Faster Unbalanced Private Set Intersection

Amanda Cristina Davi Resende and Diego de Freitas Aranha

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

Abstract. Protocols for Private Set Intersection (PSI) are important cryptographic primitives that perform joint operations on datasets in a privacy-preserving way. They allow two parties to compute the intersection of their private sets without revealing any additional information beyond the intersection itself. Unfortunately, PSI implementations in the literature do not usually employ the best possible cryptographic implementation techniques. This results in protocols presenting computational and communication complexities that are prohibitive, particularly in the case when one of the participants is a low-powered device and there are bandwidth restrictions. This paper builds on modern cryptographic engineering techniques and proposes optimizations for a promising one-way PSI protocol based on public-key cryptography. For the case when one of the parties holds a set much smaller than the other (a realistic assumption in many scenarios) we show that our improvements and optimizations yield a protocol that outperforms the communication complexity and the run time of previous proposals by around one thousand times.

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 genetic testing of fully-sequenced human genomes [5], private contact discovery [9], relationship path discovery in social networks [32], botnet detection [33] and proximity testing [36].

PSI protocols allow two parties storing a set of private data such as lists of patients, criminal suspects or telephone contacts to compute the intersection of their sets without revealing any additional information beyond the intersection to one or both parties. These 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 check [8,11,28].

PSI protocols can also be classified based on their set sizes. In the literature, Chen *et al.* [9] defined 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 primitive being used (symmetric or asymmetric): *balanced* for sets with approximately the same size and *unbalanced* for the opposite scenario[1].

However, even with several PSI protocols proposed in the literature, most real-word applications use naive solutions (as later detailed in Section 2.2). A reason for this is that some solutions are efficient when performing operations on small datasets, but become impractical for large sets. They may also be efficient in terms of execution time, but when performed in constrained environments with low bandwidth became impractical as they transmit too much data.

Protocols proposed and implemented in several papers by Pinkas *et al.* [40,42,43] 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 [5,9,23,31] need to transmit fewer data, but require less efficient operations. Thus, the choice of the protocol depends on the PSI setting, network bandwidth, storage space, security properties, among other factors.

1.1 Our contributions

Several of the PSI implementations available in the literature makes no use of modern and efficient techniques for the implementation of cryptographic protocols, mainly based on public-key cryptography. We aim at filling this gap by showing that the protocol previously proposed by Baldi *et al.* in [5] can be optimized as to reduce its communication and running time by a factor of at least three. Our implementation is available online at `http://github.com/amandadavi7/PSI`. In more detail, the main contributions of this paper are:

- **Improvements on the one-way PSI protocol based on public-key cryptography by Baldi *et al.* [5]**: We show that the version of the protocol secure against semi-honest adversaries becomes an efficient and practical one-way PSI for the unbalanced setting after our optimizations. Furthermore, if the optimized protocol is used in constrained scenarios, like 1 Mbps of network bandwidth, it remains a good choice for balanced one-way PSI too (Table 4 in Appendix B). Moreover, it satisfies a desired forward secrecy property 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.
- **We propose Cuckoo filters to reduce the amount of data to be exchanged by the protocol and stored by the client**: Cuckoo filters present many advantages: (i) they require less storage space than other similar approaches, like Bloom filter and Cuckoo hashing, for a false positive rate (FPR) less than 3% [14]; (ii) they allow the delete operation (important in some applications); and (iii) the lookup operation is performed in linear time in the number of entries per bucket. To the best of our knowledge, this is the first time that a Cuckoo filter is employed in PSI protocols, where normally a Bloom filter is used.

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

- **We provide an efficient software implementation of the protocols using the Galbraith-Lin-Scott binary elliptic curve (GLS-254) with point compression**: To the best of our knowledge, this is the first time that a state-of-the-art implementation of elliptic curves is used to instantiate PSI protocols that rely on this type of operation. Our implementation of the GLS-254 curve takes around 50,000 cycles to compute an exponentiation, which is 24× faster than the 283-bit Koblitz curve implementation used, for example, in the PSI protocol presented in [41].
- **Experimental comparison**: We implemented the original version [5] and our optimized protocol (both using the GLS-254 curve) and compared them with the most promising PSI protocols in the literature, showing the results of the (offline) preprocessing phase (when it is possible) and the online phase. Our results show that with our optimizations, this protocol is efficient even when used in bandwidth restricted scenarios.

1.2 Application to private contact discovery

In the private contact discovery problem, a user signs up to a messaging application such as WhatsApp, Signal or Telegram, and would like to discover which contacts in his/her address book are also registered. However, the user is not willing to reveal his entire list of contacts. In this setting, the user typically has a set with a few hundred contacts, while the messaging service can have from a few million to a few billion users, characterizing the unbalanced setting.

Because of the sheer number of entries in the social network server's side, secure messaging applications such as TextSecure/Signal[2] and Secret[3], currently employ naive approaches (see Section 2.2) to "solve" the private contact discovery problem, since they have both better run time and communication complexity when compared to state-of-the-art secure protocols. Signal is also experimenting with the Intel SGX, a trusted execution environment, in order to improve the security of private contact discovery[4].

At the cost of tolerating a small FPR, our optimized protocol provides a secure solution that works in this realistic scenario, being potentially useful for social networks with millions of users.

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

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

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

[4] `https://signal.org/blog/private-contact-discovery/`

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 in 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 has bit-length σ.
- For a set S, the notation $x \xleftarrow{R} S$ indicates that x was sampled from S with uniform distribution.
- The operation $a \stackrel{?}{=} b$ denotes the comparison whether a is equal or not to b.
- $\kappa = 128$ is the security parameter.
- $\rho = 40$ is the statistical security parameter, i.e., the probability of a hash collision is less than $2^{-\rho}$.
- $\varphi = 256$ is the size of the representation of a point in the GLS-254 binary elliptic curve when using point compression (number of bits to store one x-coordinate and two trace bits).
- $\mathbb{G}$ is a multiplicative group of prime order q.
- $H : \{0,1\}^* \rightarrow \{0,1\}^l$, $H_1 : \{0,1\}^\sigma \rightarrow \mathbb{G}$, $H_2 : \mathbb{G} \rightarrow \{0,1\}^l$ are hash functions modeled as random oracles in the security analysis. In some cases, the output length is defined as $l = \rho + \log n_1 + \log n_2$ bits ($l' = \lceil l/8 \rceil$ bytes), as suggested by Pinkas *et al.* [43], instead of $2 \cdot \kappa$. This produces the collision probability $2^{-\rho}$, 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, $\epsilon_{max} = 1 - (1 - \frac{1}{2^v})^{2b}$ as the upper bound on the false positive rate (FPR) and ϵ as the observed FPR, both given in %.

2.2 Classification and related work of PSI protocols

Many one-way PSI protocols have been proposed in the open research literature [9,12,19,22,25,40,42,43]. They are constructed based on several primitives such as Bloom filters [6], Homomorphic Encryption [15,20], Oblivious Pseudorandom Function (OPRF) [18], Unpredictable Function [25], Oblivious Transfer (OT) [29,35], Oblivious Polynomial Evaluation (OPE) [34], Cuckoo hashing [39], Garbled Circuits (GC) [44,45], among others.

Following Pinkas *et al.* [40,42,43], 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 to compute the hash of their elements. P_1 then computes $x'_i = H(x_i)$ while P_2 computes $y'_j = H(y_j)$, where $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$. After computing the hashes, P_1 sends values x'_i to P_2 which computes the intersection between sets x' and y' by checking if $y'_j \stackrel{?}{=} x'_i$ for each value of i and j. This approach is very efficient both in run time and communication. P_1 and P_2 must compute n_1 and n_2 hash functions, respectively. 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 $O(1)$ per element, in average, by using a hash table.

However, if the hash function inputs were taken from a low-entropy domain $\mathbb{D}$, P_2 can discover all elements of P_1 by performing a brute-force attack. For every possible element $z \in \mathbb{D}$, P_2 verifies if $H(z) \stackrel{?}{=} x'_i$. 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 $H(z) \stackrel{?}{=} x'_i$. Nonetheless, this 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 [11,12,26] 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 [26], Kamara *et al.* present a semi-honest server protocol that takes 10 minutes with 12 GB of communication and 100 threads to evaluate 2 sets of 1 billion elements each. In the protocol, P_1 samples a random k-bit key K and sends it to P_2 (in this approach both P_1 and P_2 are considered clients). Both P_1 and P_2 compute $f_1 = \pi_1(F_K(x_1), ..., F_K(x_i))$ and $f_2 = \pi_2(F_K(y_1), ..., F_K(y_j))$, respectively, where $F : \{0,1\}^k \times \mathbb{D} \rightarrow \{0,1\}^{\geq k}$ 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 computes 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 [22], Huang *et al.* presented several of these protocols using Boolean circuits, all of them constructed using Yao's garbled circuits [44,45].

The simplest protocol described in [22] involves the comparison of each element from P_1 with each from P_2. This approach is known as Pairwise-Comparison (PWC) and involves $O(n^2)$ comparisons, which does not scale well for large sets. Another more efficient approach presented in [22], the Sort-Compare-Shuffle (SCS) circuit, is more efficient, with complexity $O(n \log n)$. SCS first sorts the union of P_1 and P_2 elements, then compares if adjacent elements are the same, and finally shuffles the result to prevent information leakage.

The major 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 a threshold. However, despite the improvements in recent years, they still have a very high run time compared to others.

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.* [12], combining Bloom filters and OT [24] in their construction.

In 2014, Pinkas *et al.* [42] presented improvements to [12] and also proposed a new and more efficient protocol combining OT and hashing. In 2015, they have shown [40] that their previously proposal [42] could be improved by using the permutation-based hashing technique [3], 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.* [43] presented improvements for their earlier protocols, where the complexity no longer depends on the size of each element. This solution 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 solution is extremely efficient[8].

Public-key cryptography based PSI. Meadows [31] and Huberman *et al.* [23] proposed the earliest PSI approaches based on public-key cryptography, even before the PSI problem was formally defined in [19]. Both protocols were based on the commutative properties of the Diffie-Hellman (DH) key exchange. DH-based PSI protocols use the Random Oracle Model (ROM) to prove their security, while Freedman *et al.* [17,19] introduced PSI protocols based on the standard model, which are secure against both semi-honest and malicious adversaries and are based on the ElGamal cryptosystem. In [10], Cristofaro and Tsudik presented a PSI protocol based on blind-RSA.

[8] The protocol presented in [43] uses asymmetric operations [35] 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.

Later Jarecki *et al.* [25] presented a PSI protocol secure against malicious adversaries based on a Parallel Oblivious Unpredictable Function (POUF). In [5], Baldi *et al.* relaxed the security of [25] to semi-honest adversaries. Another relevant public-key PSI protocol was proposed by Chen *et al.* [9] and is based on the protocol presented by Pinkas *et al.* [43], but instead of performing OPRF (via OT) operations, it uses the Fan-Vercauteren (FV) leveled Fully Homomorphic Encryption (FHE) scheme [15]. This change considerably decreases the amount of data to be transmitted in the unbalanced setting. Therefore, depending on the setting and the network bandwidth, Chen *et al.* [9] is faster than [43]. The good performance is however restricted to 32-bit elements due to limitations in the parameters of the FHE scheme. The protocol proposed in [5] was used to obtain the results in this paper.

The most recent work was presented by Kiss *et al.* [27]. They independently 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 future executions to compute the intersection. They proposed using 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 protocol closest to our optimized proposal (DH-based PSI [23,31]), 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 server and the client compute n_1 exponentiations. For example, if $n_1 = 2^{24}$ it will be necessary to transmit 568 MB and to perform 2^{24} exponentiations on the server and client side.

One efficient way to instantiate PSI protocols based on public-key cryptography is to use elliptic curves. The exponentiation on elliptic curves becomes a scalar multiplication, but we will keep the exponentiation notation throughout the paper for compatibility with the other works.

3 The basic protocol

Jarecki and Lui [25] presented a one-way 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). Later, Baldi *et al.* [5] relaxed the security of this protocol to be secure against semi-honest adversaries, by removing the ZKP. This protocol is shown in Figure 1 and works as follows: for each element $x_i \in X$, the server computes the hash $H_1(x_i)$, the exponentiation $H_1(x_i)^\alpha$ with the same exponent for all the elements, and again computes the hash $tx_i = H_2(H_1(x_i)^\alpha)$, sending values tx_i to the client. For each element $y_j \in Y$, the client computes the hash $H_1(y_j)$, the exponentiation $a_j = H_1(y_j)^{\beta_j}$ with ephemeral exponents β_j and sends values a_j to the server. The server computes $a'_j = (a_j)^\alpha$ for each a_j using the same α used previously and sends values

a'_j to the client. The client then computes $ty_j = H_2((a'_j)^{1/\beta_j})$, by "removing" the exponents that were applied earlier. Finally, the client computes the intersection by checking if $ty_j \in \{tx_1, tx_2, ..., tx_{n_1}\}$. The long-term secret is the server key α.

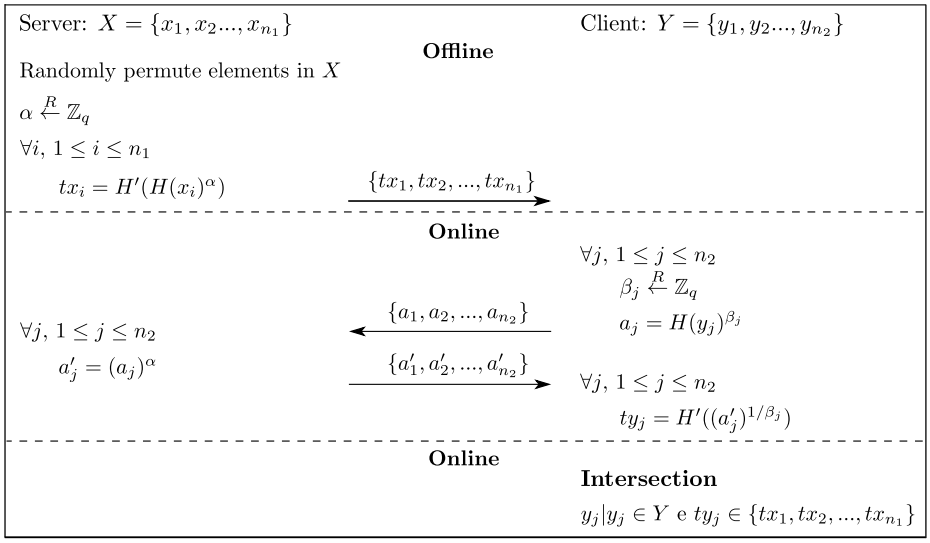

Fig. 1. Basic PSI protocol proposed in [5] that relaxes the security of [25] to be secure against semi-honest adversaries. H_1 and H_2 are hash functions modeled as random oracles. [5, Adapted].

4 Optimizations

We propose a few modifications to the protocol presented in Section 3 that drastically improve its performance during the preprocessing and online phase.

(i) The offline phase is executed just once and the results stored in a database. We significantly reduce the size of the database by using a Cuckoo filter [14].
(ii) We implement the protocol based on the GLS-254 elliptic curve, which improves its computational performance.

Below these improvements are described in detail.

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_1 and H_2 ($tx_i = H_2(H_1(x_i)^\alpha)$) before receiving connections.

Because of this feature, the offline part can be performed only once, where the server would compute 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 needs to be used. The resulting protocol is very efficient when used in unbalanced PSI setting 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 144 MB. However, if the scale changes from a few million to a few billion, as is the case of a large messaging application with approximately 2^{30} users, the server masked elements would need 10 GB 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 have been used previously in the literature like Bloom filters and their variants [6,7,16] and Cuckoo hashing [39].

We take a different approach and propose to use Cuckoo filters [14]. They have clear advantages over Bloom filters and Cuckoo hashing, since they allow the delete operation (essential in private contact discovery), besides the insertion and lookup operations, using significantly less space than the Bloom filter variants and the Cuckoo hashing by storing only the element's fingerprint. For the two examples given above, a Cuckoo filter would use 48 MB and 3 GB, respectively[9].

We give the necessary preliminaries about Cuckoo filter in Appendix A, and further on assume that the reader is familiar with the notion of Cuckoo filter. To the best of our knowledge, this is the first application of Cuckoo filters to the problem of private set intersection.

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 [37] available in SUPERCOP[10]. 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 [38] and the GLS endomorphism for fast scalar multiplications [21], 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 vector instruction sets; a regular

[9] These values may change if the FPR changes. Here we set $\epsilon_{max} = 0.009155\%$.

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

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 available in the first layer.

The approach selected for hashing was a combination of the SHA256 hash function with the binary Shallue-van de Woestijne well-bounded encoding algorithm [2]. 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 [30]. 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 concatenating 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. Our entire code runs in constant time for side-channel resistance, including the quadratic solver [2].

4.4 Our optimized protocol

Figure 2 presents our optimized protocol based on Baldi *et al.* [5]. In the first part (offline), the server generates a Cuckoo filter, from his/her masked elements, using the insertion operation (*CF.Insert*), and sends the filter (*CF*) to the client. The online part is divided in two: in the first, client and server interact in order to mask the elements of the client. In the second, the client with his/her masked elements, checks if each one of them belongs to the filter, through the lookup operation (*CF.Check*), thus computing the set intersection. The last part of the protocol is the step of updating the filter. The server has a set of elements Z that he/she would like to insert, such as new users of a messaging application; or to delete, in the case where some users no longer use the service. In both cases, the server masks each element $z_k \in Z$ using α and sends them to the client. Along with these values, a variable is also sent to tell the client what is the type of update, if it is an insertion or a deletion.

In the insertion operation, the user must first check the load factor w of the filter. If it is greater than 0.95, the user must request the server to generate a new filter using all the elements, as in the first part of the protocol. Otherwise, the client inserts the element in the filter *CF*. The value 0.95 was set in [14] to be the highest w for the filter to have high space and lookup efficiency. After that, it is hard to insert elements without errors. Therefore, when w is greater than 0.95, a new and larger filter must be generated. In the case of deletion, the element is removed from the filter without the need to generate a new one.

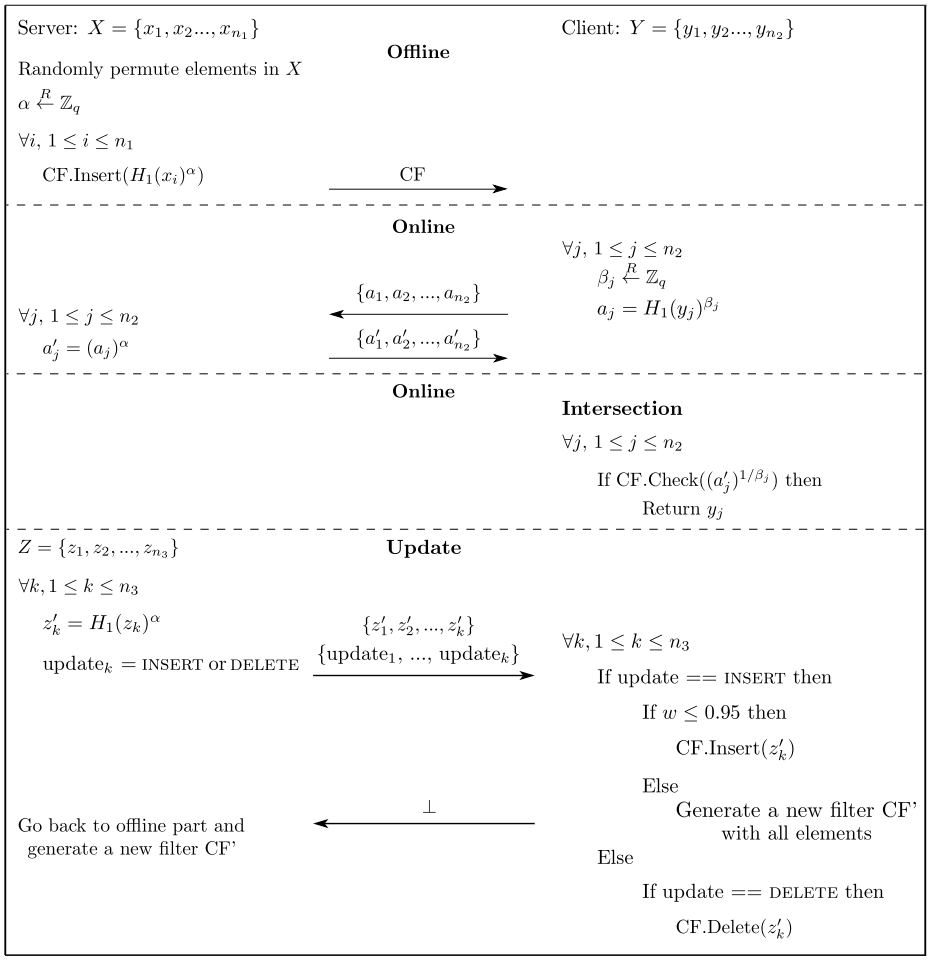

Fig. 2. Our optimized protocol combining the PSI protocol of Baldi *et al.* [5] with Cuckoo filter [14]. CF is a Cuckoo filter, *CF.Insert* is the insertion operation, *CF.Check* is the lookup operation and *CF.Delete* is the deletion operation.

4.5 Correctness and security guarantees with our modifications

The correctness guarantees of the protocol with the modifications shown above follow from the correctness of the original protocol by Jarecki and Lui [25] and Baldi *et al.* [5] and the correctness of the Cuckoo filter (up to false positives) [13,14]. This happens due to the fact that the same messages are exchanged and the same computational steps are performed by both parties, with the only difference that the server's masked elements are encoded in a Cuckoo filter.

The FPR of the Cuckoo filter can be as small as the application requires considering the cost of increasing the filter size. We have 2 different FPR: ϵ_{max} and ϵ. The first is an upper bound and does not take into account the load factor

of the filter, and the second is the observed measure that takes into account the load factor (for more details see the Appendix A).

The security guarantees of the modified protocol also follow from the security of the original protocol, which is based on the hardness of the OMGDH problem. We apply only one modification to the protocol that does not change the security: we replace the transmission of the server's masked elements to the client, in the offline phase, by sending a Cuckoo filter which encodes the masked elements.

However, sending the Cuckoo filter does not reveal any more information than sending the masked elements. By assumption, there is an algorithm $\mathcal{A}$ that the attacker can use on the modified protocol (with the Cuckoo filter) and breaks security with non-negligible probability γ. It is possible to devise the algorithm $\mathcal{A}'$ with which the attacker can use to break the original protocol and works as follows: first $\mathcal{A}'$ runs the original protocol and keeps the raw set of masked elements received from the server. Then, the attacker encodes the masked elements as a Cuckoo filter and feeds $\mathcal{A}$ with it. Therefore, $\mathcal{A}$ observes the same view as in a run of the modified protocol and thus can break security with probability γ. Thus, $\mathcal{A}'$ breaks the security with the same probability.

We have preserved a weak notion of forward secrecy on the client side, as provided in the original version [5]. This weak notion ensures that elements exchanged in the past will remain confidential even if long-term keys are exposed. In the case of private contact discovery, where the client set is always almost the same, compromising the client once reveals almost all the contacts used in previous executions, so forward secrecy does not provide much advantage. However, when the client set changes from one execution to another, as in the case of other applications, having forward secrecy is important. One such application is malware detection, where the client set may store networking data collected during a time interval. Modifying the protocol to relax forward secrecy allows precomputation in the client side which potentially improves its performance.

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 C). For the Local Area Network (LAN) setting, the two parties (client and server) are connected via local host with 10 Gbps of bandwidth and a 0.2 ms 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 80 ms RTT. These settings follow what was proposed by Chen *et al.* [9].

We evaluate the performance of the PSI protocols in the unbalanced setting and in the Appendix B we show the performance of the PSI protocols

in the balanced setting. 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.* [9]. The size of each element was set to be $\sigma = 32$ bits, but this does not impact the performance of our protocol due to hashing. The output of the hash function used 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 [43,9].

5.2 Implementation

The implementation of OT+Hashing [43] was obtained from Pinkas *et al.* [43], available at `https://github.com/encryptogroup/PSI`. They used OpenSSL (v.1.0.1e) for the symmetric cryptographic primitives, the implementation of [4] and the code available at `https://github.com/encryptogroup/OTExtension` for the OT extension, and the MIRACL library (v.5.6.1) for ECC. According to our benchmarking, an exponentiation within their codebase takes 1.2 million cycles[11], which indicates a misconfigured version of MIRACL. Up to now, there is no implementation available for the Chen *et al.* [9] protocol, but we tried to reproduce the benchmarking scenarios as close as possible to their work.

We implemented our optimized protocol and the original [5] using the software provided by Pinkas *et al.* [43] replacing the NIST K-283 curve, available in MIRACL, with the GLS-254 curve. Our implementation of the GLS-254 curve takes around 50,000 cycles to compute an exponentiation, which is 24× faster than [43]. We note, however, that the K-283 curve is a more conservative choice of parameters. We used the Cuckoo filter implementation of Fan *et al.* [14] available at `https://github.com/efficient/cuckoofilter`. Our implementation is available at `http://github.com/amandadavi7/PSI`.

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 [43] 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 two 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.* [9], 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 Chen *et al.* [9], in the basic protocol presented in Section 3, the server can preprocess the encryption/masking of all elements and send them to the

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

client, which must store and reuse this data in all subsequent executions to compute the intersection, as shown 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 the beginning of Section 5. The run times of Chen *et al.* were obtained from their own paper [9], 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.

We note that the run times for the protocol proposed by Baldi *et al.* in [5] here presented, are for an implementation based on the binary elliptic curve GLS-254 and not for the original implementation proposed in [5] which works over a 1024-bit prime number. The improvements would be way more drastic had the original implementation proposed in [5] been used for comparison.

It is interesting to note that the preprocessing run times of our optimized protocol and our implementation of the original protocol 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 optimized version transmits up to 3.3× less data than [5] and is accordingly 3.3× faster.

5.4 Comparison to others PSI protocols

The performance evaluation of the protocols was performed in the unbalanced setting. Since the code for Chen *et al.* [9] was not made available, we obtained results for this protocol and OT+Hashing [43] from [9]. As shown in Section 2.2, PSI protocols can be classified into five categories. Because the naive hashing and server-aided approaches have different security notions from the others, they are not included. PSI based on generic protocols are out of scope, because they have limitations in run time and memory. Among the two remaining categories, OT-based PSI and PSI based on public-key, we analyze the best protocol in each category comparing the results with our optimized 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. 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 optimized proposal.

Amongst the public-key protocols, our optimized proposal and the Baldi *et al.* [5], in the online phase, have the same communication cost ($2n_2\varphi$ bits) and,

					Transmission time (s)			
			Comm.	Preprocessing	LAN	WAN		
Protocol	n_1	n_2	Size (MB)	Time (s)	10 Gbps	100 Mbps	10 Mbps	1 Mbps
Chen *et al.* [9]	2^{24}	11041	-	**70.90, *76.80**	-	-	-	-
		5535	-	**64.10, *71.20**	-	-	-	-
	2^{20}	11041	-	**6.40**	-	-	-	-
		5535	-	**4.30**	-	-	-	-
	2^{16}	11041	-	**1.00**	-	-	-	-
		5535	-	**0.70**	-	-	-	-
Baldi *et al.* [5]	2^{24}	11041	160.00	334.17	0.13	15.73	136.32	1,345.55
		5535						
	2^{20}	11041	10.00	20.91	0.01	1.10	8.38	84.40
		5535						
	2^{16}	11041	0.56	1.31	0.01	0.19	0.53	5.09
		5535						
Our protocol	2^{24}	11041	**48.00**	333.62	**0.06**	**4.82**	**40.71**	**403.68**
		5535						
	2^{20}	11041	**3.00**	20.78	**0.00**	**0.60**	**2.55**	**25.63**
		5535						
	2^{16}	11041	**0.19**	1.30	**0.00**	**0.01**	**0.19**	**1.56**
		5535						

Table 1. Preprocessing and transmission time for PSI protocols in the (offline) preprocessing phase. The WAN setting has 80 ms RTT and the LAN 0.02 ms RTT. For the filter in our optimized protocol we have $v = 16$, $b = 3$, $w = 0.66$ and $\epsilon_{max} = 0.009155\%$. More details about Cuckoo filter is given in Appendix A. Chen *et al.* [9] does not have transmission time, since only the server will use the precomputed data. Zero values refer to numbers smaller than $5 \cdot 10^{-3}$ Best values marked in bold.

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 $O(b)$ per element. Because of this, we omitted the figures related to Baldi *et al.* protocol [5] in Table 2. In addition, comparing with the Chen *et al.* protocol[12] [9], our optimized approach transmits up to 59× less data and is up to 76× faster with 10 Gbps, for $n_2 = 5535$ and $n_1 = 2^{24}$. Comparing our optimized protocol with OT+Hashing [43], our approach transmits up to $1,413\times$ less data and is up to 74× faster with 10 Gbps of bandwidth and 946× faster with 1 Mbps, for $n_2 = 5535$ and $n_1 = 2^{24}$.

Our optimized approach performs well in unbalanced scenarios because our operations depend only on the client set size, with $2n_2\varphi$ bits transmitted and $3n_2$

[12] In the communication column of Table 2, the protocol [9] can have 2 different values, because according to the networking setting it is better that operations take more time and generate less data than taking less time but producing more data. This trade-off can be changed in FHE by adjusting the system parameters.

					Transmission time (s)			
		Parameters		Comm.	LAN	WAN		
Type	Protocol	n_1	n_2	Size (MB)	10 Gbps	100 Mbps	10 Mbps	1 Mbps
OT	OT + Hashing [43]	2^{24}	11041	480.90	40.50	88.00	449.50	4,084.80
			5535	480.40	40.10	87.90	449.20	4,080.60
		2^{20}	11041	30.90	3.30	7.00	29.80	263.70
			5535	30.40	3.10	6.80	29.00	260.00
		2^{16}	11041	2.60	0.70	1.50	3.30	21.60
			5535	2.10	0.70	1.40	2.90	19.80
Public key	Chen *et al.* [9]	2^{24}	11041	23.20, *21.10	44.50	46.90	63.50	*214.00
			5535	20.10, *12.50	41.10	43.10	*49.10	*139.90
		2^{20}	11041	11.50	6.40	7.60	15.80	99.00
			5535	5.60	4.30	4.90	9.00	49.30
		2^{16}	11041	4.10	2.00	2.40	5.40	35.00
			5535	2.60	1.10	1.30	3.20	21.80
	Our optimized 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 and communication for unbalanced PSI protocols. In the communication column, the Chen *et al.* [9] may have two values due to different parameters used in the FHE system. For more information, see [9]. The results of OT+Hashing [43] and Chen *et al.* [9] were obtained from [9]. Best values marked in bold.

exponentiations. Although exponentiations are considered an expensive operation, when performed a small number of times and with an efficient elliptic curve implementation, a curve-based protocol becomes competitive with the others.

Comparison with Kiss *et al.* [27]. In a recent paper, Kiss *et al.* [27] present many PSI protocols, where the closest to our proposal is ECC-DH-PSI [23,31]. 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 resource constrained device. Considering $n_1 = 2^{20}$ and according to [27], the preprocessing takes 1,325 s while our proposal takes 21 s (using a 1 Gbps network), which is 63× faster. Moreover, the server sends $n_1\varphi$ ($\varphi = 284$ in their case), that adds up to 35.5 MB, while our proposal just sends a 2.125 MB filter for $\epsilon = 0.05\%$ ($v = 16$, $w = 0.94$ and $b = 17$) and a 5 MB filter for $\epsilon = 1.6 \times 10^{-7}\%$ ($v = 32$, $w = 0.8$ and $b = 5$)[13]. This is 16.7× and 7.1× less data to be transmitted, respectively.

[13] The filter in the client side from [27] has $\epsilon = 0.1\%$ and $\epsilon = 10^{-7}\%$, respectively.

In the online phase the amount of data to be transmitted is asymptotically the same, $2n_2\varphi$ bits, but concretely Kiss *et al.* [27] use $\varphi = 284$ bits for the K-283 curve with compression, and we have $\varphi = 256$ bits 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 [27] 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.* [27] use a Bloom filter, while our optimized approach employs a Cuckoo filter. The Cuckoo filter allows deletions while the traditional Bloom filter does not and uses 30% less space for the same FPR [13]. While counting Bloom filters do allow deletions, this happens at the cost of using 3-4× more space.

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 [27] 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 to allow two parties to perform joint operations on their private sets without revealing 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. In most approaches, the computational cost is based on both the server and client set sizes, giving no advantages in the unbalanced setting.

We show that the protocol of Baldi *et al.* [5] based on public-key cryptography, with our optimizations, becomes an efficient, practical and simple one-way PSI protocol 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 allow a better comparison with the other proposed approaches.

Our optimized protocol with this implementation provides an interesting trade-off between preprocessing and the online phase of the protocol, where for $n_1 = 2^{24}$ the preprocessing takes less than six minutes of computing time (recall that this phase needs to be done only once) and the online phase for $n_2 = 11041$ takes less than 8 seconds even with 1 Mbps bandwidth. The client needs to store only 48 MB of information for this configuration. We believe that our improved protocol is a practical alternative for the solutions currently in place for privacy-preserving contact discovery in existing social networks.

Acknowledgements

This work was in part supported by the Intel/FAPESP grant 14/50704-7, project "Secure Execution of Cryptographic Algorithms". We thank Anderson Nascimento and Fabian Monrose for discussion and comments on an earlier version.

References

1. Traffic shaping, bandwidth shaping, packet shaping with linux tc htb. `https://www.iplocation.net/traffic-control`, accessed: 2017-05-30
2. Aranha, D.F., Fouque, P., Qian, C., Tibouchi, M., Zapalowicz, J.: Binary Elligator Squared. In: SAC. LNCS, vol. 8781, pp. 20–37. Springer (2014)
3. Arbitman, Y., Naor, M., Segev, G.: Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation. In: FOCS. pp. 787–796. IEEE Computer Society (2010)
4. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In: ACM Conference on Computer and Communications Security. pp. 535–548. ACM (2013)
5. Baldi, P., Baronio, R., Cristofaro, E.D., Gasti, P., Tsudik, G.: 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. Bloom, B.H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Commun. ACM **13**(7), 422–426 (1970)
7. Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An Improved Construction for Counting Bloom Filters. In: ESA. LNCS, vol. 4168, pp. 684–695. Springer (2006)
8. Camenisch, J., Zaverucha, G.M.: Private Intersection of Certified Sets. In: Financial Cryptography. LNCS, vol. 5628, pp. 108–127. Springer (2009)
9. Chen, H., Laine, K., Rindal, P.: Fast Private Set Intersection from Homomorphic Encryption. In: CCS. pp. 1243–1255. ACM (2017)
10. Cristofaro, E.D., Tsudik, G.: Practical Private Set Intersection Protocols with Linear Complexity. In: Financial Cryptography. LNCS, vol. 6052, pp. 143–159. Springer (2010)
11. Debnath, S.K., Dutta, R.: Towards Fair Mutual Private Set Intersection with Linear Complexity. Security and Communication Networks **9**(11), 1589–1612 (2016)
12. Dong, C., Chen, L., Wen, Z.: 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)
13. Eppstein, D.: Cuckoo Filter: Simplification and Analysis. In: SWAT. LIPIcs, vol. 53, pp. 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
14. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.: Cuckoo Filter: Practically Better Than Bloom. In: CoNEXT. pp. 75–88. ACM (2014)
15. Fan, J., Vercauteren, F.: Somewhat Practical Fully Homomorphic Encryption. IACR Cryptology ePrint Archive (2012)
16. Fan, L., Cao, P., Almeida, J.M., Broder, A.Z.: Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. IEEE/ACM Trans. Netw. **8**(3), 281–293 (2000)
17. Freedman, M.J., Hazay, C., Nissim, K., Pinkas, B.: Efficient Set Intersection with Simulation-Based Security. J. Cryptology **29**(1), 115–155 (2016)
18. Freedman, M.J., Ishai, Y., Pinkas, B., Reingold, O.: Keyword Search and Oblivious Pseudorandom Functions. In: TCC. LNCS, vol. 3378, pp. 303–324. Springer (2005)

19. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient Private Matching and Set Intersection. In: EUROCRYPT. LNCS, vol. 3027, pp. 1–19. Springer (2004)
20. Gentry, C.: Fully Homomorphic Encryption using Ideal Lattices. In: STOC. pp. 169–178. ACM (2009)
21. Hankerson, D., Karabina, K., Menezes, A.: Analyzing the Galbraith-Lin-Scott Point Multiplication Method for Elliptic Curves over Binary Fields. IEEE Trans. Computers **58**(10), 1411–1420 (2009)
22. Huang, Y., Evans, D., Katz, J.: Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In: NDSS. The Internet Society (2012)
23. Huberman, B.A., Franklin, M.K., Hogg, T.: Enhancing Privacy and Trust in Electronic Communities. In: EC. pp. 78–86 (1999)
24. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending Oblivious Transfers Efficiently. In: CRYPTO. LNCS, vol. 2729, pp. 145–161. Springer (2003)
25. Jarecki, S., Liu, X.: Fast Secure Computation of Set Intersection. In: SCN. LNCS, vol. 6280, pp. 418–435. Springer (2010)
26. Kamara, S., Mohassel, P., Raykova, M., Sadeghian, S.S.: Scaling Private Set Intersection to Billion-Element Sets. In: Financial Cryptography. LNCS, vol. 8437, pp. 195–215. Springer (2014)
27. Kiss, A., Liu, J., Schneider, T., Asokan, N., Pinkas, B.: Private Set Intersection for Unequal Set Sizes with Mobile Application. PoPETs **2017**(4), 97–117 (2017)
28. Kissner, L., Song, D.X.: Privacy-Preserving Set Operations. In: CRYPTO. LNCS, vol. 3621, pp. 241–257. Springer (2005)
29. Kolesnikov, V., Kumaresan, R.: Improved OT Extension for Transferring Short Secrets. In: CRYPTO (2). LNCS, vol. 8043, pp. 54–70. Springer (2013)
30. Lopez, J., Dahab, R.: New Point Compression Algorithms for Binary Curves. In: IEEE Information Theory Workshop - ITW '06. pp. 126–130 (March 2006). https://doi.org/10.1109/ITW.2006.1633795
31. Meadows, C.A.: 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)
32. Mezzour, G., Perrig, A., Gligor, V.D., Papadimitratos, P.: Privacy-Preserving Relationship Path Discovery in Social Networks. In: CANS. LNCS, vol. 5888, pp. 189–208. Springer (2009)
33. Nagaraja, S., Mittal, P., Hong, C., Caesar, M., Borisov, N.: BotGrep: Finding P2P Bots with Structured Graph Analysis. In: USENIX Security Symposium. pp. 95–110. USENIX Assoc. (2010)
34. Naor, M., Pinkas, B.: Oblivious Transfer and Polynomial Evaluation. In: STOC. pp. 245–254. ACM (1999)
35. Naor, M., Pinkas, B.: Efficient Oblivious Transfer Protocols. In: SODA. pp. 448–457. ACM/SIAM (2001)
36. Narayanan, A., Thiagarajan, N., Lakhani, M., Hamburg, M., Boneh, D.: Location Privacy via Private Proximity Testing. In: NDSS. The Internet Society (2011)
37. Oliveira, T., Aranha, D.F., Hernandez, J.L., Rodríguez-Henríquez, F.: Improving the performance of the GLS254. CHES Rump Session (2016)
38. Oliveira, T., López, J., Aranha, D.F., Rodríguez-Henríquez, F.: Two is the Fastest Prime: Lambda Coordinates for Binary Elliptic Curves. J. Cryptographic Engineering **4**(1), 3–17 (2014)
39. Pagh, R., Rodler, F.F.: Cuckoo Hashing. In: ESA. LNCS, vol. 2161, pp. 121–133. Springer (2001)

40. Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: Private Set Intersection Using Permutation-based Hashing. In: USENIX Security Symposium. pp. 515–530. USENIX Assoc. (2015)
41. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure Two-Party Computation Is Practical. In: ASIACRYPT. LNCS, vol. 5912, pp. 250–267. Springer (2009)
42. Pinkas, B., Schneider, T., Zohner, M.: Faster Private Set Intersection Based on OT Extension. In: USENIX Security Symposium. pp. 797–812. USENIX Assoc. (2014)
43. Pinkas, B., Schneider, T., Zohner, M.: Scalable Private Set Intersection Based on OT Extension. ACM Trans. Priv. Secur. **21**(2), 7:1–7:35 (2018)
44. Yao, A.C.: Protocols for Secure Computations (Extended Abstract). In: FOCS. pp. 160–164. IEEE Computer Society (1982)
45. Yao, A.C.: How to Generate and Exchange Secrets (Extended Abstract). In: FOCS. pp. 162–167. IEEE Computer Society (1986)

Appendix A - Cuckoo filter

A Cuckoo filter is a compact variant of a Cuckoo hashing proposed by Fan *et al.* [14] that stores only the element's fingerprint and consists of a set of buckets each one 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 check [13,14].

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 f' are randomly selected, and then the fingerprint f is inserted in place of f' and a new bucket $b_3 = b_2 \oplus h(f')$ is computed for f'. Note that, b_3 is equal to b_1 because the operation $\oplus$ guarantees that b_1 can be computed from b_2 and the fingerprint f'. 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.

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, compute the element fingerprint, the two possible buckets, check if any of the entries correspond to fingerprint and if so, a copy (two distinct elements can share the same bucket and the same fingerprint) 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.

The Cuckoo filter (or any similar approach) introduces a FPR which is upper bounded by $\epsilon_{max} = 1-(1-\frac{1}{2^v})^{2\times b}$, according to Fan *et al.* [14]. This bound does not take into account the load factor of the filter. The FPR increases as the filter

becomes more occupied, thus $0 \leq \epsilon \leq \epsilon_{max}$. For 16-bit fingerprints ($v = 16$) and 3 entries per buckets ($b = 3$) we have $0 \leq \epsilon \leq 0.009155\%$. The Table 3 shows the observed FPRs, for the same values of v, b and ϵ_{max} with load factor of 66.6% ($w = 0.66$). The observed FPRs are the average of 1000 executions of a Cuckoo filter, where each one was filled with random values and the queries to check the FPR were also random values. As can be seen in Table 3, the observed FPR ϵ ranges from 0.00508% to 0.00669% for a load factor of 66.6%, which is 56% to 73% in relation to $\epsilon_{max} = 0.009155\%$.

	Number of queries							
n_1	5535	11041	2^8	2^{12}	2^{16}	2^{20}	2^{24}	Number of buckets
2^{24}	0.00618	0.00649	-	-	-	-	0.00611	$2^{23} = 8,388,608$
2^{20}	0.00669	0.00619	-	-	-	0.00610	-	$2^{19} = 524,288$
2^{16}	0.00558	0.00627	-	-	0.00605	-	-	$2^{15} = 32,768$
2^{12}	-	-	-	0.00559	-	-	-	$2^{11} = 2,048$
2^8	-	-	0.00508	-	-	-	-	$2^7 = 128$

Table 3. Observed FPR (in %) and the number of buckets (m) for the Cuckoo filter presents in the Section 5.3 for 16-bit fingerprints ($v = 16$), 3 entries per buckets ($b = 3$), $\epsilon_{max} = 0.009155\%$ and load factor of 66.6% ($w = 0.66$).

Appendix B - Balanced PSI protocols

Following as Section 5, DH-ECC [23,31] implementation was obtained from Pinkas *et al.* [43], available at `https://github.com/encryptogroup/PSI`. For the balanced scenario, $n_1 = n_2 \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$ as proposed by Pinkas *et al.* [40,43]. The size of each element was set to be 32 bits ($\sigma = 32$ bits). The output of the hash function used in the DH-ECC [23,31] is l, as defined in the Section 2.1.

Table 4 presents the run time (in seconds) and the communication (in MBs) of the balanced scenario considering both a LAN and a WAN setting as defined in the Section 5.1. The results show, as expected, that the OT+Hashing protocol [43] has the best run time, being 9.5× faster than DH-ECC [23,31] and 14.5× faster than our optimized approach with 10 Gbps. This is due to the fact that OT+Hashing [43] 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 [23,31] is approximately 35% faster than the proposed approach with 10 Gbps. This happens because in DH-ECC [23,31] is possible to perform client and server operations in parallel, while in our optimized proposal it is not possible (as shown in Figure 1). However, the DH-ECC [23,31] and our protocol exchange 32% and 42% less data, respectively, than OT+Hashing [43]. For the WAN setting with 1 Mbps of bandwidth, both approaches are faster than

OT+Hashing [43] by transmitting less data and our optimized protocol is slightly faster than DH-ECC [23,31], because we transmit 10% 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 [43]	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 [31,23]	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 optimized 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 4. Run time in seconds and communication in MBs for balanced PSI protocols. Times are taken at the client because it finishes last. The WAN setting has 80 ms RTT and the LAN 0.02 ms RTT. For the filter in our optimized protocol we have 16-bit fingerprints ($v = 16$), 3 entries per buckets ($b = 3$), load factor of 66.6% ($w = 0.66$) and $\epsilon_{max} = 0.009155\%$. More details about Cuckoo filter is given in Appendix A. Best values marked in bold.

Appendix C - Network simulation

The simulation code was obtained from [1] 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
```