Unbalanced Approximate Private Set Intersection

Amanda Cristina Davi Resende and Diego F. Aranha

Institute of Computing – University of Campinas (UNICAMP)
{amanda.resende,dfaranha}@ic.unicamp.br

Abstract. Protocols for Private Set Intersection (PSI) are an important cryptographic primitive to perform joint operations on datasets in a privacy-preserving way. They allow two entities to compute the intersection of their private sets without revealing any additional information beyond the intersection, which can be learned by only one of the parties, as in one-way PSI protocols, or by both parties in mutual PSI protocols. In addition, the PSI setting may be *unbalanced* when one set is substantially smaller than the other or *balanced* when the sets have approximately the same size. However, even with several PSI protocols already proposed, applications keep using insecure naive approaches that are more efficient in both run time and communication. To make matters worse, implementations in the literature do not use the best possible implementation techniques, especially when implementing PSI protocols instantiated with curve-based public-key cryptography. This paper proposes an efficient one-way PSI protocol based on public-key cryptography for the unbalanced scenario. Security is based on the hardness of the One-More-Gap-Diffie-Hellman (OMGDH) problem against semi-honest adversaries and includes forward secrecy on the client side. A Cuckoo filter is also used to reduce the amount of data exchanged and stored by the client. Our implementation employs the state-of-the-art Galbraith-Lin-Scot (GLS-254) binary elliptic curve with point compression.

Keywords: Cuckoo filter, Private Set Intersection, unbalanced PSI, software implementation

1 Introduction

Private Set Intersection (PSI) is a special case of secure multiparty computation (MPC) where two parties perform joint operations on datasets while preserving privacy. They have been used in several applications such as relationship path discovery in social networks [1], botnet detection [2], proximity testing [3], cheater detection in online games [4], genetic testing of fully-sequenced human genomes [5] and private contact discovery [6]. Although the PSI problem was formally posed only in 2004 by Freedman *et al.* [7], who also presented a proposal for solving it, the first related protocol can be found in 1986, when Meadows [8] proposed a cryptographic matching protocol to authenticate two parties.

PSI protocols allow two parties storing a set of private data such as patients list, criminal suspects or telephone contacts to compute the intersection of their sets without revealing any additional information beyond the intersection to one or both entities. PSI protocols can be divided into one-way PSI, i.e., only one of the parties learns the intersection; or mutual PSI (mPSI), in which both parties learn the intersection. The focus of this work is one-way PSI protocols. For more information about mPSI the reader is invited to read [9,10,11,12,13,14].

PSI protocols can also be classified based on behavior. In the literature, Chen *et al.* [6] define the PSI setting as symmetric when the sets have approximately the same size and asymmetric when one of the sets is substantially smaller than the other. We propose a new terminology to prevent confusion with the type of cryptographic primitive being used (symmetric or asymmetric): *balanced* for sets with approximately the same size and *unbalanced* for the opposite scenario[1]. *Exact* PSI protocols return the exact intersection of the datasets, while *approximate* PSI protocols suit applications that tolerate a small probability for an element not in the intersection to be returned by the protocol (error rate). This is a builtin feature of the protocol (by construction), not accidentally caused by hash function collisions, for example.

However, even with several PSI protocols already proposed, many applications have used naive solutions (as later discussed in Section 2.2), because they are more efficient in both run time and communication. Protocols proposed and implemented in several papers by Pinkas *et al.* [15,16,17] are efficient in terms of computation (by using mostly symmetric operations), but need to transmit a lot of data, while other works based on public-key cryptography [8,18,5,6] need to transmit fewer data, but require less efficient operations. Thus, the choice of the PSI protocol depends on the PSI setting (balanced or unbalanced), the network bandwidth, storage space, security properties, among other factors.

This paper proposes a PSI protocol based on public-key cryptography that is very efficient for unbalanced PSI, requires a small storage space (3MB for the server with set size of 2^{20} elements) and transmits a small amount of data during each execution (less than 1MB when the client set has 11041 elements). Because of this, the network bandwidth does not impact the protocol efficiency even in constrained scenarios. Our proposal also ensures forward secrecy on the client side (usually more vulnerable than the server), which guarantees that elements exchanged in the past will remain confidential even if long-term secrets (keys) are exposed. In addition, we adapted and implemented[2] the most promising PSI protocols in the literature using techniques that are considered to be the state of the art in curve-based public-key cryptography.

Our contributions. The main contributions of this paper are:

- An efficient, practical and simple PSI protocol based on public-key cryptography for unbalanced PSI that ensures forward secrecy on the client side.

[1] Throughout this paper, the client set is always the smaller one.

[2] Some implementations and library were obtained from Pinkas *et al.* [17], available in `https://github.com/encryptogroup/PSI`. We will explain this in details in the Section 5.2.

- The application of a Cuckoo filter to reduce the amount of data to be exchanged by the protocol or stored by the client. The Cuckoo filter requires less storage space than others similar approaches, like Bloom filter and Cuckoo hashing (for false positive rates less than 3%) [19], provides the delete operation (important in some applications) and the lookup operation is performed in $O(b)$, where b is the number of entries per bucket (more details in Section 4.2). The Cuckoo filter (or any similar approach) introduces a false positive rate that can be adjusted according to the needs of the application, resulting in an *approximate* protocol.
- Efficient software implementation of the protocols using the Galbraith-Lin-Scot (GLS-254) binary elliptic curve with point compression. To the best of our knowledge, this is the first time that a state-of-the-art implementation is used to instantiate PSI protocols that rely on this type of operation.

Private contact discovery. An interesting application for our protocol is private contact discovery. In this problem, a user signs up to a social network such as WhatsApp, Signal or Telegram, and would like to discover which contacts in their address book are also registered without disclosing to the service operator the entire contents of his/her address book. The user typically has a set with a few hundred of contacts, while the social network can have from a few million to a few billion users, characterizing the unbalanced setting.

Secure messaging applications such as TextSecure/Signal[3] and Secret[4], currently employ insecure approaches (see Section 2.2) to “solve” the private contact discovery problem, due to their higher efficiency in both run time and communication when compared to secure protocols. In our proposal, both run time and the amount of data transmitted are based only on the client set size. Since this set is small, the protocol is computationally efficient and needs to exchange small amounts of data regardless of the social network set size. This happens because, during the preprocessing phase, the social network information is inserted into a Cuckoo filter and transmitted to the client to be used only at the end of the protocol for a fast pertinence test. Notice that this application tolerates approximate results and requires a delete operation when updating the contacts.

Organization. This paper is organized as follows. In Section 2, we show notation and terminology used during the development of this work, a classification of the PSI protocols into categories and a brief overview of the main protocols in each class. In Sections 3 and 4, our basic protocol is presented and the optimizations are proposed, respectively. In Section 5 we show the results and compare them with the most promising protocols from the literature. Finally, in Section 6 we present the conclusions.

[3] `https://whispersystems.org/signal/privacy/`

[4] `https://medium.com/@davidbyttow/demystifying-secret-12ab82fda29f\#.5433o6e8h`

2 Related work for PSI protocols

We start by formalizing the notation used throughout the paper and other relevant definitions.

2.1 Notation and terminology

- P_1 and P_2 are the participating parties of the protocols, where P_1 is the server and P_2 the client, except when referring to server-aided (third party) PSI protocols.
- X and Y are the respective input sets of P_1 and P_2, with size $n_1 = |X|$ and $n_2 = |Y|$. The set X is denoted by $\{x_1, x_2, ..., x_{n_1}\}$ and the set Y by $\{y_1, y_2, ..., y_{n_2}\}$ where each element x_i and y_i has bit-length σ, for $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$.
- For a set S, the notation $x \overset{R}{\leftarrow} S$ indicates that x was sampled from S with uniform probability.
- The operation $a \overset{?}{=} b$ denotes the comparison whether a is equal or not to b.
- $\kappa = 128$ is the symmetric security parameter.
- $\rho = 40$ is the statistical security parameter (hashing failure).
- $\varphi = 256$ is the size of the representation of a point in the Galbraith-Lin-Scot (GLS-254) binary elliptic curve when using point compression (number of bits to store one x-coordinate and two trace bits).
- $\eta = 30$ is the hash collision parameter, i.e., the probability of a hash collision occurring is $< 2^{-30}$.
- The output of the hash function in some cases is defined as $l = \rho + log\ n_1 + log\ n_2$, as suggested by [17], instead of $2 \cdot \kappa$. This produces the collision probability $2^{-\eta}$, which is suitable for most applications.
- For the Cuckoo filter, we also define v as the fingerprint length (in bits), w as the load factor ($0 \leq w \leq 1$), b as the number of entries per buckets, m as the number of buckets and ψ as the false positive rate.

2.2 Classification and related work of PSI protocols

Many one-way PSI protocols have been proposed in the open research literature [7,20,21,22,23,24,15,16,17,6]. They are constructed based on several primitives such as Bloom filters [25], Cuckoo hashing [26], Oblivious Polynomial Evaluation (OPE) [27], Oblivious Pseudorandom Function (OPRF) [28], Garbled Circuits (GC) [29,30], Unpredictable Function [22], Homomorphic Encryption [31,32], Oblivious Transfer (OT) [33,34], among others.

Following Pinkas *et al.* [15,16,17], PSI protocols can be classified into: naive hashing (or naive solution), server-aided PSI (or third party-based PSI), PSI based on generic protocols (or circuit-based PSI), OT-based PSI and PSI based on public-key cryptography.

Naive hashing. Both P_1 and P_2 use a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^l$ to compute the hash of their elements. P_1 then computes $h_i = H(x_i)$ while P_2 computes $h_j = H(y_j)$, where $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$. After computing the hashes, P_1 sends values h_j to P_2 which computes the intersection by checking if $h_j \stackrel{?}{=} h_i$ for all values of i and j.

This approach is very efficient both in run time and communication. P_1 must compute n_1 hash functions while P_2 computes n_2 hash functions. P_2 does not need to send any data to P_1, while P_1 sends only $n_1 l$ bits to P_2. In the end, P_2 also needs to do comparisons to find the intersection, which can be done with complexity $O(1)$ in average when using a hash table.

However, if the hash function inputs were taken from a low-entropy domain $\mathbb{D}$, P_2 can obtain all elements of P_1 by performing a brute-force attack, i.e., for every possible element $z \in \mathbb{D}$, P_2 verifies if $H(z) \stackrel{?}{=} h_i$. One solution could be to choose $\mathbb{D}$ with high entropy when possible. This would prevent the problem, but consecutive executions of the protocol would still leak repeated elements and would not guarantee forward secrecy since P_2 can verify if a specific element $z \in \mathbb{D}$ was part of the P_1 set, by just checking if $H(z) \stackrel{?}{=} h_i$. Nonetheless, the insecure protocol is employed by messaging applications for the private contact discovery problem, and social networks like Facebook[5], Twitter[6] and Snapchat[7] to measure advertisement conversion rates.

Server-aided PSI. Several works in the literature [24,35,14] have employed a third party, in this case called as server, to achieve better performance in PSI protocols. The server can be semi-honest (can not deviate from protocol, learns only by observing communication between parties), covert (if it deviates from protocol, it is detected with some probability by an honest party) or malicious (can arbitrarily deviate from the protocol). However, such protocols are secure only if the third party does not collude with any of the other parties, thus having a different security model from conventional protocols.

In [35], Kamara *et al.* present a semi-honest server protocol that takes 580s ($\cong$ 10 minutes) with 12,394MB (12GB) of communication and 100 threads to evaluate 2 sets of 1 billion elements each. In the protocol, P_1 samples $K \in \{0,1\}^k$ and sends to P_2 (in this approach both P_1 and P_2 are considered clients). Both P_1 and P_2 calculate $f_1 = \pi_1(F_K(x_1), ..., F_K(x_i))$ and $f_2 = \pi_2(F_K(y_1), ..., F_K(y_j))$, respectively, where $F : \{0,1\}^k \cdot \mathbb{D} \rightarrow \{0,1\}^{\geq k}$ is a pseudorandom permutation, π is a random permutation for $1 \leq i, j \leq 10^9$, and then send f_1 and f_2 to the semi-honest server. The server calculates the intersection $I = f_1 \cap f_2$ and sends I to P_2 which obtains the intersection by computing $F_k^{-1}(e)$ for each $e \in I$.

[5] `https://www.wired.com/2014/12/oracle-buys-data-collection-company-datalogix/`

[6] `https://support.twitter.com/articles/20170410`

[7] `https://www.wsj.com/articles/snapchat-to-enable-ad-targeting-using-third-party-data-1484823600`

PSI based on generic protocols. Generic secure computation uses Arithmetic or Boolean circuits to securely evaluate functions, among them, the set intersection. In [23], Huang *et al.* presented several of these protocols using Boolean circuits, all of them constructed using Yao's garbled circuits [29,30].

The simplest protocol described in [23] involves the comparison of each element from P_1 with each element from P_2. This approach is known as Pairwise-Comparison (PWC) and involves $O(n^2)$ comparisons, which does not scales well for large sets. Another more efficient approach presented in [23], the Sort-Compare-Shuffle (SCS) circuit, is more efficient, with complexity $O(n \log n)$. SCS first sorts the union of the P_1 and P_2 elements, then compares if adjacent elements are the same, and finally shuffles the result to prevent information leakage.

The great advantage of this type of protocol is that they can be easily adapted to any other features that PSI protocols may require, such as revealing only the intersection size or whether the size is larger or smaller than an upper bound. However, despite the improvements in recent years, generic protocols still have a very high run time compared to other protocols.

OT-based PSI. This category of protocols is the most recent and, up to date, the most promising, mainly because of the large performance improvements from OT extensions. The first protocol was proposed in 2013 by Dong *et al.* [24], combining Bloom filters and OT [36] in their construction.

In 2014, Pinkas *et al.* [15] presented improvements to [24] and also proposed a new and more efficient protocol combining OT and hashing. In 2015, Pinkas *et al.* [16] have shown that the proposal presented previously [15] could be improved by using the permutation-based hash technique [37], since it reduces the size of each element stored in the bins, which until then was the main overhead of the protocol. In 2016, Pinkas *et al.* [17] presented improvements for their earlier protocols, where the protocol complexity no longer depends on the size of each element.

The protocol presented in [17] is the state of the art for balanced one-way PSI protocols and, depending on the scenario (network bandwidth), also for unbalanced PSI protocols, with security against semi-honest adversaries. By using only symmetric operations in almost all of its construction, the protocol is extremely efficient[8].

Public-key cryptography based PSI. Meadows [8] and Huberman *et al.* [18] proposed one of the earliest PSI approaches based on public-key cryptography, even before of the PSI problem was formally defined in [7]. Both protocols were based on the Diffie-Hellman (DH) key exchange, taking advantage of its commutative properties. DH-based PSI protocols use the Random Oracle Model (ROM) to prove their security, while Freedman *et al.* [7,38] introduced PSI protocols based on the standard model, which are secure against both semi-honest

[8] The protocol presented in [17] uses asymmetric operations [33] to generate the OT bases. However, the cost of these operations is negligible when the number of elements evaluated is substantially greater than the value of κ.

and malicious adversaries and are based on the ElGamal cryptosystem. In [21], Cristofaro and Tsudik presented a PSI protocol based on blind-RSA.

Later Jarecki *et al.* [22] presented a PSI protocol that is secure against malicious adversaries based on a Parallel Oblivious Unpredictable Function (POUF). In [5], Baldi *et al.* showed a “simpler” version of this protocol, with security only against semi-honest adversaries. This protocol is used to obtain the results of this paper. Another relevant public-key PSI protocol was presented by Chen *et al.* [6], based on the protocol presented by Pinkas *et al.* [17], but instead of performing OPRF (via OT) operations, it uses the Fan-Vercauteren (FV) leveled Fully Homomorphic Encryption (FHE) scheme. This change considerably decreases the amount of data to be transmitted in the unbalanced setting.Therefore, depending on the unbalanced setting and the network bandwidth, the protocol presented by Chen *et al.* [6] is faster than Pinkas *et al.* [17]. The good performance is however restricted to 32-bit elements due to limitations in the parameters of the FHE scheme.

The most recent work was presented by Kiss *et al.* [39]. They noted that in some PSI protocols the server can perform operations on its data only once and send the result to the client, which will use them in the future executions to compute the intersection. They independently proposed a Bloom filter (or a counting Bloom filter) to decrease the amount of data to be transmitted or stored by the client. These observations are very important in an unbalanced setting, since all operations and communication are only performed considering the smaller client set. In terms of security, there is an important limitation in their approach: in the closest protocol to our proposal (DH-based PSI [8,18]), the client and server reuse the same keys across all executions, which does not provide forward secrecy. In terms of performance, during the preprocessing phase alone (the setup phase, as in the paper), the server should send $n_1\varphi$ bits to the client and the client computes n_1 exponentiations. If $n_1 = 2^{24}$, it will be necessary to transmit 512MB and to perform 2^{24} exponentiations on the client side.

3 The Basic Protocol

Jarecki and Lui [22] presented a PSI protocol secure against malicious adversaries based on the hardness of the One-More-Gap-Diffie-Hellman (OMGDH) problem and a Zero-Knowledge Proof (ZKP). Latter, Baldi *et al.* [5] simplified this protocol, by removing the ZKP, to be secure only against semi-honest adversaries.

The simplified protocol is shown in Figure 1 and works as follows: for each element x_i of the set X, the server computes the hash $H(x_i)$, the exponentiation $H(x_i)^\alpha$ with the same exponent for all the elements, and again computes the hash $t_{x_i} = H'(H(x_i)^\alpha)$, sending values tx_i to the client. For each element y_i of the set Y, the client computes the hash $H(y_j)$, the exponentiation $a_j = H(y_j)^{\beta_j}$ with ephemeral exponents β_j and sends a_j to the server. The server calculates $a'_j = (a_j)^\alpha$ for each a_j using the same α used previously and sends a'_j to the client. The client then calculates $ty_j = H'((a'_j)^{1/\beta_j})$, by “removing” the exponents that were applied earlier. Finally, the client computes the intersection by comparing if $ty_j \in \{tx_1, tx_2, ..., tx_{n_1}\}$. The long-term secret is the server key α.

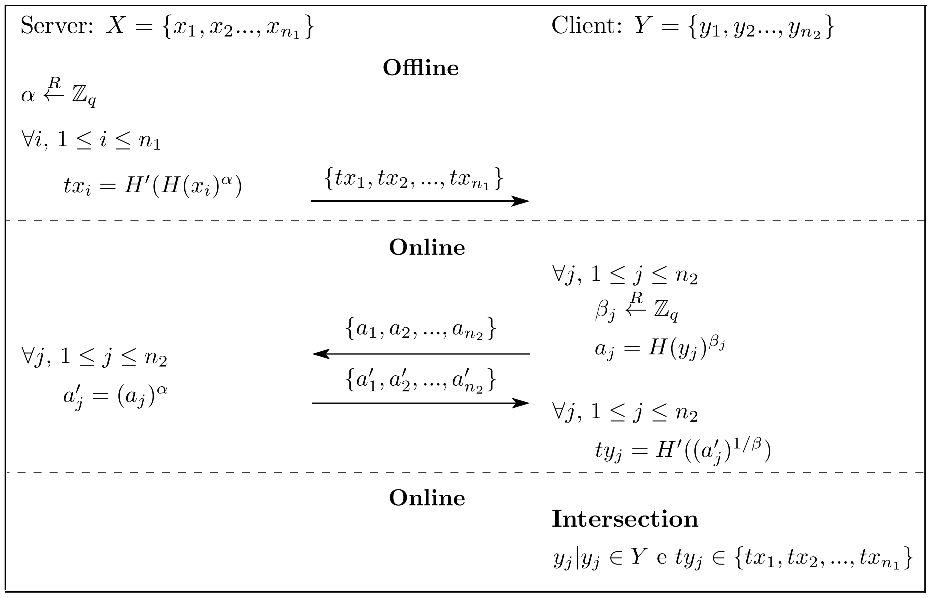
Server: $X = \{x_1, x_2..., x_{n_1}\}$
Client: $Y = \{y_1, y_2..., y_{n_2}\}$
Offline
$\alpha \overset{R}{\leftarrow} \mathbb{Z}_q$
$\forall i,\ 1 \leq i \leq n_1$
$tx_i = H'(H(x_i)^{\alpha})$
$\{tx_1, tx_2, ..., tx_{n_1}\}$
Online
$\forall j,\ 1 \leq j \leq n_2$
$\beta_j \overset{R}{\leftarrow} \mathbb{Z}_q$
$a_j = H(y_j)^{\beta_j}$
$\{a_1, a_2, ..., a_{n_2}\}$
$\forall j,\ 1 \leq j \leq n_2$
$a'_j = (a_j)^{\alpha}$
$\{a'_1, a'_2, ..., a'_{n_2}\}$
$\forall j,\ 1 \leq j \leq n_2$
$ty_j = H'((a'_j)^{1/\beta})$
Online
Intersection
$y_j | y_j \in Y$ e $ty_j \in \{tx_1, tx_2, ..., tx_{n_1}\}$

Fig. 1. Basic PSI protocol proposed in [5] as a simplification of [22] [5, Adapted].

4 Optimizations

We propose two improvements in the protocol presented in the Section 3: executing the offline phase just once, storing the informations in a database and reducing the size of this database. A similar figure as Figure 1 with our optimizations applied can be found in Appendix B. Besides that, we implemented the GLS-254 elliptic curve, which improves the performance of the protocols that use Elliptic Curve Cryptography (ECC).

4.1 Generating the database

As it can be seen in the Figure 1, the protocol is divided into two parts: offline and online. The offline part is executed without the need of any communication from the server to the client, except for any negotiation to define the initial parameters such as the group $\mathbb{G}$ and its order q. Thus, the server can mask all elements using α and the hash functions H and H' ($tx_i = H'(H(x_i)^{\alpha})$) before receiving connections.

Because of this feature, the offline part can be performed only once between server and client, where the server would calculate the mask of each element and send it to the client, which would store them for use in each execution of the protocol. Therefore, only the online part of the protocol needs to be used.

Informally, this setting does not affect the protocol security (against semi-honest adversaries), since the client cannot recover α due to the hardness of the OMGDH problem. In addition, because the preimage space for the hash function is large (high entropy), the client cannot map all possible inputs and check whether or not an element belongs to the server set without interacting with the server, thereby preserving forward secrecy.

This setting is very efficient when it is used in unbalanced PSI protocols, where the smallest set is the client. This happens because in the online part all operations are performed only on the client elements ($3n_2$ asymmetric cryptographic operations and $2n_2\varphi$ bits are transmitted).

4.2 Reducing the database size

The database size increases with the number of server elements. For example, assuming each masked server element has $l = \rho + \log n_1 + \log n_2$ bits and if $\rho = 40$, as defined in Section 2.1, with $n_2 = 2^8$ and $n_1 = 2^{24}$, each masked element would have $l = 72$ bits. Since the server has 2^{24} elements, all server masked elements will occupy 144MB. However, if the scale changes from a few million to a few billion of elements, as is the case of a large social network with approximately 2^{30} users, the server masked elements would need 9.75GB of space.

Downloading and storing this data on devices with low memory resources, such as mobile devices, or with constrained network connection (low bandwidth or/and high latency) can be prohibitive. To reduce the size of the data, techniques like Bloom filters and their variants [25,40,41], Cuckoo hashing [26] and Cuckoo filter [42] may be used.

We chose the Cuckoo filter instead of the Bloom filter or Cuckoo hashing since it has the delete operation (which is fundamental in private contact discovery) using less space than the Bloom filter variants and the Cuckoo hashing by storing only the element's fingerprint.

Cuckoo filter. A Cuckoo filter is a compact variant of a Cuckoo hashing proposed by Fan *et al.* [42] that stores only the element's fingerprint. A Cuckoo filter consists of a set of buckets each with b entries. This type of filter allows performing 3 types of operations/algorithms: insertion, lookup and deletion. In the following, we briefly summarize how these operations work. For more details, the reader is invited to read [42,43].

For an element x to be inserted, it is necessary to compute its fingerprint f and its two possible candidate buckets $b_1(x) = h(x)$ and $b_2(x) = b_1(x) \oplus h(f)$, where h is a hash function. If one of them has an empty entry, the fingerprint f is inserted into that entry, completing the insertion process. However, if all entries in the 2 buckets are filled, one of the two buckets (say b_2) and an entry containing e are randomly selected, and then the fingerprint f is inserted in place of e and a new bucket $b_3 = b_2 \oplus h(f)$ is calculated for e. This is repeated until one empty entry is found or the maximum number of attempts is reached. In a simple way, the insertion algorithm tries to relocate the elements between its two possible buckets[9].

The lookup operation is simple. Given an element x', compute its fingerprint f', the two possible buckets b'_1 and b'_2 and, if f' is in b'_1 or b'_2 then the element x' has been inserted and the algorithm returns true, otherwise false. The delete operation is as simple as the lookup. First, calculate the element fingerprint, the two possible buckets, check if any of the entries correspond to fingerprint and if so, a copy[10] is removed. This operation is important to our approach because it is not necessary to generate a new filter every time that an element (or a set of elements) is deleted, which happens when using traditional Bloom filters.

4.3 Efficient software implementation of GLS-254 elliptic curve

Our implementation of ECC is based on the latest version of the GLS-254 software [44] available in SUPERCOP[11]. The binary GLS curve is a particularly efficient choice for our target platform due to its native support to binary field arithmetic, the lambda coordinate system [45] and the GLS endomorphism for fast scalar multiplications [46], achieving the current speed record for this operation. The code is structured in three layers: an efficient vectorized implementation of binary field arithmetic targeting Intel instruction sets; a regular window-based method for variable-base scalar multiplication implemented in constant time; a thin protocol layer implementing the DH key exchange. The exponentiations in our protocol were heavily based on the two last layers, while hashing and point compression were directly implemented over the field arithmetic.

The approach selected for hashing was a combination of the SHA256 hash function with the binary Shallue-van de Woestijne well-bounded encoding algorithm [47]. Elements are first hashed to a binary field element $u \in \mathbb{F}_{2^m}$ using SHA256, and then the encoding outputs the lambda coordinates (x, λ) of a point over the binary elliptic curve. This approach requires only a single inversion, a quadratic equation solution and some cheaper field operations, and provides better statistical properties than popular try-and-increment heuristics. Point compression adapts a rather classical technique [48]. The λ coordinate defined over a quadratic extension $\mathbb{F}_{2^{2m}}[s]/(s^2 + s + 1)$ as $(\lambda_0 + \lambda_1 s)$ is compressed to a pair of trace values $(Tr(\lambda_0), Tr(\lambda_1))$, which can later be used to solve a quadratic equation and disambiguate among the four possible solutions. In total, 256 bits are used by combining the 254 bits of the x coordinate with the two trace bits. Decompression again requires a field inversion, solving a quadratic equation and some cheaper binary field operations. As a result, our entire code runs in constant time for side-channel resistance, including the quadratic solver [47].

[9] The operation $\oplus$ guarantees that b_1 can be calculated through b_2.

[10] Two distinct elements can share the same bucket and the same fingerprint. This implies that it is possible to have false positives.

[11] `https://bench.cr.yp.to`

5 Implementation and experimental evaluation

5.1 Benchmarking environment

We ran our experiments in a computer equipped with an Intel Haswell i7-4770K quadcore CPU with 3.4 GHz and 16 GB of RAM with Turbo Boost turned off. All tests were performed using only this machine, and network bandwidth and latency were simulated using the Linux command `tc` (network simulation code can be found in Appendix A). For the Local Area Network (LAN) setting, the two parties (client and server) are connected via localhost with 10Gbps of throughput and a 0.2ms Round-Trip Time (RTT). In addition to the LAN, we also consider three Wide Area Network (WAN) settings with 100 Mbps, 10 Mbps and 1 Mbps of bandwidth, each with an 80ms RTT. These settings follow what was proposed by Chen *et al.* [6].

To evaluate the performance of the PSI protocols both balanced and unbalanced PSI settings were used. In the balanced scenario, $n_1 = n_2 \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$ as proposed by Pinkas *et al.* [16,17]. In the unbalanced scenario $n_2 \in \{5535, 11041\}$ and $n_1 \in \{2^{16}, 2^{20}, 2^{24}\}$, as proposed by Chen *et al.* [6]. The size of each element was set to be 32 bits ($\sigma = 32$ bits), but this does not impact the performance of our protocol due to hashing. The output of the hash function used in the DH based on ECC [8,18] and in Baldi *et al.* [5] is l, as defined in the Section 2.1. The run time of each protocol was measured from the beginning of the execution until the client computes the intersection. Each protocol was executed 10 times and the run times were computed as the average of these executions, as done in [17,6].

5.2 Implementation

Implementations DH-ECC [8,18] and OT + Hashing [17] were obtained from Pinkas *et al.* [17], available at `https://github.com/encryptogroup/PSI`. They used OpenSSL (v.1.0.1e) for the symmetric cryptographic primitives, the implementation of [49] for the OT extension, available at `https://github.com/encryptogroup/OTExtension`, and the MIRACL library (v.5.6.1) for ECC. According to our benchmarking, a scalar multiplication within their code base takes 1.2 million cycles[12] for an exponentiation, which indicates a misconfigured version of MIRACL. Up to now, there is no implementation available for the Chen *et al.* [6] protocol, but we try to reproduce the benchmarking scenarios as close as possible to their work.

We implemented our proposal and the Baldi *et al.* protocol [5] using the software provided by Pinkas *et al.* [17], but replacing the implementation of the Koblitz K-283 elliptic curve available in MIRACL. Our implementation of the GLS-254 curve takes around 50,000 cycles to compute an exponentiation, which is 24x faster than [17]. We used the Cuckoo filter implementation of Fan *et al.* [42] available at `https://github.com/efficient/cuckoofilter`.

[12] Average of 2^{20} exponentiations performed on our Haswell machine.

All protocols were implemented using C and C++ programming languages and executed using the same hardware. The same libraries were used to perform the cryptographic operations, except for the OTs in OT + Hashing [17] which still use the Koblitz curve. This does not impact the run time of the protocol, since the cost of this operation is negligible when the number of elements is large.

5.3 Preprocessing

To improve performance, some PSI protocols can be divided into 2 phases (online and offline) without impact in security. The offline part of the protocol can be executed only once and reused in future executions. In the protocol proposed by Chen *et al.* [6], the server precomputes some values to facilitate the underlying FHE multiplications. In this case, only the server will use the precomputed data and no transfer to the client is required.

Unlike the Chen *et al.* protocol [6], in our basic protocol [5] presented in Section 3, the server can preprocess the encryption/masking of all elements and send them to the client. The client must store and reuse this data in all subsequent executions of the protocol to calculate the intersection, as presented in Section 4.1. Beyond using the precomputing allowed in the basic protocol, our approach also inserts each encrypted element into a Cuckoo filter (according to Section 4.2) in order to reduce the data that must be transmitted to the client and that should be stored.

Table 1 presents the preprocessing and data transmission time using the network settings defined in Section 5.1. The run times of Chen *et al.* were obtained from [6], and since some parameters of the FHE can generate more efficient processing depending on the configuration, the preprocessing column may have two different values (we separate them with the symbol *) that will be used in the next section. It is interesting to note that the preprocessing run times of our proposal and our implementation of Baldi *et al.* [5] are practically the same, since we perform the same operations, however by employing a Cuckoo filter to reduce the amount of data to be transmitted, our approach is up to 3.3x faster than Baldi *et al.* [5].

5.4 Comparison to others PSI protocols

The performance evaluation of the protocols will be divided into two scenarios: unbalanced and balanced setting. As the code of the protocol presented by Chen *et al.* [6] was not available, we compared our protocol with theirs only in the unbalanced case, with results obtained from [6]. However, according to Chen *et al.* [6], the protocol can be easily extended to the balanced PSI scenario.

As shown in Section 2.2, PSI protocols can be classified into 5 categories. Because the naive hashing and server-aided approaches have different security notions from the other protocols, they will not be analyzed. PSI based on generic protocols are out of scope, because they have limitations in run time and memory.

					LAN	WAN		
Protocol	n_1	n_2	Preprocessing	Comm.	10 Gbps	100 Mbps	10 Mbps	1 Mbps
Chen *et al.* [6]	2^{24}	11041	**70.9, 76.8***	-	-	-	-	-
		5535	**64.1, 71.2***	-	-	-	-	-
	2^{20}	11041	**6.4**	-	-	-	-	-
		5535	**4.3**	-	-	-	-	-
	2^{16}	11041	**1.0**	-	-	-	-	-
		5535	**0.7**	-	-	-	-	-
Baldi *et al.* [5]	2^{24}	11041	334.17	160.00	0.13	15.73	136.32	1,345.55
		5535						
	2^{20}	11041	20.91	10.00	0.01	1.10	8.38	84.40
		5535						
	2^{16}	11041	1.31	0.56	0.01	0.19	0.53	5.09
		5535						
Our protocol	2^{24}	11041	333.62	**48.00**	**0.06**	**4.82**	**40.71**	**403.68**
		5535						
	2^{20}	11041	20.78	**3.00**	**0.00**	**0.60**	**2.55**	**25.63**
		5535						
	2^{16}	11041	1.30	**0.19**	**0.00**	**0.01**	**0.19**	**1.56**
		5535						

Table 1. Preprocessing and transmission time for PSI protocols. The WAN setting has 80ms RTT and the LAN 0.02ms RTT. For the filter in our proposal we have 16-bit fingerprints ($v = 16$), 3 entries per buckets ($b = 3$), load factor of 66.6% ($w = 0.66$) and m buckets. The communication is given in MB and the time in seconds. Zero values refer to numbers smaller than $5 \cdot 10^{-3}$. Best values marked in bold.

Among the two remaining categories, OT-based PSI and PSI based on public-key cryptography, we will analyze the best protocol in each category comparing the results with our proposal.

Unbalanced PSI protocols. In many applications where it is necessary to compute the private set intersection, the sets have unequal sizes. In the client/server approach, the server usually has a set from millions to billions of elements while the client has only a few hundred, such as in the case of private contact discovery. We followed what was proposed by Chen *et al.* [6] with $n_2 \in \{5535, 11041\}$ and $n_1 \in \{2^{16}, 2^{20}, 2^{24}\}$.

Table 2 shows the run time (in seconds) and the communication (in MBs) of the unbalanced scenario considering both the LAN and WAN settings. We have analyzed the best protocol for the OT-based PSI, the two best protocols for PSI based on public-key cryptography and we compare them with our proposal.

Amongst the public-key protocols, our proposal and the Baldi *et al.* protocol [5] have the same communication cost ($2n_2$ bits) and, regarding run time, our approach is slightly better by employing the Cuckoo filter in the server database, what makes the final computation of the intersection more efficient, since the filter is already constructed and the lookup is done in $O(b)$ ($b = 3$, in this case). Because of this, we omitted the figures related to Baldi *et al.* protocol [5] in the table. In addition, comparing with the Chen[13] *et al.* protocol [6], our approach transmits up to 59x less data and is up to 76x faster with 10 Gbps, for $n_2 = 5535$ and $n_1 = 2^{24}$. Comparing our protocol with OT + Hashing [17], our approach transmits up to 1,413x less data and is up to 74x faster with 10 Gbps of bandwidth and 946x faster with 1 Mbps, for $n_2 = 5535$ and $n_1 = 2^{24}$.

		Parameters		Comm	LAN	WAN		
Type	Protocol	n_1	n_2	Size (MB)	10 Gbps	100 Mbps	10 Mbps	1 Mbps
OT	OT+Hashing [17]	2^{24}	11041	480.9	40.5	88.0	449.5	4,084.8
			5535	480.4	40.1	87.9	449.2	4,080.6
		2^{20}	11041	30.9	3.3	7.0	29.8	263.7
			5535	30.4	3.1	6.8	29.0	260.0
		2^{16}	11041	2.6	0.7	1.5	3.3	21.6
			5535	2.1	0.7	1.4	2.9	19.8
Public key	Chen *et al.* [6]	2^{24}	11041	23.2, 21.1*	44.5	46.9	63.5	214.0*
			5535	20.1, 12.5*	41.1	43.1	49.1*	139.9*
		2^{20}	11041	11.5	6.4	7.6	15.8	99.0
			5535	5.6	8.6	9.2	13.3	53.6
		2^{16}	11041	4.1	2.0	2.4	5.4	35.0
			5535	2.6	1.1	1.3	3.2	21.8
	Our protocol	2^{24}	11041	**0.67**	**0.87**	**1.52**	**1.86**	**7.81**
			5535	**0.34**	**0.54**	**1.04**	**1.21**	**4.31**
		2^{20}	11041	**0.67**	**0.67**	**1.31**	**1.65**	**7.59**
			5535	**0.34**	**0.34**	**0.83**	**1.00**	**3.97**
		2^{16}	11041	**0.67**	**0.66**	**1.29**	**1.64**	**7.57**
			5535	**0.34**	**0.33**	**0.82**	**0.99**	**3.93**

Table 2. Run time in seconds and communication in MBs for unbalanced PSI protocols. Times are taken at the client because it finishes last. In the communication column, the protocol of Chen *et al.* [6] may have 2 values due to different parameters used in the FHE system. For more information, see [6]. Best values marked in bold.

Our approach performs well for unbalanced scenario because our operations depend only on the client set, with $2n_2$ bits transmitted and $3n_2$ exponentiations. Although exponentiations are considered an expensive operation when performed a small number of times and with a good elliptic curve implementation, a curve-based protocol becomes competitive with the others.

[13] In the communication column of Table 2, the protocol [6] can have 2 different values, because according to the network setting it is better that the operations take more time and generate less data than to the operations take less time but produce more data. This trade-off can be raised in the FHE by adjusting the system parameters.

Balanced PSI protocols. Table 3 presents the run time (in seconds) and the communication (in MBs) of the balanced scenario considering both a LAN and a WAN setting. The results show, as expected, that the OT + Hashing protocol [17] has the best run time, being 9.5x faster than DH-ECC [8,18] and 14.5x faster than our approach with 10 Gbps. This is due to the fact that OT + Hashing [17] uses practically only symmetric operations, that are faster than asymmetric used by public-key protocols.

Furthermore, by analyzing only public-key based protocols, the DH-ECC protocol [8,18] is approximately 35% faster than the proposed approach with 10 Gbps. This happens because in DH-ECC [8,18] it is possible to perform client and server operations in parallel, while in our proposal it is not possible (as shown in Figure 1). However, the DH-ECC [8,18] and our protocol exchange 32% and 42% less data, respectively, than OT + Hashing [17]. For the WAN setting with 1Mbps of bandwidth, both approaches are faster than OT + Hashing [17] by transmitting less data.

		Parameters	Comm	LAN	WAN		
Type	Protocol	$n_1 = n_2$	Size (MB)	10 Gbps	100 Mbps	10 Mbps	1 Mbps
OT	OT + Hashing [17]	2^{24}	1,756.83	**67.86**	**218.62**	**1,518.18**	14,800.33
		2^{20}	106.83	**4.70**	**14.79**	**93.54**	902.31
		2^{16}	6.52	**0.66**	**2.13**	6.74	57.11
		2^{12}	0.43	0.36	0.93	1.09	3.88
		2^{8}	0.05	0.34	0.66	0.68	0.87
Public key	DH-ECC [8,18]	2^{24}	1,200.00	641.09	741.44	1,647.09	10,716.27
		2^{20}	74.00	39.91	46.56	102.37	662.58
		2^{16}	4.56	2.49	3.40	**6.61**	41.88
		2^{12}	0.28	**0.18**	**0.59**	**0.67**	2.80
		2^{8}	0.02	**0.01**	**0.25**	**0.25**	0.32
	Our protocol	2^{24}	**1,024.00**	991.83	1,090.96	1,863.41	**9,599.94**
		2^{20}	**64.00**	62.00	68.77	116.41	**600.66**
		2^{16}	**4.00**	3.87	5.24	7.81	**38.68**
		2^{12}	**0.25**	0.24	0.72	0.80	**2.51**
		2^{8}	**0.02**	0.02	**0.25**	**0.25**	**0.26**

Table 3. Run time in seconds and communication in MB for balanced PSI protocols where $n_1 = n_2$. Times are the client because they are the biggest one. Best values marked in bold.

Comparison with Kiss *et al.* [39]. In a very recent paper, Kiss *et al.* [39] present several PSI protocols, where the closest to our proposal is ECC-DH-PSI [8,18]. In the preprocessing stage, the server needs to compute n_1 exponentiations like in our protocol, but the client also needs to compute n_1 exponentiations. In some applications, such as private contact discovery, this amount of exponentiations in the client side could be prohibitive, because typically the client has a smartphone. Considering $n_1 = 2^{20}$ and according to [39], the preprocessing takes 1,325.400s while our proposal takes 21s (using a 1Gbps network), that is 63x faster. Moreover, the server sends $n_1\varphi$ ($\varphi = 284$ in their case), that adds up to 35.5MB for $n_1 = 2^{20}$, while our proposal just sends a 2.25MB filter (with $\psi = 0.07\%$). This is 15.7x less data to be transmitted.

In the online phase the amount of data to be transmitted is asymptotically the same, $2n_2\varphi$, but concretely Kiss *et al.* [39] use $\varphi = 284$ for the K-283 curve with compression and we have $\varphi = 256$ for the GLS-254 curve. Considering the number of exponentiations, their approach needs to compute $2n_2$ operations while our protocol computes $3n_2$. This advantage happens because the ECC-DH-PSI from [39] does not provide forward secrecy on the client side and reuse the same key across all protocol executions.

In order to reduce the amount of data to be stored by the client, Kiss *et al.* [39] use a Bloom filter, while our approach employs a Cuckoo filter. The Cuckoo filter allows deletions while the Bloom filter does not (counting Bloom filter allows deletions using 3-4x more space) and use 30% less space than the Bloom filter for the same false positive rate [43].

In summary, our protocol provides an efficient preprocessing phase, forward secrecy on the client side and a filter that needs less storage space. The ECC-DH-PSI protocol from [39] has an asymptotically faster online phase, but the performance improvement is small in the unbalanced setting when n_2 is small. Moreover, their protocol does not provide any forward secrecy to clients and the preprocessing phase is expensive and can be prohibitive on mobile devices.

6 Conclusions

Private set intersection is an important cryptographic primitive allowing two parties to perform joint operations on their private sets without revealing any additional information beyond the intersection. Despite many protocols available in the literature, few of them provide solutions that are efficient in both run time and data transmission. Usually, in most approaches, the computational cost is based in both the server and in the client set sizes, giving no advantages in the unbalanced setting.

We proposed an efficient, practical and simple PSI protocol that is based on public-key cryptography for unbalanced sets that ensures forward secrecy on the client side. Additionally, we implemented the protocol using the GLS-254 binary elliptic curve with point compression using techniques considered state of the art, that allows a better comparison with the other proposed approaches.

Our protocol with this implementation provides an interesting trade-off between preprocessing and the online phase of the protocol, where for n_1^{24} the preprocessing takes less than 6 minutes (remembering that this phase needs to be done only once) and the online phase for $n_2 = 11041$ takes less than 8 seconds even with 1Mbps bandwidth. The client needs to store only 48MB of information for this configuration.

Acknowledgements

We thank Fabian Monrose for suggesting the Cuckoo filter as a data structure and Anderson Nascimento for discussions about this work.

References

1. G. Mezzour, A. Perrig, V. D. Gligor, and P. Papadimitratos, "Privacy-Preserving Relationship Path Discovery in Social Networks," in *CANS*, vol. 5888 of *Lecture Notes in Computer Science*, pp. 189–208, Springer, 2009.
2. S. Nagaraja, P. Mittal, C. Hong, M. Caesar, and N. Borisov, "BotGrep: Finding P2P Bots with Structured Graph Analysis," in *USENIX Security Symposium*, pp. 95–110, USENIX Association, 2010.
3. A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh, "Location Privacy via Private Proximity Testing," in *NDSS*, The Internet Society, 2011.
4. 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*, pp. 506–520, IEEE Computer Society, 2011.
5. P. Baldi, R. Baronio, E. D. 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*, pp. 691–702, ACM, 2011.
6. H. Chen, K. Laine, and P. Rindal, "Fast Private Set Intersection from Homomorphic Encryption," *IACR Cryptology ePrint Archive*, vol. 2017, p. 299, 2017.
7. M. J. Freedman, K. Nissim, and B. Pinkas, "Efficient Private Matching and Set Intersection," in *EUROCRYPT*, vol. 3027 of *Lecture Notes in Computer Science*, pp. 1–19, Springer, 2004.
8. C. A. Meadows, "A More Efficient Cryptographic Matchmaking Protocol for Use in the Absence of a Continuously Available Third Party," in *IEEE Symposium on Security and Privacy*, pp. 134–137, IEEE Computer Society, 1986.
9. L. Kissner and D. X. Song, "Privacy-Preserving Set Operations," in *CRYPTO*, vol. 3621 of *Lecture Notes in Computer Science*, pp. 241–257, Springer, 2005.
10. J. Camenisch and G. M. Zaverucha, "Private Intersection of Certified Sets," in *Financial Cryptography*, vol. 5628 of *Lecture Notes in Computer Science*, pp. 108–127, Springer, 2009.
11. M. Kim, H. T. Lee, and J. H. Cheon, "Mutual Private Set Intersection with Linear Complexity," in *WISA*, vol. 7115 of *Lecture Notes in Computer Science*, pp. 219–231, Springer, 2011.
12. C. Dong, L. Chen, J. Camenisch, and G. Russello, "Fair Private Set Intersection with a Semi-trusted Arbiter," in *DBSec*, vol. 7964 of *Lecture Notes in Computer Science*, pp. 128–144, Springer, 2013.
13. S. K. Debnath and R. Dutta, "A Fair and Efficient Mutual Private Set Intersection Protocol from a Two-Way Oblivious Pseudorandom Function," in *ICISC*, vol. 8949 of *Lecture Notes in Computer Science*, pp. 343–359, Springer, 2014.
14. S. K. Debnath and R. Dutta, "Towards Fair Mutual Private Set Intersection with Linear Complexity," *Security and Communication Networks*, vol. 9, no. 11, pp. 1589–1612, 2016.
15. B. Pinkas, T. Schneider, and M. Zohner, "Faster Private Set Intersection Based on OT Extension," in *USENIX Security Symposium*, pp. 797–812, USENIX Association, 2014.

16. B. Pinkas, T. Schneider, G. Segev, and M. Zohner, "Phasing: Private Set Intersection Using Permutation-based Hashing," in *USENIX Security Symposium*, pp. 515–530, USENIX Association, 2015.
17. B. Pinkas, T. Schneider, and M. Zohner, "Scalable Private Set Intersection Based on OT Extension," *IACR Cryptology ePrint Archive*, vol. 2016, p. 930, 2016.
18. B. A. Huberman, M. K. Franklin, and T. Hogg, "Enhancing Privacy and Trust in Electronic Communities," in *EC*, pp. 78–86, 1999.
19. B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo Filter: Practically Better than Bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014* (A. Seneviratne, C. Diot, J. Kurose, A. Chaintreau, and L. Rizzo, eds.), pp. 75–88, ACM, 2014.
20. S. Jarecki and X. Liu, "Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection," in *TCC*, vol. 5444 of *Lecture Notes in Computer Science*, pp. 577–594, Springer, 2009.
21. E. D. Cristofaro and G. Tsudik, "Practical Private Set Intersection Protocols with Linear Complexity," in *Financial Cryptography*, vol. 6052 of *Lecture Notes in Computer Science*, pp. 143–159, Springer, 2010.
22. S. Jarecki and X. Liu, "Fast Secure Computation of Set Intersection," in *SCN*, vol. 6280 of *Lecture Notes in Computer Science*, pp. 418–435, Springer, 2010.
23. Y. Huang, D. Evans, and J. Katz, "Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?," in *NDSS*, The Internet Society, 2012.
24. C. Dong, L. Chen, and Z. Wen, "When Private Set Intersection Meets Big Data: An Efficient and Scalable Protocol," in *ACM Conference on Computer and Communications Security*, pp. 789–800, ACM, 2013.
25. B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
26. R. Pagh and F. F. Rodler, "Cuckoo Hashing," in *ESA*, vol. 2161 of *Lecture Notes in Computer Science*, pp. 121–133, Springer, 2001.
27. M. Naor and B. Pinkas, "Oblivious Transfer and Polynomial Evaluation," in *STOC*, pp. 245–254, ACM, 1999.
28. M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, "Keyword Search and Oblivious Pseudorandom Functions," in *TCC*, vol. 3378 of *Lecture Notes in Computer Science*, pp. 303–324, Springer, 2005.
29. A. C. Yao, "Protocols for Secure Computations (Extended Abstract)," in *FOCS*, pp. 160–164, IEEE Computer Society, 1982.
30. A. C. Yao, "How to Generate and Exchange Secrets (Extended Abstract)," in *FOCS*, pp. 162–167, IEEE Computer Society, 1986.
31. C. Gentry, "Fully Homomorphic Encryption using Ideal Lattices," in *STOC*, pp. 169–178, ACM, 2009.
32. J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.
33. M. Naor and B. Pinkas, "Efficient Oblivious Transfer Protocols," in *SODA*, pp. 448–457, ACM/SIAM, 2001.
34. V. Kolesnikov and R. Kumaresan, "Improved OT Extension for Transferring Short Secrets," in *CRYPTO (2)*, vol. 8043 of *Lecture Notes in Computer Science*, pp. 54–70, Springer, 2013.
35. S. Kamara, P. Mohassel, M. Raykova, and S. S. Sadeghian, "Scaling Private Set Intersection to Billion-Element Sets," in *Financial Cryptography*, vol. 8437 of *Lecture Notes in Computer Science*, pp. 195–215, Springer, 2014.

36. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending Oblivious Transfers Efficiently," in *CRYPTO*, vol. 2729 of *Lecture Notes in Computer Science*, pp. 145–161, Springer, 2003.
37. Y. Arbitman, M. Naor, and G. Segev, "Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation," in *FOCS*, pp. 787–796, IEEE Computer Society, 2010.
38. M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas, "Efficient Set Intersection with Simulation-Based Security," *J. Cryptology*, vol. 29, no. 1, pp. 115–155, 2016.
39. A. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas, "Private Set Intersection for Unequal Set Sizes with Mobile Application," *PoPETs*, vol. 2017, no. 4, pp. 97–117, 2017.
40. L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.
41. F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," in *ESA*, vol. 4168 of *Lecture Notes in Computer Science*, pp. 684–695, Springer, 2006.
42. B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *CoNEXT*, pp. 75–88, ACM, 2014.
43. D. Eppstein, "Cuckoo Filter: Simplification and Analysis," in *SWAT*, vol. 53 of *LIPIcs*, pp. 8:1–8:12, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
44. T. Oliveira, D. F. Aranha, J. L. Hernandez, and F. Rodríguez-Henríquez, "Improving the performance of GLS254." CHES Rump Session, 2016.
45. T. Oliveira, J. López, D. F. Aranha, and F. Rodríguez-Henríquez, "Two is the Fastest Prime: Lambda Coordinates for Binary Elliptic Curves," *J. Cryptographic Engineering*, vol. 4, no. 1, pp. 3–17, 2014.
46. D. Hankerson, K. Karabina, and A. Menezes, "Analyzing the galbraith-lin-scott point multiplication method for elliptic curves over binary fields," *IEEE Trans. Computers*, vol. 58, no. 10, pp. 1411–1420, 2009.
47. D. F. Aranha, P. Fouque, C. Qian, M. Tibouchi, and J. Zapalowicz, "Binary Elligator Squared," *IACR Cryptology ePrint Archive*, vol. 2014, p. 486, 2014.
48. J. Lopez and R. Dahab, "New Point Compression Algorithms for Binary Curves," in *2006 IEEE Information Theory Workshop - ITW '06 Punta del Este*, pp. 126–130, March 2006.
49. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More Efficient Oblivious Transfer and Extensions for Faster Secure Computation," in *ACM Conference on Computer and Communications Security*, pp. 535–548, ACM, 2013.
50. "Traffic shaping, bandwidth shaping, packet shaping with linux tc htb." `https://www.iplocation.net/traffic-control`. Accessed: 2017-05-30.

Appendix A - Network simulation

The simulation code was obtained from [50] and a few changes were made.

```
#!/bin/bash
#
#  tc uses the following units when passed as a parameter.
#  kbps: Kilobytes per second
#  mbps: Megabytes per second
#  kbit: Kilobits per second
#  mbit: Megabits per second
#  bps: Bytes per second
#       Amounts of data can be specified in:
#       kb or k: Kilobytes
#       mb or m: Megabytes
#       mbit: Megabits
#       kbit: Kilobits
#  To get the byte figure from bits, divide the number by 8 bit

# Name of the traffic control command.
TC=/sbin/tc

IF=lo              # The network interface
IP=127.0.0.1       # IP address of the machine we are controlling

DNLD=100mbit       # Download limit (in Megabits)
UPLD=100mbit       # Upload limit (in Megabits)
RTT=40ms           # RTT (in mega bits)

# Filter options for limiting the intended interface.
U32="$TC␣filter␣add␣dev␣$IF␣protocol␣ip␣parent␣1:0␣prio␣1␣u32"

start() {
# We'll use Hierarchical Token Bucket (HTB) to shape bandwidth.
# For detailed configuration options, please consult Linux man page.

$TC qdisc add dev $IF root handle 1: htb default 30
$TC class add dev $IF parent 1: classid 1:1 htb rate $DNLD ceil $DNLD
$TC class add dev $IF parent 1: classid 1:2 htb rate $UPLD ceil $UPLD
$U32 match ip dst $IP/32 flowid 1:1
$U32 match ip src $IP/32 flowid 1:2

$TC qdisc add dev $IF parent 1:1 netem delay $RTT
$TC qdisc add dev $IF parent 1:2 netem delay $RTT
}

stop() {
# Stop the bandwidth shaping.
$TC qdisc del dev $IF root
}

restart() {
# Self-explanatory.
stop
sleep 1
start
}

show() {
# Display status of traffic control status.
$TC -s qdisc ls dev $IF
}

case "$1" in

start)
echo -n "Starting␣bandwidth␣shaping:␣"
start
echo "done"
;;

stop)
echo -n "Stopping␣bandwidth␣shaping:␣"
stop
echo "done"
;;

restart)
echo -n "Restarting␣bandwidth␣shaping:␣"
```

```
restart
echo "done"
;;

show)
echo "Bandwidth␣shaping␣status␣for␣$IF:"
show
echo ""
;;

*)
pwd=$(pwd)
echo "Usage:␣tc.bash␣{start|stop|restart|show}"
;;

esac
exit 0
```

Appendix B - Our protocol

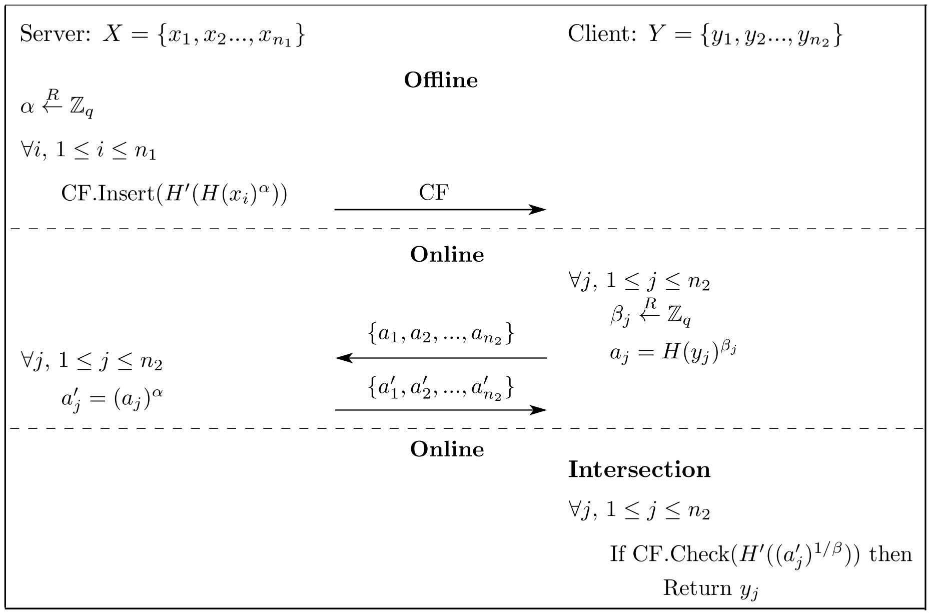
Server: $X = \{x_1, x_2 ..., x_{n_1}\}$
Client: $Y = \{y_1, y_2 ..., y_{n_2}\}$
Offline
$\alpha \overset{R}{\leftarrow} \mathbb{Z}_q$
$\forall i,\ 1 \le i \le n_1$
CF.Insert$(H'(H(x_i)^{\alpha}))$
CF
Online
$\forall j,\ 1 \le j \le n_2$
$\beta_j \overset{R}{\leftarrow} \mathbb{Z}_q$
$a_j = H(y_j)^{\beta_j}$
$\{a_1, a_2, ..., a_{n_2}\}$
$\forall j,\ 1 \le j \le n_2$
$a'_j = (a_j)^{\alpha}$
$\{a'_1, a'_2, ..., a'_{n_2}\}$
Online
Intersection
$\forall j,\ 1 \le j \le n_2$
If CF.Check$(H'((a'_j)^{1/\beta}))$ then
Return y_j

Fig. 2. Our protocol combining the PSI protocol of Baldi *et al.* [5] with Cuckoo filter [42]. CF is a Cuckoo filter, CF.Insert is the insertion operation and CF.Check is the lookup operation, presented in Section 4.2.