $\mu(\cdot)$ such that for every $n \in \mathbb{N}$,

$$|Pr[D(X_n, 1^n) = 1] - Pr[D(Y_n, 1^n) = 1]| \leq \mu(n)$$

For a set X, we denote by $x \overset{r}{\leftarrow} X$ the process of choosing an element x of X uniformly at random.

3.2 Bloom Filters

A Bloom filter [9] is a compact data structure for probabilistic set membership testing. A Bloom filter is an array of m bits that can represent a set S of at most n elements. A Bloom filter comes with a set of k independent uniform hash functions $H = \{h_0, ..., h_{k-1}\}$ such that each h_i maps elements to index numbers over the range $[0, m-1]$ uniformly. In the rest of the paper, we use (m, n, k, H)-Bloom filter to denote a Bloom filter parameterized by (m, n, k, H), use BF_S to denote a Bloom filter that encodes the set S, and use $BF_S[i]$ to denote the bit at index i in BF_S.

Initially, all bits in the array are set to 0. To insert an element $x \in S$ into the filter, the element is hashed using the k hash functions to get k index numbers. The bits at all these indexes in the bit array are set to 1, i.e. set $BF_S[h_i(x)] = 1$ for $0 \leq i \leq k-1$. To check if an item y is in S, y is hashed by the k hash functions, and all locations y hashes to are checked. If any of the bits at the locations is 0 , y is not in S, otherwise y is *probably* in S.

Because the hash functions are deterministic, if y is encoded in the filter then in the query phase every $BF_S[h_i(y)]$ must be 1, so a Bloom filter never yields a false negative. However, a false positive is possible, i.e. it is possible that y is not in the set S, but all $BF_S[h_i(y)]$ are set to 1. The probability that a particular bit in the Bloom filter is set to 1 is $p = 1 - (1 - 1/m)^{kn}$, and according to [10], the upper bound of the false positive probability is:

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

which is negligible in k.

In practice we often need to build a Bloom filter with a capped false positive probability, i.e. it represents any set of at most n elements from a universe in a manner that allows false positive probability to be at most ε. The efficiency of such a Bloom filter depends on the parameters m and k. It turns out the lower bound of m in this case is $m \geq n \log_2 e \cdot \log_2 1/\varepsilon$, where e is the base of natural logarithms. The optimal number of hash functions is $k = (m/n) \cdot \ln 2$ and if m is also optimal then the optimal k is $\log_2 1/\varepsilon$. In the rest of the paper, we always assume optimal k and m unless otherwise stated.

A standard Bloom filter trick is that if we have two (m, n, k, H)-Bloom filters that each encodes a set S_1 and S_2, we can obtain another (m, n, k, H)-Bloom filter $BF_{S_1 \cap S_2}$ by bit-wisely ANDing BF_{S_1} and BF_{S_2}. The resulting Bloom filter has no false negative, which means the query result of any element $y \in S_1 \cap S_2$ against $BF_{S_1 \cap S_2}$ is always true. The false positive probability of the resulting Bloom filter is no higher than either of the constituent Bloom filter [37]. Note that due to collisions, it is possible that the jth bit is set in BF_{S_1} by an element in $S_1 - S_1 \cap S_2$ and jth bit is set in BF_{S_2} by an element in $S_2 - S_1 \cap S_2$. Therefore the resulting Bloom filter usually contains more 1 bits than the Bloom filter built from scratch using $S_1 \cap S_2$.

3.3 Secret Sharing

Secret sharing is a fundamental cryptographic primitive. It allows a dealer to split a secret s into n shares such that the secret s can be recovered efficiently with any subset of t or more shares. With any subset of less than t shares, the secret is unrecoverable and the shares give no information about the secret. Such a system is called a (t, n)-secret sharing scheme. An example of such a scheme is Shamir's secret sharing scheme [40].

When $t = n$, an efficient and widely used secret sharing scheme can be obtained by simple $\oplus$ (XOR) operations [39]. The scheme works by generating $n - 1$ random bit strings $r_1, ..., r_{n-1}$ of the same length as the secret s, and computing $r_n = r_1 \oplus, ..., \oplus r_{n-1} \oplus s$. Each r_i is a share of the secret. It is easy to see that s can be recovered by computing $r_1 \oplus, ..., \oplus r_n$ and any subset of less than n shares reveals no information about the secret.

3.4 Oblivious Transfer

Oblivious transfer [38, 20] allows a sender to send part of its input to a receiver in a manner that protects both parties. Namely, the sender does not know which part the receiver receives and the receiver does not learn any information about the other part of the sender's input. Generally, an oblivious transfer protocol can be denoted as OT_l^m. The notation means the sender holds m pairs l-bit strings $(x_{j,0}, x_{j,1})$ $(0 \leq j \leq m - 1)$, while the receiver holds an m-bit selection string $r = (r_0, ..., r_{m-1})$. At the end of the protocol execution, the receiver outputs x_{j,r_j} for $0 \leq j \leq m - 1$.

Oblivious transfer protocols are costly and often become the efficiency bottleneck in protocol design. However it has been shown by Beaver that it is possible to obtain a large number oblivious transfers given only a small number of actual oblivious transfer calls [7]. In this direction, efficient *OT extensions* were proposed in [26]. The extensions rely on the Random Oracle Model [8] (or the existence of correlation robust hash functions) and can reduce OT_l^m to $\text{OT}_\lambda^\lambda$ where λ is a security parameter. The latter can be further reduced to λ invocations of OT_λ^1. In our implementation, we use the above OT extension scheme to reduce the actual cost of an OT_λ^m invocation to λ calls to the Naor-Pinkas OT protocol [34]. For the detail of the reduction, please consult [26].

3.5 The Semi-honest Model

We prove the security of the basic protocol in the presence of *static semi-honest* adversaries. In the model, the adversary controls one of the two parties and follows the protocol specification exactly. However, it may try to learn more information about the other party's input. The definitions and model are according to [22].

A two-party protocol π computes a function that maps a pair of inputs to a pair of outputs $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, where $f = (f_1, f_2)$. For every pair of inputs $x, y \in \{0,1\}^*$, the output-pair is a random variable $(f_1(x, y), f_2(x, y))$. The first party obtains $f_1(x, y)$ and the second party obtains $f_2(x, y)$. The function can be asymmetric such that only one party gets the result. It is captured as $f(x, y) \overset{def}{=} (f_1(x, y), \Lambda)$, where Λ denotes the empty string.

In the semi-honest model, a protocol π is secure if whatever can be computed by a party in the protocol can be obtained from its input and output only. This is formalized by the simulation paradigm. We require a party's *view* in a protocol execution to be simulatable given only its input and output. The view of the party i during an execution of π on (x, y) is denoted by $\textsf{view}_i^\pi(x, y)$ and equals $(w, r^i, m_1^i, ..., m_t^i)$ where $w \in (x, y)$ is the input of i, r^i is the outcome of i's internal random coin tosses and m_j^i represents the jth message that it received.

DEFINITION 1. *Let $f = (f_1, f_2)$ be a deterministic function. We say that the protocol π securely computes f in the presence of static semi-honest adversaries if there exists probabilistic polynomial-time algorithms S_1 and S_2 such that*

$$\{S_1(x, f_1(x, y))\}_{x,y} \overset{c}{\equiv} \{\textsf{view}_1^\pi(x, y)\}_{x,y}$$

$$\{S_2(y, f_2(x, y))\}_{x,y} \overset{c}{\equiv} \{\textsf{view}_2^\pi(x, y)\}_{x,y}$$

4. THE BASIC PROTOCOL

In this section we present the basic protocol that is secure in the semi-honest model. Conceptually the protocol is very simple: the client computes a Bloom filter that encodes its set C and the server computes a garbled Bloom filter (see below) that encodes its set S. Then they run an oblivious transfer protocol so that the client obtains a garbled Bloom filter that represents the intersection and the server learns nothing. Then the client queries the intersection garbled Bloom filter and obtains the intersection.

4.1 Garbled Bloom Filters

We introduce a new variant of Bloom filters called garbled Bloom filters (GBF). A garbled Bloom filter is the garbled version of a standard Bloom filter. From a high level point of view, there is no difference between a garbled Bloom filter and a Bloom filter: it encodes a set of at most n elements in an array of length m, it supports membership query with no false negative and negligible false positive. To add an element, the element is mapped by k independent uniform hash functions into k index numbers and the corresponding array locations are set. To query an element, the element is mapped by the same k hash functions into k index numbers and the corresponding array locations are checked.

From a low level point of view, a garbled Bloom filter is backed by a different data structure. Namely, instead of using an array of bits, a garbled Bloom filter uses an array of λ-bit strings, where λ is a security parameter. In the rest of the paper, we use (m, n, k, H, λ)-garbled Bloom filter to denote a garbled Bloom filter parameterized by (m, n, k, H, λ), we denote a garbled Bloom filter encoding a set S by GBF_S and denote the λ-bit string at index i by $GBF_S[i]$.

To add an element $x \in S$ to a garbled Bloom filter, we split the element into k λ-bit shares using the the XOR-based secret sharing scheme as described in section 3.3. The element is also mapped into k index numbers and we store one share in each location $h_i(x)$. Note this is a very loose description, the actual process is more complicated. To query an element y, we collect all bit strings at $h_i(y)$ and XOR them together. If the result is y then y is in S, otherwise y is not in S. The correctness is obvious: if $y \in S$, the XOR operation will recover y from its k shares which are retrievable from the garbled Bloom filter by their indexes. If $y \notin S$, then the probability of the XOR result is the same as y is negligible in λ. The algorithm to encode a set into a garbled Bloom filter and the algorithm to query an element are given in Algorithm 1 and 2.

Algorithm 1: $BuildGBF(S, n, m, k, H, \lambda)$

```
   input : A set $S, n, m, k, \lambda$, $H = \{h_0, ...h_{k-1}\}$
   output: An $(m, n, k, H, \lambda)$-garbled Bloom filter $GBF_S$
1  $GBF_S$= new $m$-element array of bit strings;
2  for $i = 0$ to $m - 1$ do
3      $GBF_S$[i]=NULL;    // NULL is the special symbol that means
       "no value"
4  end
5  for each $x \in S$ do
6      emptySlot = $-1$, finalShare= $x$;
7      for i=0 to k-1 do
8          $j = h_i(x)$;        // get an index by hashing the element
9          if $GBF_S[j]$==NULL then
10             if emptySlot ==$-1$ then
11                 emptySlot=j;        // reserve this location for
                   finalShare
12             else
13                 $GBF_S[j] \xleftarrow{r} \{0,1\}^\lambda$;  // generate a new share
14                 finalShare=finalShare$\oplus GBF_S[j]$;
15             end
16         else
17             finalShare=finalShare$\oplus GBF_S[j]$;     // reuse a
               share
18         end
19     end
20     $GBF_S[emptySlot]$=finalShare;    // store the last share
21 end
22 for $i = 0$ to $m - 1$ do
23     if $GBF_S[i]$==NULL then
24         $GBF_S[i] \xleftarrow{r} \{0,1\}^\lambda$;
25     end
26 end
```

In Algorithm 1, we first create an empty garbled Bloom filter and initialize each location to NULL (line 1-4). To add $x \in S$, we split x into k shares on the fly and store the shares in $GBF_S[h_i(x)]$ (line 5-21). Note that in this process, some location $j = h_i(x)$ may have been occupied by a previously added element. In this case we reuse the existing share stored at $GBF_S[j]$ (line 16-18). For example, in Figure 1 we first add x_1 to GBF_S and split it into 3 shares s_1^1, s_1^2, s_1^3. Then when we add x_2, $GBF_S[4]$ has already been occupied by s_1^2. So we reuse the string s_1^2 as a share of x_2, i.e. $x_2 = s_1^2 \oplus s_2^1 \oplus s_2^2$. This is because if we replace s_1^2 with another string, x_1 will not be recoverable in the query phase. Reusing shares will not cause security problems as far as the protocol concerns, we will show in Theorem 3 that the probability of getting all shares of an element that is not in the intersection in our protocol is negligible. After adding all elements in S, we generate and store random λ-bit strings at all locations that are still NULL (line 22-26). Algorithm 1 will succeed with an overwhelming probability, as stated in Theorem 1. When m and k are optimal, the success probability in Theorem 1 is approximately $1 - 2^{-k}$.

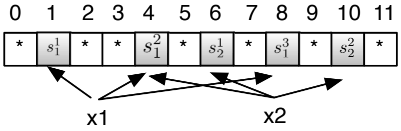


Figure 1: Add elements into a garbled Bloom filter

Algorithm 2: $QueryGBF(GBF_S, x, k, H)$

```
   input : A gabled Bloom filter $GBF_S$, an element $x$, $k$,
           $H = \{h_0, ...h_{k-1}\}$
   output: True if $x \in S$, False otherwise
1  $recovered = \{0\}^\lambda$;
2  for $i = 0$ to $k - 1$ do
3      $j = h_i(x)$;
4      $recovered = recovered \oplus GBF_S[j]$;
5  end
6  if $recovered == x$ then
7      return True;
8  else
9      return False;
10 end
```

THEOREM 1. *Algorithm 1 will succeed with a probability at least* $1 - p'^k \times (1 + O(\frac{k}{p'}\sqrt{\frac{\ln m - k \ln p'}{m}}))$ *where* $p' = 1 - (1 - 1/m)^{k(n-1)}$.

PROOF. Algorithm 1 fails when *emptySlot* remains -1 after the inner loop (line 20). This happens when adding an element to the GBF, all locations the element hashes to have been occupied by previously added elements. Because in this case, at most $n - 1$ elements have been added to the GBF, the probability of a particular position is occupied is at most $p' = 1 - (1 - 1/m)^{k(n-1)}$. The probability of all k locations have been occupied can be obtained in the same way as the false positive probability of an (m, n, k, H)-BF, which is at most $p'^k \times (1 + O(\frac{k}{p'}\sqrt{\frac{\ln m - k \ln p'}{m}}))$. The success probability is then 1 minus the probability of failure. □

In a garbled Bloom filter, each location is a λ-bit string that is either a share of certain elements or a random string. Analogously, a share in a gabled Bloom filter is equivalent to a "1" bit in a Bloom filter, and a random string is equivalent to a "0" bit. Same as the Bloom filters, there is no false negative when using a GBF because all shares of an encoded element are guaranteed to be retrievable and the XOR-based secret sharing scheme always produces the original element when all shares are available. When using a GBF, we need to consider and differentiate the following two probabilities:

- The collision probability of a GBF is the probability when y is not in S, but it hashes to the same set of index numbers as some $x \in S$. A collision does not cause false positive: the

$recovered$ string (Algorithm 2) is x but not y so the query result is still false. However it reveals x. The collision probability is negligible in k. Loosely, we can use the upper bound of the false positive probability of a Bloom filter as the upper bound of the collision probability of a garbled Bloom filter. Note that collisions do not affect the security of our protocol, but may be a concern if a GBF is used in other protocols.

- The false positive probability of GBF_S is the probability when y is not in S but the $recovered$ string equals y coincidentally. This probability is at most $2^{-\lambda}$.

More formally, we have the following theorem:

THEOREM 2. *Let GBF_S be an (m, n, k, H, λ)-garbled Bloom filter, (i) $\forall y \notin S, x \in S : Pr[(\bigoplus_{i=0}^{k-1} GBF_S[h_i(y)]) = x] \leq \epsilon$, where ϵ is the maximum false positive probability in equation (1). (ii) $\forall y \notin S : Pr[(\bigoplus_{i=0}^{k-1} GBF_S[h_i(y)]) = y] \leq 2^{-\lambda}$.*

PROOF. We start from the collision probability. Let BF_S be the (m, n, k, H)-Bloom filter that encodes the same set S as GBF_S. Now for any $y \notin S$, we query y against both GBF_S and BF_S. Whenever the GBF query results in a collision, the Bloom filter query must return a false positive. This is because by definition, y hashes to the same set of index numbers as some $x \in S$, so all locations are set to 1 in BF_S by x, therefore the Bloom filter query returns true, but $y \notin S$ so this is a false positive. Since a GBF collision implies a Bloom filter false positive, the collision probability is bounded by the false positive probability of the Bloom filter.

Let's consider the false positive probability of a GBF. A false positive occurs when y is not in S but the *recovered* string equals y. The *recovered* string is $GBF_S[h_0(y)] \oplus \ldots \oplus GBF_S[h_{k-1}(y)]$. Each constitution string $GBF_S[h_i(y)]$ is either a share of certain elements or a random string. When $y \notin S$, there are three cases:

Case 1: All constitution strings are shares of the same element in S. We denote the probability of this case as p_1. In this case for sure $recovered \neq y$ because $y \notin S$.

Case 2: The constitution strings are shares of several elements in S. We denote the probability of this case as p_2. In this case we can divide the constitution strings into several groups of size at most $k-1$, each group contains the shares of a particular element. From the security of the XOR-based secret sharing scheme, the XOR result of each group should be a uniformly random string. Therefore the $recovered$ string is a uniformly random string.

Case 3: At least one of the constitution strings is a random string. The probability of this case as $p_3 = 1 - p_1 - p_2$. In this case the $recovered$ string is also a uniformly random string.

In all three cases, a false positive occurs if $recovered = y$. In case 1, the false positive probability is 0. In the other two cases, the false positive probability is $2^{-\lambda}$. Let B denote the event that a false positive occurs, and let a_1, a_2, a_3 denote the events that case 1, case 2, case 3 occurs respectively, by the law of total probability, the false positive probability is:

$$\begin{aligned} Pr[B] &= Pr[a_1]Pr[B|a_1] + Pr[a_2]Pr[B|a_2] + Pr[a_3]Pr[B|a_3] \\ &= 0 \cdot p_1 + 2^{-\lambda} \cdot p_2 + 2^{-\lambda} \cdot p_3 \\ &= 2^{-\lambda}(1 - p_1) \leq 2^{-\lambda} \end{aligned}$$

□

In summary, with proper parameters, a garbled Bloom filter exhibits similar properties when encoding set membership: no false negative and negligible false positive.

4.2 Produce an Intersection GBF

In this section we show how to produce an intersection garbled Bloom filter from an (m, n, k, H, λ)-garbled Bloom filter and an (m, n, k, H)-Bloom filter. The idea is quite similar to creating an intersection Bloom filter by ANDing two Bloom filters.

Let's say we have an (m, n, k, H)-Bloom filter BF_C that encodes a set C and an (m, n, k, H, λ)-garbled Bloom filter GBF_S that encodes a set S. We use Algorithm 3 to build the intersection garbled Bloom filter $GBF_{C \cap S}$.

```
Algorithm 3: $GBFIntersection(GBF_S, BF_C, m)$
input : An $(m, n, k, H, \lambda)$-garbled Bloom filter $GBF_S$, an
        $(m, n, k, H)$-Bloom filter $BF_C$, $m$
output: An $(m, n, k, H, \lambda)$-garbled Bloom filter $GBF_{C \cap S}$
$GBF_{C \cap S}$= 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
```

The intuition of the algorithm is this: if an element x is in $C \cap S$, then for every position i it hashes to, $BF_C[i]$ must be a 1 bit and $GBF_S[i]$ must be a share of x. Therefore by running the algorithm, all shares of x are copied to the new garbled Bloom filter. That is, all elements in $C \cap S$ are preserved in the new garbled Bloom filter. On the other hand, if x is not in $C \cap S$, then with a high probability, at least one share will not be copied. Or in other words, elements not in $C \cap S$ are eliminated from the new garbled Bloom filter. Thus the new garbled Bloom filter is indeed a garbled Bloom filter that encodes the intersection. Formally, we have the following theorem:

THEOREM 3. *Let $GBF_{C \cap S}$ be an (m, n, k, H, λ)-garbled Bloom filter produced in Algorithm 3. For $0 \leq i \leq k - 1$, let a_i be the event that $GBF_{C \cap S}[h_i(x)]$ equals the ith share of x, we have (i) $\forall x \in C \cap S$: $Pr[a_0 \wedge \ldots \wedge a_{k-1}] = 1$, (ii) $\forall x \notin C \cap S$: $Pr[a_0 \wedge \ldots \wedge a_{k-1}]$ is negligible in k.*

PROOF. The first part: we can see from the algorithm that for any element $x \in C \cap S$, all the shares will be copied from GBF_S to $GBF_{C \cap S}$ because the corresponding locations in BF_C are all set to 1.

The second part: Firstly, $GBF_{C \cap S}$ does not encode any element $x \notin S$ because GBF_S contains no share of any element $x \notin S$. Secondly, for any element $x \in S - C \cap S$, the probability of all its shares are copied from GBF_S to $GBF_{C \cap S}$ is ϵ, where ϵ is the upper bound of the false positive probability of an (m, n, k, H)-BF. This is because if all shares of x are copied to $GBF_{C \cap S}$ then it means all locations that x hashes to in BF_C are set to 1. However $x \notin C \cap S$ and consequently $x \notin C$, then it implies a false positive when we query x against BF_C and the probability is ϵ. □

From security point of view, a more interesting property of the intersection GBF is that it is indistinguishable from a GBF built from scratch that encodes $C \cap S$.

THEOREM 4. *Given sets C, S and their intersection $C \cap S$, let $GBF_{C \cap S}$ be an (m, n, k, H, λ)-garbled Bloom filter produced by Algorithm 3 from GBF_S and BF_C, let $GBF'_{C \cap S}$ be another (m, n, k, H, λ)-garbled Bloom filter produced by Algorithm 1 using $C \cap S$, we have $GBF_{C \cap S} \overset{c}{\equiv} GBF'_{C \cap S}$.*

PROOF. Given $GBF_{C \cap S}$, we modify it to get $GBF''_{C \cap S}$. We scan $GBF_{C \cap S}$ from the beginning to the end and for each location i, we modify $GBF_{C \cap S}[i]$ using the following procedure:

1. If $GBF_{C\cap S}[i]$ is a share of an element in $C \cap S$, then do nothing.
2. Else if $GBF_{C\cap S}[i]$ is a random string, do nothing.
3. Else if $GBF_{C\cap S}[i]$ is a share of an element in $S - C \cap S$, replace it with a uniformly random λ-bit string.

The result is $GBF''_{C\cap S}$. Every $GBF_{C\cap S}[i]$ must fall into one of these three cases, so there is no unhandled case.

Now we argue that the distribution of $GBF''_{C\cap S}$ is identical to $GBF'_{C\cap S}$. To see that, let's compare each location in $GBF''_{C\cap S}$ and $GBF'_{C\cap S}$. From Algorithm 1 and the above procedure, we can see that $GBF''_{C\cap S}$ and $GBF'_{C\cap S}$ contain only shares of elements in $C \cap S$ and random strings. Because $GBF''_{C\cap S}$ and $GBF'_{C\cap S}$ use the same set of hash functions, for each $0 \leq i \leq m-1$, $GBF''_{C\cap S}[i]$ is a share of an element in $C \cap S$ iff $GBF'_{C\cap S}[i]$ is a share of the same element; $GBF''_{C\cap S}[i]$ is a random string iff $GBF'_{C\cap S}[i]$ is a random string. The distribution of a share depends only on the element and the random strings are uniformly distributed. So the distribution of every location in $GBF''_{C\cap S}$ and $GBF'_{C\cap S}$ are identical therefore the distributions of $GBF''_{C\cap S}$ and $GBF'_{C\cap S}$ are identical.

Then we argue that the distribution of $GBF''_{C\cap S}$ is identical to $GBF_{C\cap S}$ except for a negligible probability η.

Case 1, $GBF_{C\cap S}$ encodes at least one elements in $S - C \cap S$. In this case the distribution of $GBF''_{C\cap S}$ differs from the distribution of $GBF_{C\cap S}$. From Theorem 3, the probability of each element in $S - C \cap S$ being encoded in $GBF_{C\cap S}$ is ϵ. Since there are $d = |S| - |C \cap S|$ elements in $S - C \cap S$, the probability of at least one element is falsely contained in $GBF_{C\cap S}$ is:

$$\eta = \sum_{i=1}^{d} \binom{d}{i} \cdot \epsilon^i = \sum_{i=1}^{d} \frac{d(d-1)...(d-i+1)}{i(i-1)...1} \cdot \epsilon^i \leq \sum_{i=1}^{d} (d\epsilon)^i \leq 2d\epsilon$$

As we can see η is negligible if ϵ is negligible.

Case 2: $GBF_{C\cap S}$ encodes only elements from $C \cap S$. In this case, each element in $S - C\cap S$ may leave up to $k-1$ shares in $GBF_{C\cap S}$. The only difference between $GBF_{C\cap S}$ and $GBF''_{C\cap S}$ is that in $GBF''_{C\cap S}$, all "residue" shares of elements in $S - C \cap S$ are replaced by random strings. From the security of the XOR-based secret sharing scheme, the residue shares should be uniformly random (otherwise they leak information about the elements). Thus the procedure does not change the distribution when modifying $GBF_{C\cap S}$ into $GBF''_{C\cap S}$. So the distributions of $GBF_{C\cap S}$ and $GBF''_{C\cap S}$ are identical. The probability of this case is at least $1-\eta$.

Since $GBF''_{C\cap S} \equiv GBF'_{C\cap S}$ always holds and $GBF_{C\cap S} \equiv GBF''_{C\cap S}$ holds in case 2, we can conclude that $Pr[GBF_{C\cap S} \equiv GBF'_{C\cap S}] \geq 1 - \eta$ thus

$|Pr[D(GBF_{C\cap S}) = 1] - Pr[D(GBF'_{C\cap S}) = 1]| \leq \eta$ □

Theorem 4 shows that the probability of $GBF_{C\cap S}$ and $GBF'_{C\cap S}$ are distinguishable is η. In our implementation we set $k = \lambda$ so ϵ is about $2^{-\lambda}$, then a question may arise whether this is appropriate: since η is bounded by $2d\epsilon$, will the security be weakened? For example if $\lambda = 80$ and $d = 2^{20}$, will the security be weakened to about 60-bit rather then desired 80-bit? The answer is no. Loosely speaking, a bigger d means that an adversary can distinguish $GBF_{C\cap S}$ and $GBF'_{C\cap S}$ with a smaller number of attempts, but in each attempt the amount of computation required to distinguish the two also increases. Therefore the total amount of work needed to distinguish the two remains unchanged. We demonstrate it through the following game: an adversary can query an oracle with two sets S and C of its choice. The oracle randomly chooses $b \xleftarrow{r} \{0, 1\}$, if $b = 1$, it returns $GBF_{C\cap S}$, if $b = 0$, it returns $GBF'_{C\cap S}$.The adversary can repeatedly query the oracle. At the end of the game, it challenges the oracle and outputs b'. It wins the game if $b' = b$. The advantage is $|Pr[b' = b] - \frac{1}{2}|$. As we show in Theorem 5, the advantage depends only on ϵ, not η.

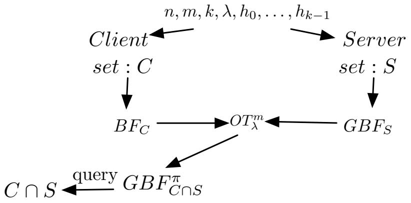


Figure 2: The basic PSI protocol $\pi_\cap$

THEOREM 5. *For an adversary runs in time t, the adversary's advantage in the above game is no more than $O(t) \cdot \epsilon$.*

PROOF. In each oracle query, the adversary has a probability of η to distinguish $GBF_{C\cap S}$ and $GBF'_{C\cap S}$. Therefore if it makes q oracle queries, the advantage will be $q \cdot \eta$. The number of oracle queries the adversary can make is t/t_d, where t_d is the time needed to check whether the GBF encodes an element that is not in the intersection. As there is no way other than querying the GBF to decide, the best the adversary can do is to query all elements in $S - C\cap S$ against the GBF. Therefore $t_d = |S - C\cap S| \cdot t_g = d \cdot t_g$, where t_g is the time of a GBF query. Therefore the advantage of the adversary is: $q \cdot \eta = \frac{t}{t_d} \cdot \eta \leq \frac{t}{d \cdot t_g} \cdot 2d\epsilon = O(t) \cdot \epsilon$. □

4.3 Oblivious Bloom Intersection

The idea of the basic protocol is shown in Figure 2. That is, to run Algorithm 3 by two parties using oblivious transfer. Thus we call it oblivious Bloom intersection. The protocol runs as follows:

1. The server's private input is S, and the client's private input is C. The auxiliary inputs include the security parameter λ, the maximum set size n, the optimal Bloom filter parameters m, k and $H = \{h_0, ..., h_{k-1}\}$. The parameter k is set to be the same as the security parameter λ.
2. The client generates an (m, n, k, H)-BF that encodes its private set C, the server generates an (m, n, k, H, λ)-GBF that encodes its private set S. The client uses its Bloom filter as the selection string and acts as the receiver in an OT^m_λ protocol. The server acts as the sender in the OT protocol to send m pair of λ-bit strings $(x_{i,0}, x_{i,1})$ where $x_{i,0}$ is a uniformly random string and $x_{i,1}$ is $GBF_S[i]$. For $0 \leq i \leq m-1$, if $BF_C[i]$ is 0, then the client receives a random string, if $BF_C[i]$ is 1 it receives $GBF_S[i]$. The result is $GBF^\pi_{C\cap S}$.
3. The client computes the intersection by querying all elements in its set against $GBF^\pi_{C\cap S}$.

At the end of step 2, the client receives a new garbled Bloom filter $GBF^\pi_{C\cap S}$. The OT protocol does exactly what we want to achieve in Algorithm 3.

THEOREM 6. *Given an (m, n, k, H, λ)-Garbled Bloom filter GBF_S and an (m, n, k, H)-Bloom filter BF_C. the garbled Bloom filter $GBF^\pi_{C\cap S}$ is equivalent to a garbled Bloom filter $GBF_{C\cap S}$ that is built by Algorithm 3 using GBF_S and BF_C .*

PROOF. Let's run the algorithm and protocol simultaneously and use the same random coins for the random strings that are to be placed in $GBF^\pi_{C\cap S}$ and $GBF_{C\cap S}$. From the description of the algorithm and the protocol, we can see that for $0 \leq i \leq m-1$,

	PK ops	SK ops	Memory	Comm.
Huang's	$O(\lambda)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
De Cristofaro's	$O(n)$	$O(n)$	$O(n)$	$O(n)$
The Basic Protocol	$O(\lambda)$	$O(n)$	$O(n)$	$O(n)$

Table 1: Asymptotic Costs Comparison: n is size of the input sets, λ is the security parameter, PK (SK) ops means public (symmetric) key operations.

if $BF_C[i] = 1$, then $GBF^{\pi}_{C\cap S}[i] = GBF_{C\cap S}[i] = GBF_S[i]$; if $BF_C[i] = 0$, then $GBF^{\pi}_{C\cap S}[i] = GBF_{C\cap S}[i] = r_i$ where r_i is a uniformly random strings. Therefore the two garbled Bloom filters are equivalent. □

Informally, the correctness of the protocol follows from Theorem 3 and 6. The protocol produces a garbled Bloom filter that encodes $C \cap S$, then by querying it the client can obtain the correct intersection except for a negligible probability. To see why the protocol is secure, notice that the only messages being sent in the protocol are the messages in the OT protocol. The client's privacy is protected because the server learns no information about BF_C in the OT execution. The server's privacy is protected because the client receives only $GBF^{\pi}_{C\cap S}$ from the server and it contains only information about elements in $C \cap S$.

The reader may have noticed that the OT protocol can also be used to AND two Bloom filters in a similar way and create an intersection Bloom filter $BF_{C\cap S}$ on the client side. Then do we really need the garbled Bloom filter? Can the server just encode its set into a Bloom filter and run the protocol? The quick answer is we do need the garbled Bloom filter. $BF_{C\cap S}$ leaks information about the server's set because it contains more 1 bits than the Bloom filter built from scratch using $C \cap S$. The expected number of additional 1 bits is $\frac{(t_S - t_\cap)(t_C - t_\cap)}{m - t_\cap}$, where $t_S, t_C, t_\cap$ are the number of 1 bits in BF_S, BF_C and the the Bloom filter built from scratch using $C \cap S$ respectively [37]. The additional knowledge the client gets is the additional 1 bits in $BF_{C\cap S}$.

The protocol makes a single call to OT^m_λ, so the efficiency depends largely on the efficiency of the underlying OT protocol. If we use the semi-honest OT extension protocol from [26] and the Naor-Pinkas OT [34], then:

Computational complexity: To build BF_C or GBF_S, each party needs $k \cdot n$ hash operations. Then the server needs λ public key operations and the client need 2λ public key operations for the Naor-Pinkas OT, and both parties need $m = kn \log_2 e \approx 1.44kn$ hash operations for the OT extension.

Memory complexity: The client needs to keep a copy of the Bloom filter and a copy of the intersection Garbled Bloom filter which in total need at most $(\lambda + 1)m$ bits. This can be optimized to $(\lambda/2 + 1)m$ bits because the client can throw away the string received when $BF_C[i] = 0$ and leave $GBF^{\pi}_{C\cap S}[i] = NULL$. The server needs to store the garbled Bloom filter that is $\lambda \cdot m$ bits.

Communication complexity: The main data sent in the protocol is a bit matrix required by the OT extension and the strings sent by the server in the OT extension. In total $2\lambda \cdot m$ bits. All other communication costs are much less significant and can be ignored.

A quick asymptotic costs comparison of Huang's, De Cristofaro's and our basic protocol is shown in Table 1.

4.4 Security Analysis

Now we sketch the security proof of the basic protocol. The basic protocol is secure in the semi-honest model. The main theorem is stated below:

THEOREM 7. *Let C, S be two sets from a predefined universe, $f_\cap$ be the set intersection function defined as:*

$$f_\cap(C, S) = (f_C(C, S), f_S(C, S)) = (C \cap S, \Lambda).$$

Assuming the underlying OT^m_λ protocol is secure, then the basic PSI protocol $\pi_\cap$ in Section 4.3 securely computes $f_\cap$ in the presence of semi-honest adversaries.

PROOF. (sketch) If the OT^m_λ is secure then the simulators for the sender and receiver are guaranteed to exist, we can use them as subroutines when constructing our simulators.

Server's view We start from the case in which the server is corrupted. We construct a simulator Sim_S that receives the server's private input and output and generates the view of the server in the protocol. Given S, the simulator Sim_S uniformly chooses its random coins r^s and generates the garbled Bloom filter GBF_S that encodes its set S. Then Sim_S invokes the simulator of the OT sender Sim^{OT}_{snd} that is guaranteed to exist. Sim_S obtains Sim^{OT}_{snd}'s view for the OT protocol. Finally Sim_S outputs the simulated view: $(S, r^s, \mathsf{Sim}^{OT}_{snd}(GBF_S, \Lambda))$. We then need to show that the view is indistinguishable from a view in an execution of $\pi_\cap$. A view of the real protocol execution contains the input S, the random coins and the messages in the OT protocol. In the simulated view, the input set S is the same as in the view of a real execution, the outcome of internal random coins r^s is uniformly random thus the distribution is the same as in a real execution. As the OT protocol is secure, then the distribution of the view produced by $\mathsf{Sim}^{OT}_{snd}(GBF_S, \Lambda)$ should be indistinguishable from the view in a real execution of the OT protocol. Thus we conclude the simulated view is indistinguishable from a real view.

Client's view We construct a simulator Sim_C that is given the client's private input C and the output $C \cap S$. Sim_C chooses its random coins r^c. It then generates the Bloom filter BF_C to encode its set and the garbled Bloom filter $GBF_{C\cap S}$ from scratch using Algorithm 1. It then invokes the simulator of the OT receiver Sim^{OT}_{rec} with BF_C and $GBF_{C\cap S}$. Sim_C obtains the view for the OT protocol. Finally Sim_C outputs the simulated view: $(C, r^c, GBF_{C\cap S}, \mathsf{Sim}^{OT}_{rec}(BF_C, GBF_{C\cap S}))$. The view of a real protocol execution contains the input set C, the random coins, the garbled Bloom filter $GBF^{\pi}_{C\cap S}$, and the messages in the OT protocol. In the simulated view, the input set C and r^c should be indistinguishable from the counter parts in the real view. The garbled Bloom filter $GBF_{C\cap S}$ is indistinguishable from $GBF^{\pi}_{C\cap S}$ as we have shown in Theorem 4 and 6. The rest parts in the views are the simulated OT messages and the OT messages in the real execution. As the OT protocol is secure, then they should be indistinguishable. Thus we conclude the simulated view is indistinguishable from a real view.

Combine the above, we conclude that:

$$\{\mathsf{Sim}_S(S, f_S(C, S)\}_{C,S} \overset{c}{\equiv} \{\mathsf{view}^{\pi}_S(C, S)\}_{C,S}$$
$$\{\mathsf{Sim}_C(C, f_C(C, S))\}_{C,S} \overset{c}{\equiv} \{\mathsf{view}^{\pi}_C(C, S)\}_{C,S}$$

and finish our proof. □

5. THE ENHANCED PROTOCOL

In this section, we present a fully secure PSI protocol whose security holds in the presence of malicious parties. The protocol is shown in Figure 3. The security model and proof can be found in the full version [19].

In the basic protocol, the interaction between the two parties is essentially an oblivious transfer. At the first glance, it seems that we can easily obtain a fully secure protocol by replacing the semi-honest OT protocol with one that is secure against malicious parties. However, this is not enough. A fully secure OT protocol can prevent malicious behaviors such as changing input during the protocol execution but it cannot prevent a malicious client from mounting a full universe attack.

Server's input: Set S
Client's input: Set C
Auxiliary input: the security parameter λ , parameters for BF and GBF $n, k = \lambda, m = 2kn, H = \{h_0, \dots, h_{k-1}\}$, a secure block cipher E.

1. The client generates a Bloom filter BF_C. The client then generates m λ-bit random strings, say $r_0, ...r_{m-1}$. The client sends the random strings to the server.
2. The server generates the garbled Bloom filter GBF_S. The server generates a random key sk for the block cipher E. For $0 \leq i \leq m-1$, the server computes $c_i = E(sk, r_i||GBF_S[i])$. The server also uses a $(m/2, m)$-secret sharing scheme to split sk into m shares $(t_0, ..., t_{m-1})$.
3. The server and the client engage in an OT protocol that is secure against malicious parties. The client uses BF_C as the selection string and the server uses as input two sets of strings c_i and t_i $(0 \leq i \leq m-1)$. As a result of the protocol, if $BF_C[i] = 1$, the client receives c_i; if $BF_C[i] = 0$,the client receives t_i.
4. The client recovers sk from the shares it received in the OT. The client creates a garbled Bloom filter $GBF_{C\cap S}$ of size m as follows. For $0 \leq i \leq m-1$ if $BF_C[i] = 0$ then $GBF_{C\cap S}[i] \xleftarrow{r} \{0,1\}^\lambda$; if $BF_C[i] = 1$, the client decrypts c_i and gets $d_i = E^{-1}(sk, c_i)$, checks whether the first λ-bit equals r_i that is sent in step 1. If yes then skip the first λ bits in d_i and copy the second λ bits to $GBF_{C\cap S}[i]$. Otherwise output $\perp$ and terminate. Finally, the client queries $GBF_{C\cap S}$ with its own set C and outputs $C \cap S$.

Figure 3: The Enhanced PSI protocol

In a full universe attack, a malicious client encodes the full universe of all possible elements in its Bloom filter and uses it in the PSI protocol to learn the server's entire set. A Bloom filter can easily represent the full universe by setting all the bits to 1. This is a special feature of Bloom filters and it causes a problem when we try to construct a simulator for the client in the malicious model. Namely, when the adversary uses the all-one Bloom filter, the simulator needs to enumerate all elements in the universe and send them to the trusted party in the ideal process. Without making any assumptions, the universe is potentially too large and a polynomial time algorithm may fail to enumerate all elements.

To prevent the full universe attack, we add a step to make sure that the client's Bloom filter is not all-one. More specifically, the server uses a symmetric key block cipher to encrypt strings in its garbled Bloom filter before transferring them to the client. It forces the client to behave honestly by splitting the key into m shares using a $(m/2, m)$-secret sharing scheme. The client uses the bit array in its Bloom filter as the selection string to receive the intersection garbled Bloom filter and the shares of the key. If the bit in the selection string is 0, the client receives a share of the key; if the bit is 1, the client receives an encrypted string in GBF_S. The intuition is that if the client cheats by using an all-one Bloom filter, it will not be able to gather enough shares to recover the key, and thus will not be able to decrypt the encrypted garbled Bloom filter. In the protocol we set $m = 2kn$ in order to make sure that the client's Bloom filter has at least $m/2$ 0 bits to receive enough shares to recover the key. Since the client has at most n elements and each element needs to be hashed k times, then the number of 1 bits in BF_C will never exceed $kn = m/2$, consequently the number of 0 bits will always be at least $m/2$. Although in this setting m is not optimal, the overhead is acceptable given the optimal number of m is about $1.44kn$.

The added step will not affect the client's privacy, but may affect the correctness of the protocol if a malicious server sends wrong shares of the key or uses a different key to encrypt its garbled Bloom filter. The client cannot detect it because the key is random and the strings in the garbled Bloom filter look random. To prevent this malicious behavior, we also require the client to send m λ-bit random strings $(r_0, ..., r_{m-1})$ to the server before the OT. For each $GBF_S[i]$, the server encrypts $r_i||GBF_S[i]$ ($||$ means concatenation) and sends the ciphertext in the OT. After the transfer, the client can recover the key and decrypt the received ciphertexts. If the server is honest, then the client can correctly decrypt using the key it recovered and r_i should present in the decrypted message. For each garbled Bloom filter string the client received, the probability of the server getting away with cheating is $2^{-\lambda}$.

	Ours	De Cristofaro's	Huang's
80	SHA-1, NIST P-192 curve	RSA 1024, SHA-1	1024-bit p, 160-bit q, SHA-1
128	SHA-1 (filter), SHA-256 (OT), NIST P-256 curve	RSA 3072, SHA-1	3072-bit p, 256-bit q, SHA-1
192	SHA-1 (filter), SHA-384 (OT), NIST P-384 curve	RSA 7680, SHA-1	7680-bit p, 384-bit q, SHA-256
256	SHA-1 (filter), SHA-512 (OT), NIST P-521 curve	RSA 15360, SHA-1	15360-bit p, 512-bit q, SHA-256

Table 2: Security parameters and settings

Efficiency In [26] a fully secure version of the OT extension protocol is given. It uses the cut-and-choose approach to ensure a malicious party can cheat with at most $2^{-\Omega(\lambda)}$ probability. The major overhead of the fully secure protocol is introduced by the non-optimal m and cut-and-choose, which increase the communication and computation complexity of the semi-honest one by a factor of 1.4λ. Overhead introduced by other parts of our protocol is small. The additional computational overhead in our protocol includes: the server needs to perform m encryptions and to use the threshold secret sharing scheme to split the key, the client needs to perform $m/2$ decryptions, to recover the key. The additional communication overhead in our protocol includes: $m \cdot \lambda$ bits for sending the random strings of in step 1.

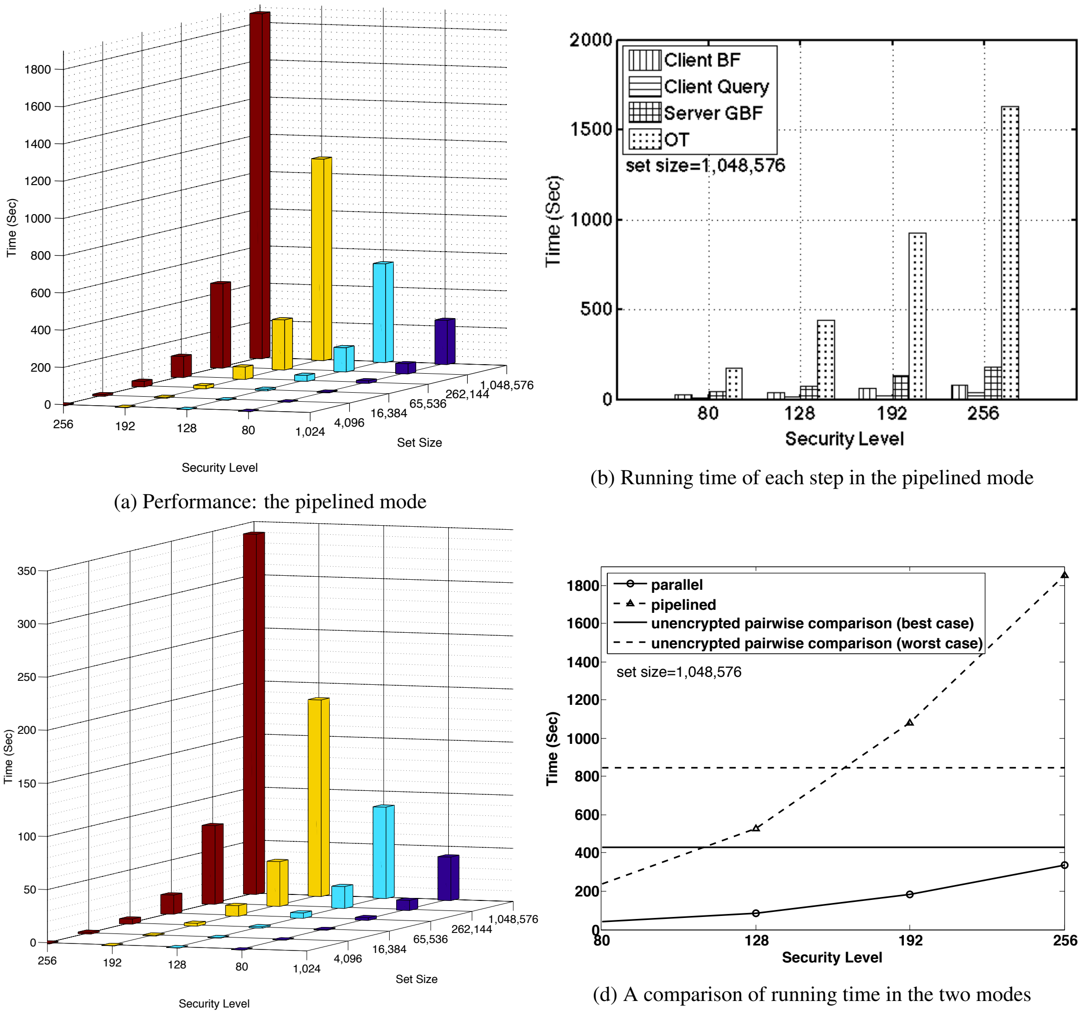
6. IMPLEMENTATION AND EVALUATION

6.1 Implementation

We have implemented a prototype of the basic protocol in C. The source code (and its Java port) is released online[1]. It uses OpenSSL (1.0.1e) for the cryptographic operations. We currently use keyed SHA-1 to build/query Bloom filters and garbled Bloom filters[2]. Namely each $h_i(x)$ is instantiated as $sha1(s_i||x) \bmod m$, where s_i is a unique salt. We implement the semi-honest OT extension protocol [26] on top of the Naor-Pinkas OT protocol [34].

[1] http://personal.cis.strath.ac.uk/changyu.dong/PSI/PSI.html

[2] Cryptographically strong hash functions are not necessary here. Later we will change to more efficient hash functions e.g. MurmurHash [2] that has been used by Apache Hadoop and Cassandra in their Bloom filter implementation.



(a) Performance: the pipelined mode

(b) Running time of each step in the pipelined mode

(c) Performance: the parallel mode

(d) A comparison of running time in the two modes

Figure 4: Performance of our basic protocol

The hash functions in the OT extension protocol are instantiated depending on the security parameters. When hash values need to be truncated, the truncation follows the steps specified by the NIST [14]. We use the NIST elliptic curve groups over $\mathbb{F}_p$ [36] for the public key operations required by the Naor-Pinkas OT protocol. We use elliptic curve groups because they are much faster than integer groups at high security levels.

The C prototype has two executables, one for the client and one for the server. The client and server communicate through TCP. The prototype can work in two modes: pipelined and parallel. In the pipelined mode, on each side, the computation is done in a single thread, an additional thread transmits data in parallel when possible. Parallel data transmission enables the server or the client to start working immediately without waiting for the other party to complete its computation. The parallel mode extends the pipeline mode by utilizing all CPU cores and distributing tasks on all cores evenly. Our test result shows that the parallel mode can improve the performance significantly on multicore systems. This is due to the fact that the computation in our protocol is dominated by independent hashing. Namely, on each side, n independent set elements each needs to be hashed k times to build the Bloom filter or the garbled Bloom filter, also hashing of m matrix rows are needed in the OT extension protocol. As the data to be hashed is independent, this is a perfect SPMD (single program multiple data) scenario. The program detects the number of cores available, decides the number of threads, evenly allocates a portion of data to each thread, and then launch the threads to execute the tasks in parallel. The hash values are then consumed by main threads that run the protocol. This approach requires only minimal changes to the program structure. For example, only one line (line 8) needs to be changed in Algorithm 1. Namely instead of hashing the element, the algorithm reads from an array a precomputed index number.

6.2 Performance Evaluation

In this section we show the performance evaluation results of our prototype. All experiments were conducted on two Mac computers. The server is a Mac Pro with 2 Intel E5645 6-core 2.4GHz CPUs, 32 GB RAM and runs Mac OS X 10.8. The client is a Macbook Pro laptop with an Intel 2720QM quad-core 2.2 GHz CPU, 16 GB RAM and runs Mac OS X 10.7. The two computers are connected by 1000M Ethernet. The security settings of the experiments in this and the next section are summarized in Table 2. In all experiments we set BF/GBF parameter $k = \lambda$ so the false positive probability of a BF is at most $2^{-\lambda}$, we set m to be the optimal value $kn \log_2 e$. For example, at 80-bit security $k = \lambda = 80$, and when $n = 2^{20}$, $m = 120795960$. We use randomly generated `int` sets in the ex-

Set size / protocol	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
	80-bit Security					
Huang's(Java)	19	65	331	2049	22853	98468†
Our pipelined (Java)	0.693	2.34	7.02	31.5	110.6	426
Our parallel (Java)	0.195	0.431	1.42	6.31	25	91
De Cristofaro's (C)	0.590	2.41	9.84	41.3	159	641
Our pipelined (C)	0.275	0.863	3.37	13.9	54.0	237
Our parallel (C)	0.075	0.207	0.642	2.49	9.49	40.9
	256-bit Security					
Huang's (Java)	32	157	733	4647	43156	185570†
Our pipelined (Java)	8.2	20.3	68.44	313.4	1298	5421
Our parallel (Java)	1.5	3.2	10.5	54	215	1132
De Cristofaro's (C)	462	1850	7419	29654	118286	473144†
Our pipelined (C)	4.09	8.94	29.8	113	453	1852
Our parallel (C)	0.741	1.53	4.68	17.8	74.2	339

All time shown in the table are in seconds. † – estimated running time

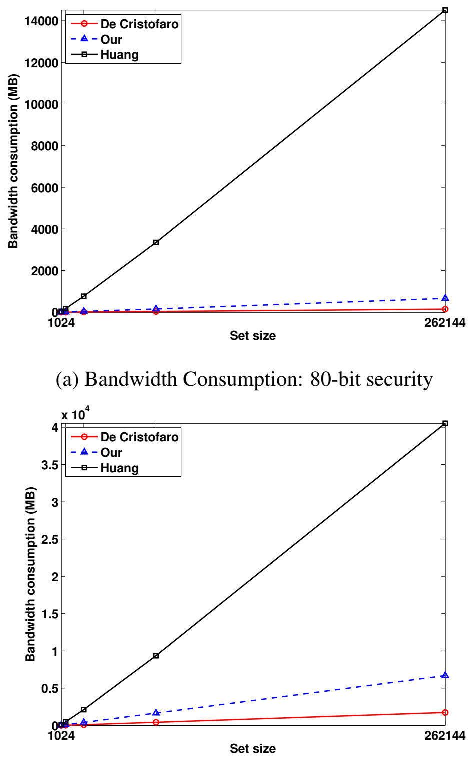
Table 3: Performance comparison

periments. We measure the total running time of the protocol. The measurement starts from the client sending the request and ends immediately after the client outputting the intersection. The time includes all operations such as building the Bloom filter, building the garbled Bloom filter, the full OT extension protocol (including the underlying Naor-Pinkas OT), data transmission, and the client-side query for obtaining the intersection. We do not, however, include the time for initialization tasks, e.g. to generate random sets, to interpret the command line arguments, and to setup sockets.

We first show the performance of the prototype working in the pipelined mode. In the pipelined mode, all computation on each side is done in a single thread. We vary the set size (n) from 2^{10} to 2^{20} and security parameters (λ and k) from 80 to 256. The result is shown in Figure 4a. We can see the running time increases almost linearly in the set size at each security level. And for each increase in security parameter, the running time increases only by a factor of approximately 2. We also measured the time for each individual step of the protocol. In the experiments, we fix the set size (2^{20}) and vary only security levels. The result is shown in Figure 4b. We can see the protocol running time is dominated by the OT execution. This suggests that with a more efficient OT protocol, the total running time can be further reduced.

Then we show the performance of the parallel mode. In the parallel mode, we use multiple threads for computation. The result is shown in Figure 4c. The total running time in the parallel mode is much less than in the pipelined mode. At 80-bit security, million elements set intersection can be done in 41 seconds. In the highest security setting, the same computation can be done in 339 seconds – that is less than 6 minutes. A comparison of the performance in the two modes is shown in Figure 4d. The client has 4 cores and the server has 12 cores, and we can see that the parallel mode is about 5 times faster than the pipelined mode. This shows that our protocol can fully take the advantage of the multicore architecture. We believe the ability to easily scale up to multiple cores is a clear advantage of our protocol and makes the protocol suitable for large scale private data processing.

The performance of our protocol can even beat some inefficient plain algorithms in some settings. For example, Figure 4d shows the time needed for a single threaded C program to compute the intersection of two unencrypted random sets ($n = 2^{20}$) by pairwisely comparing the elements. It needs 429 seconds in the best case when $C = S$, and needs 844 seconds in the worst case when $C \cap S = \emptyset$.



(a) Bandwidth Consumption: 80-bit security

(b) Bandwidth Consumption: 256-bit security

Figure 5: Bandwidth Consumption Comparison

6.3 Performance Comparison

We compared the performance of our basic protocol against two other semi-honest PSI protocols. The protocols we compared to are De Cristofaro's RSA-OPRF protocol (implemented in C) and Huang's Sort-Compare-Shuffle with Waksman Network protocol (implemented in Java). They are previously the fastest PSI protocols and the code has been optimized by the authors. We test the two protocols on the same hardware and OSes that we use for testing ours. De Cristofaro's C implementation is compiled with OpenSSL 1.0.1e and GMP 5.1.1 using gcc. The RSA public exponent is 3 in all tests. We run Huang's Java code using Java 1.7.0_12. The element bit length in Huang's protocol is set to 32. As it is unfair to compare the performance of Huang's Java code with our C code, we ported our C code to Java and measured the performance.

We measured the total running time of the protocols. De Cristofaro's code outputs running time so we use the output directly[3]. Huang's code has no such output, and we measure the running time of the $execution()$ function in the $Program$ class.

The comparison in Table 3 shows that in all settings, both modes of our protocol are faster than the other two protocols. Both De Cristofaro's implementation and Huang's implementation pipeline the protocol execution, which is exactly what we do in the pipelined mode. Therefore the performance of these three can be compared directly[4]. The performance of De Cristofaro's protocol is close to ours at 80-bit security and is faster than Huang's. But when the se-

[3] We exclude the running time of the last step in the protocol. In this step the client searches the hash values received in the protocol to find the intersection. This step is excluded because it uses an inefficient pairwise comparison and the authors plan to replace it with a hashtable search.

[4] De Cristofaro's code uses two threads on each side for computation. But this does not affect the comparison result.

curity parameter increases to 256-bit, it becomes much slower than our protocol and Huang's. This is because De Cristofaro's protocol is based mainly on public key operations, while ours and Huang's protocols rely on mostly symmetric key operations. Put aside differences caused by languages and implementation, our protocol is faster than Huang's because it requires the same number of public key operations but significantly less symmetric key operations. For example, at 80-bit security with 2^{20} input size, our protocol requires 0.4 billion symmetric key operations, while Huang's requires 8.5 billion (1.7 billion non-free gates, each requires 4 symmetric key operations to build and another 1 to evaluate).

We skip the test with the biggest input size (2^{20}) on De Cristofaro's protocol at 256-bit security because it would take too long. The running time of De Cristofaro's protocol is linear in the input size, our estimation is that it would need 131 hours to finish. This estimation is based on the result of test with 2^{18}-element sets at the same security level. The JVM on the client computer ran out of memory (16 GB) when we testing Huang's protocol with 2^{20}-element sets at 80-bit security. The test was repeated twice and both times we got the same error. We could not finish the test but base on the test result of input size 2^{18}, we estimate the test would need 27 hours. This estimation is somehow far from the time reported by the authors, that is 6 hours. However the test had been running for more than 24 hours before the JVM threw the error. Therefore we believe the estimation is reasonable. We observed excessive paging activities during the test on the client computer because the JVM occupied all free memory (14 GB). This may account for the difference between our estimation and the authors' measurement. Because at 256-bit security Huang's protocol requires even more memory, we skip the test with 2^{20} input size and estimate the running time to be 51 hours from test result of input size 2^{18}.

We also measured bandwidth consumption of the protocols. As we couldn't finish the tests with the other protocols using 2^{20} input size, the largest input size we used in the experiment was 2^{18}. The results are shown in Figure 5. As we can see, the bandwidth consumption of De Cristofaro's and our protocol is almost linear. Our protocol consumes more bandwidth than De Cristofaro's protocol but less than Huang's protocol.

6.4 Further Parallelization

GPGPUs For many personal computers, a readily available massive parallel computing device is the graphic cards. Modern GPUs have hundreds of processing cores and can provide ample computation cycles and high memory bandwidth to massively parallel applications. The computation in our protocol can be easily parallelized and therefore is an ideal application for GPU acceleration. We have started implementing the protocol on top of OpenCL [3]. A test on our GPU version of SHA-1 shows that on an ATI Radeon HD 5770 graphic card, it only takes 37.5 milliseconds to perform 1 million hash operations. This is about 5 times faster than a single 2.4 GHz CPU core.

Extremely Big Data Set & Cloud Computing In practice, to process extremely big data set, we have to distribute the task on multiple computers. New computing paradigms such as cloud computing make it possible to execute such distributed tasks "on demand". Our protocol can be easily deployed on cloud platforms. Here we show how to do it with the semi-honest protocol. The fully secure protocol case is similar. From a high level point of view, the client and the server throw their elements into bins using an hash function. Each side has b bins and each bin contains about $\lceil \frac{n}{b} \rceil$ elements. Then they build Bloom filters and garbled Bloom filters for each bin. The parameter k is still determined by the desired false positive probability, the parameter m is determined by k and the bin size. The filters are associated with the bin number. Then for each $0 \leq i < b$, the server uses OT to transfer the garbled Bloom filter for bin i to the client, who uses its Bloom filter for bin i as the selection string. The client then queries all elements in its bin i against the received garbled Bloom filter and adds any positive elements into the result set. In the end, the client has the intersection. Conceptually, this splits a big set into b smaller sets that each can be handled by a single node. It is correct because the two parties use the same hash function so an element thrown by the server into bin i will also be threw by the client into bin i. The idea can be implemented using the MapReduce programming model [18] easily. For example, figure 6 depicts the MapReduce procedure of the first step on the server side with 3 bins: the map function takes a portion of the input set and maps an element into a key-value pair such that the key is the bin number and the value is a tuple consists of the element and k index numbers. The MapReduce framework shuffles and groups together the values returned by the map function that have the same key. The reduce function generates a garbled Bloom filter of a certain bin and outputs it for OT. We are currently experimenting with Hadoop [1] to implement the protocol in MapReduce.

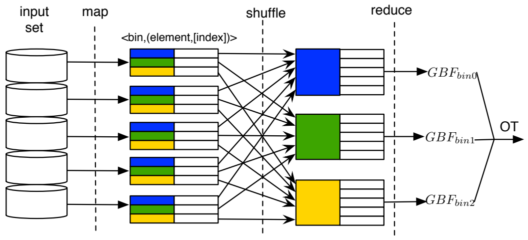


Figure 6: MapReduce on the server side

7. CONCLUSION AND FUTURE WORK

In this paper we presented a highly efficient and scalable PSI protocol based on oblivious Bloom intersection. The protocol depends mostly on efficient symmetric key operations and the operations can be parallelized easily. We presented two variants of the protocol: the basic one is secure in the semi-honest model and the enhanced one is secure in the malicious model. The performance evaluation and comparison results show that our protocol is orders of magnitude faster than the previously fastest protocols. The results also show that our protocol can fully utilize the parallel processing capability provided by the multicore architecture. The efficiency and scalability make our protocol suitable for large scale privacy preserving data processing.

As discussed in Section 6.4, we are in the process of prototyping the protocol on GPGPUs and MapReduce. The preliminary results of this work is encouraging. We hope more parallelization options could enable more applications in various computing environments.

In the field of cryptographic protocols, we have seen many examples that a new protocol improves performance of previous work by using a better algorithm. It is different in this work: the performance gain comes mainly from a better data structure. We would like to continue our research along this line. Namely we will investigate, adapt and design better data structures, so that they can be used in the design of more efficient cryptographic protocols.

Acknowledgements

Changyu Dong is supported by a science faculty starter grant from the University of Strathclyde. Zikai Wen is supported by an undergraduate research internship from the University of Strathclyde. We thank Emiliano De Cristofaro and Yan Huang for sharing their PSI source code. We also thank the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] Hadoop. http://hadoop.apache.org/.

[2] Murmurhash. https://code.google.com/p/smhasher/.

[3] Opencl. http://www.khronos.org/opencl/.

[4] C. C. Aggarwal and P. S. Yu, editors. *Privacy-Preserving Data Mining - Models and Algorithms*, volume 34 of *Advances in Database Systems*. Springer, 2008.

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

[6] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *ACM Conference on Computer and Communications Security*, pages 691–702, 2011.

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

[8] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

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

[10] 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.

[11] 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.

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

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

[14] Q. Dang. SP 800-107 (rev. 1). recommendation for applications using approved hash algorithms. Technical report, Gaithersburg, MD, United States, 2012.

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

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

[17] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *TRUST*, pages 55–73, 2012.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[19] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: An efficient and scalable protocol. Cryptology ePrint Archive, Report 2013/515, 2013.

[20] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

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

[22] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[23] 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.

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

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

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

[27] 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.

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

[29] F. Kerschbaum. Outsourced private set intersection using homomorphic encryption. In *ASIACCS*, pages 85–86, 2012.

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

[31] D. Many, M. Burkhart, and X. Dimitropoulos. Fast private set operations with sepia. Technical Report 345, Mar 2012.

[32] G. Mezzour, A. Perrig, V. D. Gligor, and P. Papadimitratos. Privacy-preserving relationship path discovery in social networks. In *CANS*, pages 189–208, 2009.

[33] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. Botgrep: Finding P2P bots with structured graph analysis. In *USENIX Security Symposium*, pages 95–110, 2010.

[34] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA*, pages 448–457, 2001.

[35] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *NDSS*, 2011.

[36] NIST. Recommended elliptic curves for federal government use, 1999.

[37] O. Papapetrou, W. Siberski, and W. Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *Distributed and Parallel Databases*, 28(2-3):119–156, 2010.

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

[39] B. Schneier. *Applied cryptography - protocols, algorithms, and source code in C (2. ed.)*. Wiley, 1996.

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