Fast Private Set Intersection from Homomorphic Encryption

Hao Chen[1], Kim Laine[2], and Peter Rindal[3]

[1] Microsoft Research, Redmond, WA, USA; haoche@microsoft.com
[2] Microsoft Research, Redmond, WA, USA; kim.laine@microsoft.com
[3] Oregon State University, Corvallis, OR, USA; rindalp@oregonstate.edu

Abstract. Private Set Intersection (PSI) is a cryptographic technique that allows two parties to compute the intersection of their sets without revealing anything except the intersection. We use fully homomorphic encryption to construct a fast PSI protocol with a small communication overhead that works particularly well when one of the two sets is much smaller than the other, and is secure against semi-honest adversaries.

The most computationally efficient PSI protocols have been constructed using tools such as hash functions and oblivious transfer, but a potential limitation with these approaches is the communication complexity, which scales linearly with the size of the larger set. This is of particular concern when performing PSI between a constrained device (cellphone) holding a small set, and a large service provider (e.g. *WhatsApp*), such as in the Private Contact Discovery application.

Our protocol has communication complexity linear in the size of the smaller set, and logarithmic in the larger set. More precisely, if the set sizes are $N_{\text{Y}} < N_{\text{X}}$, we achieve a communication overhead of $O(N_{\text{Y}} \log N_{\text{X}})$. Our running-time-optimized benchmarks show that it takes 36 seconds of online-computation, 71 seconds of non-interactive (receiver-independent) pre-processing, and only 12.5MB of round trip communication to intersect five thousand 32-bit strings with 16 million 32-bit strings. Compared to prior works, this is roughly a 38–115× reduction in communication with minimal difference in computational overhead.

1 Introduction

1.1 Private Set Intersection

Private Set Intersection (PSI) refers to a setting where two parties each hold a set of private items, and wish to learn the intersection of their sets without revealing any information except for the intersection itself. Over the last few years, PSI has become truly practical for a variety of applications due to a long list of publications, e.g. [PSZ14,PSSZ15,PSZ16,KKRT16,OOS17,RR16,Lam16,BFT16a,DCW13]. The most efficient protocols have been proposed by Pinkas *et al.* [PSZ16] and Kolesnikov *et al.* [KKRT16]. While these protocols are extremely fast, their communication complexity is linear in the sizes of both sets. When one set is significantly smaller than the other, the communication overhead becomes considerable compared to the non-private solution, which has communication linear in the size of the smaller set.

1.2 Fully Homomorphic Encryption

Fully homomorphic encryption is a powerful cryptographic primitive that allows arithmetic circuits to be evaluated directly on encrypted data, as opposed to having to decrypt the data first. Despite the basic idea being old [RAD78], the first construction was given only in 2009 by Craig Gentry [Gen09]. While the early fully homomorphic encryption schemes were impractical, in only a few years researchers managed to construct much more efficient schemes (e.g. [BV14,FV12,BGV12,LATV12,BLLN13,GSW13]), bringing practical applications close to reality [NLV11,GHS12,GBDL+16].

At first glance, it might seem easy to use fully homomorphic encryption to achieve a low communication cost in PSI. The party with smaller set sends its encrypted set to the other party, who evaluates the intersection circuit homomorphically, and sends back the encrypted result for the first party to decrypt. The total communication is only

$$2 \times \text{ ciphertext expansion } \times \text{ size of the smaller set.}$$

However, a naive implementation of the above idea will result in a very inefficient solution. The reason is that—for all known fully homomorphic encryption schemes—the computational cost not only grows with the size of the inputs (in this case, the sum of the two set sizes), but also grows rapidly with the *depth* of the circuit. Thus our main challenge is to come up with various optimizations to make the solution practical, and even faster than the state-of-the-art protocols in many scenarios. In short, we will show that it is possible to construct a fast fully homomorphic encryption based PSI protocol, with a low communication overhead.

1.3 Related Work

Meadows [Mea86] proposed one of the first secure PSI protocols, which was later fully described by Huberman, Franklin and Hogg in [HFH99]. This approach was based on public-key cryptography, and leveraged the multiplicative homomorphic property of Diffie-Hellman key exchange. While these schemes have relatively good communication cost, the running time can be prohibitive when the set sizes become large due to the need to perform modular exponentiation for every item in both sets several times.

Since [HFH99], several other paradigms have been considered. Freedman *et al.* [FNP04] proposed a protocol based on oblivious polynomial evaluation. This approach leveraged partially homomorphic encryption, and was later extended to the malicious setting in [DSMRY09,HN10,HN12]. Another approach was proposed by Hazay *et al.* [HL08], and was based on a so-called *Oblivious PRF*.

Recently, more promising approaches based on *Oblivious Transfer (OT)* have been invented [IKNP03,OOS17]. At the time, by far the most efficient scheme was introduced by Pinkas *et al.* [PSZ14], and later improved in [PSZ16,OOS17,KKRT16]. We will denote the two parties engaged in a PSI protocol by *sender* and *receiver*, and maintain that after the execution of the protocol, the receiver learns the intersection of the sets, whereas the sender learns no information. The high level idea of the OT-based protocols is that the receiver engages in many OTs with the sender, and obliviously learns a randomized encoding for each item in its set, without revealing which values were encoded. The sender can then encode its items locally, and send them to receiver, who computes a plaintext intersection on the encodings. Due to the encodings being randomized, they do not reveal any information beyond the intersection. One inherent property of this approach is that the communication is linear in both set sizes due to the need to encode and send all of the encodings. The approach we take is similar, except that we use fully homomorphic encryption in place of Oblivious Transfer.

In a concurrent work, Kiss *et al.* [KLS+17] extend the OT-based protocol to an online/offline setting, where the bulk of the communication can occur once during the offline phase, and then be reused. However, these benefits require the larger set to be static and the communication/storage for both parties be linear in the size of the larger set, making the deployment of such a protocol challenging in many real world scenarios, e.g. private contact discovery.

Yet another OT-based approach was introduced by Dong *et al.* [DCW13], which builds on a data structure known as a *Bloom filter*. This data structure allows efficient membership test by setting specific bits in a long bit array. Importantly, the bit-wise AND of two Bloom filters is itself a valid Bloom filter for the intersection of the two original sets. With a few modifications to this idea, a secure protocol can then be constructed by allowing one of the parties to learn the bit-wise AND of the two Bloom filters with the use of OT. This approach requires a greater amount of communication than the approach introduced by Pinkas *et al.* [PSZ14], and results in inferior performance.

A commonly cited solution for PSI is to use generic secure multi-party computation protocols to compute the intersection. Huang *et al.* [HEK12] was the first to implement such a protocol using garbled circuits, which [PSZ14] later improved, and provided an implementation. They showed that a garbled circuit approach requires significantly more communication compared to OT-based methods. For a more complete survey of practical approaches, we point the reader to [PSZ16].

A very efficient server-aided protocol has also been proposed by Kamara *et al.* [KMRS14]. In this setting, it is assumed that there exists a non-colluding server. The basic idea is that a random function is sampled between the two parties which is applied to the elements in their respective sets. These encodings are then sent to the server who reports the intersection. While conceptually simple and very efficient, the reliance on such as server is undesirable. Moreover, the communication complexity is linear in both set sizes.

In all of the above protocols it is assumed that the set sizes (or upper bounds) are made public at the beginning of the protocol. Ateniese *et al.* [ACT11] introduced a protocol based on RSA accumulators [BdM94], which relaxes this assumption by hiding the receiver's set size. This protocol works by having the receiver

construct and send an RSA accumulator for its set. The sender can then construct a response for each of its items, which allows the receiver to test whether they were contained in the RSA accumulator. An important property of the RSA accumulator is that its size is small, and independent of the receiver's set size. As such, this protocol is most interesting when the *receiver* has a set much larger than the sender. In a follow-up work Bradley *et al.* [BFT16b] extended this protocol to imposing an upper bound on the number of items in the accumulator, thereby preventing a so-called "full-domain attack" by the receiver.

1.4 Contributions and Roadmap

As our discussion has shown, all of the prior PSI protocols require both parties to encode and send data over the network that is proportional to their entire sets. However, the trivial insecure solution only requires the smaller set to be sent. We address this gap by constructing the first secure and practical PSI protocol with low communication overhead based on a leveled fully homomorphic encryption scheme.

Our basic protocol requires communication linear in the smaller set, achieving optimal communication that is on par with the naive solution. We then combine an array of optimizations to significantly reduce communication size, computational cost, and the depth of the homomorphic circuit, while only adding a logarithmic overhead to the communication. In summary, we

- Propose a basic PSI protocol based on fully homomorphic encryption;
- Combine various optimizations to vastly reduce the computational and communication cost;
- Use fine-tuned fully homomorphic encryption parameters for the homomorphic computation to avoid the costly *bootstrapping* operation [Gen09,GHS12], and to achieve good performance;
- Develop a prototype implementation in C++ and demonstrate a 38–115× reduction in communication over previous state-of-the-art protocols.

In Section 2 we review the setups and tools we use to build the protocol: the PSI setup and its definition of security, and preliminaries on (leveled) fully homomorphic encryption. In Section 3 we propose our basic strawman PSI protocol. Then, in Section 4, we apply optimizations to vastly improve the strawman protocol and make it practical. The formal description of the optimized protocol, along with a security proof, is presented in Section 5. In Section 6 we provide a performance analysis of our implementation, and compare our performance results to [PSZ16] and [KKRT16].

2 Preliminaries

2.1 Notations

Throughout this paper, we will use the notation $[n]$ to denote the set $\{1, ..., n\}$. The computational and statistical security parameters will be denoted by κ, λ, respectively. Other parameters include:

- $X, Y \subseteq \{0,1\}^\sigma$ are the sender's and receiver's sets, each of size N_X, N_Y;
- m denotes the size of a hash table, and d denotes the number of items to be inserted into a hash table;
- n, q and t denote the encryption parameters described in Section 2.3;
- h denotes the number of hash functions used for cuckoo hashing in Section 4.2;
- B denotes the bin size for the simple hashing scheme described in Section 4.2;
- ℓ denotes the windowing parameter described in Section 4.3;
- α denotes the partitioning parameter described in Section 4.3.

2.2 Private Set Intersection

We use standard notations and call the two parties engaging in PSI the *sender* and the *receiver*. The sender holds a set X of size N_X, and the receiver holds a set Y of size N_Y. Both sets consist of σ-bit strings. We always assume the set sizes N_X and N_Y are public. The ideal PSI functionality computes the intersection, outputs nothing to the sender, and $X \cap Y$ to the receiver. We construct a new protocol for PSI from fully homomorphic encryption, and prove it to be secure in the semi-honest security model, where both parties correctly follow the protocol, but may try to learn as much as possible from their view of the protocol execution.

Our protocol is particularly powerful when the sender's set is much larger than the receiver's set. Hence we assume $N_X \gg N_Y$ throughout the paper, even though the protocol works for arbitrary set sizes with no changes. More precisely, we achieve a communication complexity of $O(N_Y \log N_X)$. Also, we require only the sender to perform work linear in the larger set size N_X. Intuitively, the receiver encrypts and sends its set to the sender, who computes the intersection on homomorphically encrypted data by evaluating an appropriate comparison circuit. The output is then compressed to much smaller size using homomorphic multiplication, and sent back to the receiver for decryption. We note that the receiver only performs relatively light computation in the protocol, i.e. encryptions and decryptions of data linear in its set size N_Y. This is particularly useful when the receiver is limited in its computational power, e.g. when the receiver is a mobile device.

Private contact discovery One particularly interesting application for our PSI protocol is *private contact discovery*. In this setting, a service provider, e.g. *WhatsApp*, has a set of several million users. Each of these users holds their own set of contacts, and wants to learn which of them also use the service. The insecure solution to this problem is to have the user send the service provider their set of contacts, who then performs the intersection on behalf of the user. While this protects the privacy of the service provider, it leaks the user's private contacts to the service provider.

While PSI offers a natural solution to this problem, one potential issue with applying existing protocols to this setting is that both the communication and computation complexity for both parties is linear in the larger set. As a result, a user who may have only a few hundred contacts has to receive and process data linear in the number of users that the service has, resulting in a suboptimal protocol for constrained hardware, such as cellphones. This problem was initially raised in an article by Moxie Marlinspike from *Open Whisper Systems*—the company that developed the popular secure messaging app *Signal*—when they were trying to deploy PSI for contact discovery [Mar14]. Our PSI protocol addresses this issue by allowing the constrained devices to process and receive data that is linear in only their set size, and only logarithmic in the service provider's set size. Moreover, the major part of the computation can be performed by the service provider in a large data center, where processing power is relatively inexpensive, whereas the user only performs a light computation.

2.3 Leveled Fully Homomorphic Encryption

Fully homomorphic encryption schemes are encryption schemes that allow arithmetic circuits to be evaluated directly on ciphertexts, ideally enabling powerful applications such as outsourcing of computation on private data [RAD78,Gen09]. For improved performance, the encryption parameters are typically chosen to support only circuits of a certain bounded depth (*leveled* fully homomorphic encryption), and we use this in our implementation.

Many of the techniques and algorithms presented in this paper are agnostic to the exact fully homomorphic encryption scheme that is being used, but for simplicity we restrict to RLWE-based cryptosystems using power-of-2 cyclotomic rings of integers [LPR10]. In such cryptosystems the plaintext space is $\mathbb{Z}_t[x]/(x^n+1)$, and the ciphertext space is $\mathbb{Z}_q[x]/(x^n+1)$, where n is a power of 2 and $t \ll q$ are integers. It is customary to denote $R = \mathbb{Z}[x]/(x^n+1)$, so that the plaintext and ciphertext spaces become $R_t = R/tR$, and $R_q = R/qR$, respectively. We assume the fully homomorphic encryption scheme to have plaintext and ciphertext spaces of this type, and the notation (n, q, t) will always refer to these parameters. For example, the Brakerski-Gentry-Vaikuntanathan (BGV) [BGV12] and the Fan-Vercauteren (FV) [FV12] schemes have this structure.

A leveled fully homomorphic encryption scheme can be described by the following set of randomized algorithms:

- $\mathsf{FHE.Setup}(1^\lambda)$: Given a security parameter λ, outputs a set of encryption parameters `parms`.
- $\mathsf{FHE.KeyGen}(\texttt{parms})$: Outputs a secret key `sk` and a public key `pk`. Optionally outputs one or more evaluation keys `evk`.
- $\mathsf{FHE.Encrypt}(m, \texttt{pk})$: Given message $m \in R_t$, outputs ciphertext $c \in R_q$.
- $\mathsf{FHE.Decrypt}(c, \texttt{sk})$: Given ciphertext $c \in R_q$, outputs message $m \in R_t$.

- $\mathsf{FHE.Evaluate}(C, (c_1, \ldots, c_k), \mathtt{evk})$: Given an arithmetic circuit f with k input wires, and inputs $c_1, \ldots, c_k$ with $c_i \to \mathsf{FHE.Encrypt}(m_i, \mathtt{pk})$, outputs a ciphertext c such that

$$\Pr\left[\mathsf{FHE.Decrypt}(c, \mathtt{sk}) \neq f(m_1, \ldots, m_k)\right] = \mathrm{negl}(\lambda)\,.$$

  We also require that the size of the output of $\mathsf{FHE.Evaluate}$ is not more than polynomial in λ independent of what f is (*compactness*) (see e.g. [ABC+15]).

We say that a fully homomorphic encryption scheme is secure if it is IND-CPA secure, and *weakly circular secure*, which means that the scheme remains secure even when the adversary is given encryptions of the bits of the secret key. A fully homomorphic encryption scheme achieves *circuit privacy* if the distribution of the outputs of any fixed homomorphic evaluation is indistinguishable from the distribution of fresh encryptions of the plaintext outputs. In this way, one can effectively hide the circuit that was evaluated on encrypted data. We refer the reader to [ABC+15,BGV12,DS16] for more details.

3 The Basic Protocol

We describe our basic protocol in Figure 1 as a strawman protocol. The receiver encrypts each of its items y, and sends them to the sender. For each y, the sender then evaluates homomorphically the product of differences of y with all of the sender's items x, randomizes the product by multiplying it with a uniformly random non-zero plaintext, and sends the result back to the receiver. The result decrypts to zero precisely when y is in the sender's set, and to a uniformly random non-zero plaintext otherwise, revealing no information about the sender's set to the receiver.

To be more precise, we assume from now on that the plaintext modulus t in our FHE scheme is a prime number, large enough to encode σ-bit strings as elements of $\mathbb{Z}_t$. We also temporarily restrict the plaintext space to its subring of constant polynomials (this restriction will be removed in Section 4.1), and assume plaintexts to be simply elements of $\mathbb{Z}_t$. Recall that the sizes of the sets X and Y, and the (common) bit-length σ of the items, are public information.

Input: Receiver inputs set Y of size N_Y; sender inputs set X of size N_X. Both sets consist of bit strings of length σ. N_X, N_Y, and σ are public.
Output: Receiver outputs $X \cap Y$; sender outputs $\perp$.

1. **Setup**: Sender and receiver jointly agree on a fully homomorphic encryption scheme. Receiver generates a public-secret key pair for the scheme, and keeps the secret key to itself.
2. **Set encryption**: Receiver encrypts each element y_i in its set Y using the fully homomorphic encryption scheme, and sends the N_Y ciphertexts $(c_1, \ldots, c_{N_Y})$ to sender.
3. **Computing intersection**: For each c_i, sender
   (a) samples a random non-zero plaintext element r_i;
   (b) homomorphically computes
   $$d_i = r_i \prod_{x \in X} (c_i - x)\,.$$
   Sender return the ciphertexts $(d_1, \ldots, d_{N_Y})$ to receiver.
4. **Reply extraction**: Receiver decrypts the ciphertexts $(d_1, \ldots, d_{N_Y})$ and outputs
   $$X \cap Y = \{y_i : \mathsf{FHE.Decrypt}(d_i) = 0\}\,.$$

Fig. 1: Basic PSI protocol.

We have the following informal theorem with regards to the security and correctness of the basic protocol.

Theorem 1 (informal). *The protocol described in Figure 1 securely and correctly computes the private set intersection of X and Y in the semi-honest security model, provided that the fully homomorphic encryption scheme is IND-CPA secure and achieves circuit privacy.*

Proof (Proof sketch). Receiver's security is straightforward: the receiver sends an array of ciphertexts, which looks pseudorandom to the sender since the fully homomorphic encryption scheme is IND-CPA secure. For sender's security, we note that the receiver's view consists of an array of ciphertexts. It follows from circuit privacy that the receiver only learns the decryptions of these ciphertexts, and nothing more.

For a fixed index i, we have

$$\mathsf{FHE.Decrypt}(d_i) = r_i \prod_{x \in X} (y_i - x)\,,$$

which is zero precisely when $y_i \in X$ (correctness), and otherwise a uniformly random element in $\mathbb{Z}_t \setminus \{0\}$, because $\mathbb{Z}_t$ is a field. Thus, the receiver learns no additional information beyond the intersection $X \cap Y$.

This basic strawman protocol is extremely inefficient: it requires the sender to perform $O(N_{\text{X}} N_{\text{Y}})$ homomorphic multiplications and additions, and the depth of the circuit is high, pushing the FHE parameter sizes to be huge. In addition, the sender and the receiver need to communicate $O(N_{\text{Y}})$ FHE ciphertexts, which can be prohibitive even for state-of-the-art fully homomorphic encryption schemes. It is therefore quite surprising that the protocol becomes very efficient when combined with the enhancements described in the next section.

4 Optimizations

4.1 Batching

Our first step to improve performance is through the use of *batching*, which is a well-known and powerful technique in fully homomorphic encryption to enable SIMD (Single Instruction, Multiple Data) operations on ciphertexts. We give a brief explanation here, and refer the reader to [GHS12,BGH13,SV14,LCP16,GBDL+16] for more details and example applications.

For suitable choices of the plaintext modulus t, there is a ring isomorphism from the plaintext space R_t to $\mathbb{Z}_t^n$. As an example, a constant polynomial $a \in R_t$ corresponds to the vector $(a, \ldots, a) \in \mathbb{Z}_t^n$. Moreover, this isomorphism translates polynomial additions and multiplications into coefficient-wise additions and multiplications in each of the n fields $\mathbb{Z}_t$. To simplify the exposition, we use the polynomial and vector notations for plaintexts interchangeably, omitting the conversions from one representation to the other.

We can apply batching to reduce both the computational and communication cost of the basic protocol as follows. The receiver groups its items into vectors of length n, encrypts them, and sends N_{Y}/n ciphertexts to the sender. Upon seeing each ciphertext c_i, the sender samples a *vector* $r_i = (r_{i1}, \ldots, r_{in}) \in (\mathbb{Z}_t^*)^n$ of uniformly random non-zero elements of $\mathbb{Z}_t$, homomorphically computes $d_i = r_i \prod_{x \in X}(c_i - x)$, and sends it back to the receiver. Note that these modifications do not affect correctness or security, since the exact same proof can be applied per each vector coefficient.

The batching technique allows the sender to operate on n items from the receiver simultaneously, resulting in n-fold improvement in both the computation and communication. Since in typical cases n has size several thousands, this results in a significant improvement over the basic protocol.

4.2 Hashing

Even with the batching techniques of Section 4.1, the sender still needs to encode each of its set elements into separate plaintexts, and individually compare them to the receiver's items. Instead, it would be nice if the sender could also take advantage of batching. We will achieve this through the use of hashing techniques. Specifically, we use batching in conjunction with *cuckoo hashing* and *permutation-based hashing*, which have been developed and explored in detail in the context of PSI in e.g. [PSZ14,PSSZ15].

Before jumping into the technicalities of cuckoo hashing and permutation-based hashing, we start with a high-level explanation of why hashing is beneficial in our context. Suppose the two parties hash the items in their sets into two hash tables using some agreed-upon deterministic hash function. Now they only need to perform a PSI for each bin, since items in different bins are necessarily different.

One important point is that all bins must be padded to a fixed size to maintain security. Observe that the bins prior to padding will have uneven loads, and the load of a specific bin (the number of items mapped into the bin) can reveal additional information beyond the intersection. To overcome this, we need to pad each bin with dummy items up to a pre-determined maximum size.

The *simple hashing* technique just described significantly reduces the complexity of our protocol. It is well known that hashing d items into a hash table of size $m = d$ results in a maximum load of $O(\log d)$ with high probability. For example, in the case that both parties have $d = N_X = N_Y$ items, the overall complexity of the basic protocol reduces to $O(d \log^2 d)$, where the $\log^2 d$ factor comes from performing the basic PSI protocol on a single bin. Next, we will reduce the complexity even further via better hashing techniques.

Cuckoo hashing Cuckoo hashing [PR01,DM03,FPSS03] is a way to build dense hash tables by using $h > 1$ hash functions $H_1, ..., H_h$. To insert an item x, we choose a random index i from $[h]$, and insert the tuple (x, i) at location $H_i(x)$ in the table. If this location was already occupied by a tuple (y, j), we replace (y, j) with (x, i), choose a random j' from $[h] \setminus \{j\}$, and recursively re-insert (y, j') into the table. For $m \approx d$ and fairly small h, cuckoo hashing succeeds with very high probability, i.e. the recursive re-insertion process always succeeds before a pre-determined upper bound on the recursion depth is reached. We will discuss the success probability of cuckoo hashing in Section 4.2.

In order to apply cuckoo hashing to our PSI protocol, we must ensure that bin-wise comparisons will always yield the correct intersection. This is done by letting the receiver perform cuckoo hashing with $m \gtrsim N_Y$ bins. The sender must insert each of its items into a two-dimensional hash table using *all* h hash functions $H_1, ..., H_h$ (simple hashing), because there is no way for it to know which one of the hash functions the receiver eventually ended up using for the items in the intersection. To determine the maximum load on the sender's side, we apply a standard balls-into-bins argument. Concretely, when inserting $d = hN_X$ balls into m bins, we have

$$\begin{aligned} &\Pr[\text{at least one bin has load } > B] \\ &\leq m \sum_{i=B+1}^{d} \binom{d}{i} \left(\frac{1}{m}\right)^i \left(1 - \frac{1}{m}\right)^{d-i}. \end{aligned} \tag{1}$$

Our default assumption is that the sender (who performs simple hashing) has a larger set, so that $d > m \log m$. In this case B is upper-bounded by $d/m + O(\sqrt{d \log m/m})$ with high probability [RS98].

Permutation-based hashing Independent of the exact hashing scheme, permutation-based hashing [ANS10] is an optimization to reduce the length of the items stored in the hash tables by encoding a part of an item into the bin index. For simplicity, we assume m is a power of two, and describe permutation-based hashing only in connection with cuckoo hashing. To insert a bit string x into the hash table, we first parse it as $x_L \| x_R$, where the length of x_R is equal to $\log_2 m$. The hash functions $H_1, ..., H_h$ are used to construct location functions as

$$\mathtt{Loc}_i(x) = H_i(x_L) \oplus x_R\,, \quad 1 \leq i \leq h\,,$$

which we will use in cuckoo hashing. Moreover, instead of inserting the entire tuple (x, i) into the hash table as in regular cuckoo hashing, we only insert (x_L, i) at the location specified by $\mathtt{Loc}_i(x)$.

The correctness of the PSI protocol still holds after applying permutation-based hashing. The reason is if $(x_L, i) = (y_L, j)$ for two items x and y, then $i = j$ and $x_L = y_L$. If in addition these are found in the same location, then $H_i(x_L) \oplus x_R = H_j(y_L) \oplus x_R = H_j(y_L) \oplus y_R$, so $x_R = y_R$, and hence $x = y$. The lengths of the strings stored in the hash table are thus reduced by $\log_2 m - \lceil \log_2 h \rceil$ bits. The complete hashing routine is specified in Figure 2.

Hashing failures In an unlikely event where cuckoo hashing fails, it could leak some information of the receiver's set to the sender. To prevent this, we must ensure that with overwhelming probability the cuckoo hashing algorithm will succeed. While some asymptotic results exist for estimating the failure probability of cuckoo hashing [FMM09,DGM+10], the hidden constants are difficult to determine precisely. Instead, to obtain optimal parameters, we choose to determine the failure probability using empirical methods. The general technique we use is similar to that of [PSZ16], with two exceptions: first, we omit an auxiliary data structure known as the *stash* due to its incompatibility with the fully homomorphic encryption approach; second, we primarily focus on $h = 3$ in our experiments (see below), whereas [PSZ16] focused on $h = 2$.

Input: Receiver inputs set Y of size N_{Y}; sender inputs set X of size N_{X}. Both sets consist of bit strings of length σ. N_{X}, N_{Y}, σ are public. Both parties input integers h, m, B and a set of hash function $H_1, ..., H_h : \{0,1\}^{\sigma - \log_2 m} \to \{0,1\}^{\log_2 m}$. The location functions $\mathtt{Loc}_i$ is defined with respect to H_i for $i \in [h]$.
Output: Receiver outputs a permutation-based cuckoo hash table with the items in Y inserted, or $\perp$. Sender outputs a permutation-based hash table with the items in X inserted using simple hashing and all location functions, or $\perp$.

1. **[Sender]** Let $\mathfrak{B}_{\text{X}}$ be an array of m bins, each with capacity B, and value $\{(\perp, \perp)\}^B$. For each $x \in X$ and $i \in [h]$, the sender samples $j \leftarrow [B]$ s.t. $\mathfrak{B}_{\text{X}}[\mathtt{Loc}_i(x)][j] = \perp$, and sets $B_{\text{X}}[\mathtt{Loc}_i(x)][j] := (x_L, i)$. If the sampling fails due to a bin being full, the sender outputs $\perp$. Otherwise it outputs $\mathfrak{B}_{\text{X}}$.
2. **[Receiver]** Let $\mathfrak{B}_{\text{Y}}$ be an array of m bins, each with capacity 1, and value $(\perp, \perp)$. For each $y \in Y$, the receiver
   (a) sets $w = y$, and $i \leftarrow [B]$;
   (b) defines and calls the function $\mathsf{Insert}(w, i)$ as follows: swap (w, i) with the entry at $\mathfrak{B}_{\text{Y}}[\mathtt{Loc}_i(w)]$. If $(w, i) \neq (\perp, \perp)$, recursively call $\mathsf{Insert}(w, j)$, where $j \leftarrow [h] \setminus \{i\}$.
   If for any $y \in Y$ the recursive calls to Insert exceeds the system limit, the receiver halts and outputs $\perp$. Otherwise it outputs $\mathfrak{B}_{\text{Y}}$.

Fig. 2: Hashing routine.

Table size m	Insert size d											
	$3 \cdot 2^8$		$3 \cdot 2^{12}$		$3 \cdot 2^{16}$		$3 \cdot 2^{20}$		$3 \cdot 2^{24}$		$3 \cdot 2^{28}$	
	$\lambda = 30$	40	30	40	30	40	30	40	30	40	30	40
8192	8	9	17	20	68	74	536	556	6727	6798	100611	100890
16384	7	8	13	16	46	51	304	318	3492	3543	50807	51002

Table 1: Simple hashing bin size upper bound B for failure probability $2^{-\lambda}$, with $\lambda \in \{30, 40\}$, and $h = 3$; see equation (1).

We start by fixing the cuckoo hash table consisting of m bins, and vary the number for items $d < m$ to be inserted. For each (d, m) pair, we run the cuckoo hashing algorithm 2^{30} times. For $d \ll m$, we find that the algorithm never fails in the experiments. To compute the required ratio $\epsilon = m/d$ to achieve a statistical security level of $\lambda \geq 40$ (i.e. cuckoo hashing fails with probability at most 2^{-40}), we begin by setting ϵ to a value slightly larger than one, and gradually increase it until we can expect zero hashing failures. From this we observe that λ increases linearly with the scaling factor ϵ when $h \geq 3$.

Over the course of our experiments, we observed that cuckoo hashing with no stash performs very poorly when $h = 2$, which was also observed and discussed in detail in [PSZ16], which is why we shift our focus to $h = 3$. Furthermore, the marginal gain of $h = 4$ is outweighed by the increased cost of simple hashing. By applying linear regression to the empirical data for $\lambda \geq 0$, we observe that $\lambda = 124.4\epsilon - 144.6$ for $m = 16384$, and $\lambda = 125\epsilon - 145$ for $m = 8192$. To achieve a statistical security level of $\lambda = 40$, the maximum number of items that can be cuckoo hashed into 8192 bins with $h = 3$ is therefore 5535. For $m = 16384$, the corresponding maximum number of items is 11041. The respective simple hashing parameter for the given hash table size and different $d = hN_{\text{X}}$ values are given in Table 1.

Dummy values In order to make the sender's simple hash table evenly filled, we need to pad each bin with dummy items after hashing. We let the sender and receiver fix two different dummy values from $\mathbb{Z}_t$, as long as they do not occur as legitimate values. For example, if legitimate values have at most σ bits, then we can set the receiver's dummy value to 2^σ, and the sender's dummy value to $2^{\sigma+1} - 1$.

Hashing to a smaller representation In many cases the total number of items $N_{\text{X}} + N_{\text{Y}}$ is much smaller than the number 2^σ of all possible strings of length σ. Since the performance of our protocol will degrade with increasing string length, it is beneficial for the parties to compress their strings with an agreed-upon hash function to a fixed length $\sigma_{\max}$, and then execute the PSI protocol on these hashed strings. Indeed, this is a well-known technique in the PSI community.

More precisely, when a total of $N_X + N_Y$ random strings are hashed to a domain of size $2^{\sigma_{\max}}$, the probability of a collision is approximately $(N_X + N_Y)^2/2^{\sigma_{\max}+1}$. For a statistical security parameter λ, we require that Pr[collision occurs] $\leq 2^{-\lambda}$. Therefore, the compressed strings should have length at least

$$\sigma_{\max} = 2\log_2(N_X + N_Y) + \lambda - 1.$$

Now we apply permutation-based cuckoo hashing to the compressed strings, further reducing the string length to

$$\sigma_{max} - \log_2 m + \lceil \log_2 h \rceil.$$

In addition, we need to reserve two more values in the plaintext space for the dummy values discussed in Section 4.2. Thus, by choosing the encryption parameter t so that

$$\log_2 t > \sigma_{max} - \log_2 m + \lceil \log_2 h \rceil + 1 \tag{2}$$

we can accommodate arbitrarily long strings in our PSI protocol.

Combining with batching It is straightforward to combine hashing techniques introduced in this section with the batching technique in Section 4.1. After the receiver hashes its items into a table of size m, it parses the table into m/n vectors of length n. The receiver then encrypts each vector using batching, and proceeds as usual. Similarly, the sender performs the same batching step for each of the B columns of its two-dimensional hash table, resulting in Bm/n plaintext vectors. The rest of the protocol remains unchanged, and we see that adding batching to the hashing techniques provides an n-fold reduction in both computation and communication.

4.3 Reducing the Circuit Depth

With the optimizations discussed in Section 4.1 and Section 4.2, our protocol already achieves very low communication cost: typically just a few homomorphically encrypted ciphertexts. Unfortunately, the depth of the arithmetic circuit that needs to be homomorphically evaluated is still $O(\log N_X)$, which can be prohibitively high for currently known fully homomorphic encryption schemes.

We use two tricks—*windowing* and *partitioning*—to critically reduce this depth. For simplicity of exposition, we will discuss how these two tricks work over the basic protocol, and briefly explain how to combine them with previous optimizations.

Windowing We use a standard windowing technique to lower the depth of the arithmetic circuit that the sender needs to evaluate on the receiver's homomorphically encrypted data, resulting in a valuable computation-communication trade-off.

Recall that in the basic protocol, for each item $y \in Y$, the receiver sends one ciphertext $c = \mathsf{FHE.Encrypt}(y)$ to the sender, who samples a random element r in $\mathbb{Z}_t \setminus \{0\}$, homomorphically evaluates $r\prod_{x\in X}(c - x)$, and sends the result back to the receiver. If the receiver sends encryptions of extra powers of y, the sender can use these powers to evaluate the same computation with a much lower depth circuit. More precisely, for a *window size* of ℓ bits, the receiver computes and sends $c^{(i,j)} = \mathsf{FHE.Encrypt}(y^{i\cdot 2^{\ell j}})$ to the sender for all $1 \leq i \leq 2^\ell - 1$, and all $0 \leq j \leq \lfloor \log_2(N_X)/\ell \rfloor$. For example, when $\ell = 1$, the receiver sends encryptions of $y, y^2, y^4, \ldots, y^{2^{\lfloor \log_2 N_X \rfloor}}$.

This technique results in a significant reduction in the circuit depth. To see this, we write

$$r\prod_{x\in X}(y - x) = ry^{N_X} + ra_{N_X-1}y^{N_X-1} + \ldots + ra_0\,. \tag{3}$$

If the sender only has an encryption of y, it needs to compute *at worst* the product ry^{N_X}, which requires a circuit of depth $\lceil \log_2(N_X + 1) \rceil$. Now if the encryptions $c^{(i,j)}$ are already given to the sender, then we can separate the sender's computation into two steps. First, the sender computes an encryption of y^i for all $0 \leq i \leq N_X$. The sender needs to compute *at worst* a product of $\lfloor \log_2(N_X)/\ell \rfloor + 1$ terms, requiring a circuit of depth $\lceil \log_2(\lfloor \log_2(N_X)/\ell \rfloor + 1) \rceil$. In an extreme case, if the receiver gives the sender encryptions of all powers

of y up to y^{N_x}, the depth in this step becomes zero. Then, the sender computes a dot product of encryptions of y^i $(0 \leq i \leq N_x)$ with the vector of coefficients $(r, ra_{N_x-1}, \ldots, ra_0)$ in plaintext from its own data. This second step has multiplicative depth one.

The cost of windowing is in increased communication. The communication from the receiver to the sender is increased by a factor of $(2^\ell - 1)(\lfloor \log_2(N_x)/\ell \rfloor + 1)$, and the communication back from the sender to the receiver does not change.

It is easy to incorporate batching and hashing methods with windowing. The only difference is that batching and hashing effectively reduce the sender's set size by nearly a factor of n. More precisely, the depth of the circuit becomes $\lceil \log_2(\lfloor \log_2(B)/\ell \rfloor + 1) \rceil + 1$, where B is as in Figure 2. Without windowing, batching and hashing encode the entire set Y into one hash table of size $m \gtrsim N_Y$, producing m/n ciphertexts to be communicated to the sender. With windowing this is expanded to $(2^\ell - 1)(\lfloor \log_2(B)/\ell \rfloor + 1) \cdot m/n$ ciphertexts.

Finally, we note that security of windowing technique is guaranteed by the IND-CPA security of the underlying fully homomorphic encryption scheme.

Partitioning Another way to reduce circuit depth is to let the sender partition its set into α subsets, and perform one PSI protocol execution per each subset. In the basic protocol, this reduces sender's circuit depth from $\lceil \log_2(N_x + 1) \rceil$ to $\lceil \log_2(N_x/\alpha + 1) \rceil$, at the cost of increasing the return communication from sender to receiver by a factor of α.

Partitioning can be naturally combined with windowing in a way that offers an additional benefit of reducing the number of homomorphic operations. Recall from Section 4.3 that the sender needs to compute encryptions of all powers $y, \ldots, y^{N_x}$ for each of the receiver's items y. With partitioning, the sender only needs to compute encryptions of $y, \ldots, y^{N_x/\alpha}$, which it can reuse for each of the α partitions. Thus, with both partitioning and windowing, the sender's computational cost in the first step described in Section 4.3 reduces by a factor of α, whereas the cost in the second step remains the same.

We may combine batching and hashing with partitioning in the following way. The sender performs its part of the hashing routine (Figure 2) as usual, but splits the *contents* of its bins (each of size B) into α parts of equal size, resulting in α tables each with bin size $\approx B/\alpha$. It then performs the PSI protocol with the improvements described in Section 4.1, 4.2, and 4.3 using each of the α hash tables. Now sender's circuit depth reduces to $\lceil \log_2(\lfloor \log_2(B/\alpha)/\ell \rfloor + 1) \rceil + 1$, where B is as in Figure 2. The communication from the sender to the receiver is α ciphertexts.

We would like to note that in order to preserve the sender's security, it is essential that after using simple hashing to insert its items into the hash table, the sender partitions the contents of the bins—including empty locations with value $(\perp, \perp)$—in a uniformly random way. Since in the hashing routine (Figure 2) the sender inserts its items in random locations within each bin, the correct partitioning can be achieved by evenly splitting the contents of each bin into α subsets using any deterministic partitioning method.

4.4 Reducing Reply Size via Modulus Switching

Finally, we employ *modulus switching* (see [BGV12]), which effectively reduces the size of the response ciphertexts. Modulus switching is a well-known operation in lattice-based fully homomorphic encryption schemes. It is a public operation, which transforms a ciphertext with encryption parameter q into a ciphertext encrypting the same plaintext, but with a smaller parameter $q' < q$. As long as q' is not too small, correctness of the encryption scheme is preserved. Since FHE ciphertexts have size linear in $\log q$, modulus switching reduces ciphertext sizes by a factor of $\log q / \log q'$. This trick allows the sender to "compress" the return ciphertexts before sending them to the receiver. In practice, we are able to reduce the return ciphertexts to about 15–20% of their original size. We note that the security of the protocol is trivially preserved as long as the smaller modulus q' is determined at setup.

5 Full Protocol and Security Proof

5.1 Formal Description

We detail the full protocol in Figure 4, given a secure fully homomorphic encryption scheme with circuit privacy. The ideal functionality of this protocol is given in Figure 3.

Parameters: Two parties denoted as the sender and receiver with sets of items of bit-length σ. Receiver's set is of size N_Y; sender's set is of size N_X. sid denotes the session ID of the protocol instance.

Functionality: On input $(\text{RECEIVE}, \mathsf{sid}, Y)$ from the receiver and $(\text{SEND}, \mathsf{sid}, X)$ from the sender, where $X, Y \subseteq \{0,1\}^\sigma$, $|X| = N_X$, $|Y| = N_Y$. The functionality sends $(\text{OUTPUT}, \mathsf{sid}, X \cap Y)$ to the receiver, and nothing to the sender.

Fig. 3: Ideal functionality $\mathcal{F}_{\text{PSI}}$ for private set intersection with one-sided output.

We prove security in the standard *semi-honest* simulation-based paradigm. Loosely put, we say that the protocol Π_{PSI} of Figure 4 securely realizes the functionality $\mathcal{F}_{\text{PSI}}$, if it is correct, and there exist two simulators (PPT algorithms) $\mathsf{Sim}_R, \mathsf{Sim}_S$ with the following properties. The simulator Sim_R takes the receiver's set and the intersection as input, and needs to generate a transcript for the protocol execution that is indistinguishable from the receiver's view of the real interaction. Sim_S is similarly defined, with the exception of not taking the intersection as input. For a formal definition of simulation based security in the semi-honest setting, we refer the reader to [Lin16].

Theorem 2. *The protocol in Figure 4 is a secure protocol for $\mathcal{F}_{\text{PSI}}$ in the semi-honest setting.*

Proof. It is easy to see that the protocol correctly computes the intersection conditioned on the hashing routine succeeding, which happens with overwhelming probability $1 - 2^{-\lambda}$.

We start with a corrupt receiver, and show the existence of Sim_R. For easy of exposition, we will assume that the simulator/protocol is parameterized by $(h, m, B, n, q, t, \alpha, \ell, H', \{H_i\}_{1 \le i \le h})$, which are fixed and public, and that hashing to a smaller representation (Section 4.2) is used. We will then define the receiver's simulator Sim_R as follows. Sim_R computes the set $Y' = H'(Y)$, and uses a modified hashing routine to cuckoo-hash its elements into a table of size m. The modification is that if an element y is in $X \cap Y$, then a 0 is inserted, and otherwise a random non-zero element in $\mathbb{Z}_t$ is inserted. After hashing finishes, Sim_R inserts random non-zero elements from $\mathbb{Z}_t$ into the remaining empty slots. Next, Sim_R creates $\alpha - 1$ more tables of the same size, and fills them with random non-zero elements from $\mathbb{Z}_t$. It then randomly permutes the values inserted in the matching bins among all α tables. Finally, it batches each table into m/n FHE plaintext polynomials, and homomorphically encrypts them into m/n ciphertexts. The resulting $m/n \cdot \alpha$ ciphertexts will serve as a simulation of the receiver's view. Due to the circuit privacy assumption on underlying fully homomorphic encryption scheme, this view is indistinguishable from the receiver's view in the real execution of the protocol.

The case of a corrupt sender is straightforward. The simulator Sim_S can generate new encryptions of zero in place of the encryptions in Step 5. By the IND-CPA security of the fully homomorphic encryption scheme, this result is indistinguishable from the sender's view in the real protocol.

5.2 Discussion

Function privacy While our protocol (Figure 4) assumes a fully homomorphic encryption scheme with circuit privacy, in practice it is much more efficient to instantiate it with leveled fully homomorphic encryption (recall Section 2.3), i.e. choose encryption parameters large enough to avoid the costly bootstrapping operation. This does not change the security properties of the protocol, as the encryption parameters are selected purely based on public parameters N_X, N_Y and σ.

While circuit privacy can be achieved in fully homomorphic encryption using e.g. the techniques of [DS16], in practice the slightly weaker notion of (statistical) *function privacy* [GHV10] suffices, and is easier to achieve in the leveled setting using *re-randomization* and *noise flooding*, where the sender re-randomizes the output ciphertexts by homomorphically adding to them an encryption of zero with a very large noise [Gen09,DS16]. A standard "smudging lemma" (see e.g. [AJLA+12]) implies that in order to achieve $2^{-\lambda}$ statistical distance between output ciphertexts of different executions, it suffices to add encryptions of zero with noise $\lambda + \log_2 n + \log_2 \alpha$ bits larger than an upper bound on the noise in the original outputs of the computation. We used the heuristic results in [CS16] to bound the amount of noise in the output ciphertexts before flooding.

Input: Receiver inputs set $Y \subset \{0,1\}^\sigma$ of size N_Y; sender inputs set $X \subset \{0,1\}^\sigma$ of size N_X. $N_\text{X}, N_\text{Y}, \sigma$ are public. κ and λ denote the computational and statistical security parameters, respectively.
Output: The receiver outputs $Y \cap X$; the sender outputs $\perp$.

1. **[Perform hashing]** Hashing parameters h, m, B are agreed upon such that simple hashing hN_X balls into m bins with max load B, and cuckoo hashing N_Y balls into m bins succeed with probability $\geq 1 - 2^{-\lambda}$.
   (a) **[Hashing to shorter strings]** Let $\sigma' = 2\log_2(N_\text{X} + N_\text{Y}) + \lambda - 1$. If $\sigma > \sigma'$, then both parties hash their sets to a smaller representation. First, a random hash function $H' : \{0,1\}^\sigma \to \{0,1\}^{\sigma'}$ is sampled. Let $X' = \{H'(x) \mid x \in X\}$ and $Y' = \{H'(y) \mid y \in Y\}$. Perform the rest of the protocol with (X', Y', σ') replacing (X, Y, σ), and output the corresponding items in X, Y as the intersection.
   (b) **[Hashing to bins]** The parties perform Figure 2 with parameters h, m, B, and randomly sampled hash functions $H_1, ..., H_h : \{0,1\}^{\sigma - \log_2 m} \to \{0,1\}^{\log_2 m}$ as input. The sender performs Step 1 of Figure 2 with set X to obtain $\mathfrak{B}_\text{X}$, and the receiver performs Step 2 with Y to obtain $\mathfrak{B}_\text{Y}$.
2. **[Choose FHE parameters]** The parties agree on parameters (n, q, t) for an IND-CPA secure FHE scheme with circuit privacy. They choose t to be large enough so that $\log_2 t > \sigma - \log_2 m + \lceil \log_2 h \rceil + 1$.
3. **[Choose circuit depth parameters]** The parties agree on the windowing parameter $\ell \in [1, \log_2 B]$ and partitioning parameter $\alpha \in [1, B]$ as to minimize the overall cost.
4. **[Pre-process X]**
   (a) **[Partitioning]** The sender partitions its table $\mathfrak{B}_\text{Y}$ vertically (i.e. by columns) into α subtables $\mathfrak{B}_{\text{Y},1}, \mathfrak{B}_{\text{Y},2}, \ldots, \mathfrak{B}_{\text{Y},\alpha}$, each having $B' := B/\alpha$ columns.
   (b) **[Computing coefficients]** For each row v of each subtable, the sender replaces the row v with coefficients of the polynomial $\prod_s (x - v_s)$, i.e. it replaces v by $\text{Sym}(v) = ((-1)^j \sum_{S \subset [B'], |S| = j} \prod_{s \in S} v_s)_{0 \leq j \leq B'}$.
   (c) **[Batching]** For each subtable obtained from the previous step, the sender interprets each of its column as a vector of length m with elements in $\mathbb{Z}_t$. Then the sender batches each vector into m/n plaintext polynomials. As a result, the r-th subtable is transformed into $m/n \cdot B'$ polynomials $S_{i,j}^{(r)}$, $1 \leq i \leq m/n$, $0 \leq j \leq B'$.
5. **[Encrypt Y]**
   (a) **[Batching]** The receiver interprets $\mathfrak{B}_\text{Y}$ as a vector of length m with elements in $\mathbb{Z}_t$. It batches this vector into m/n plaintext polynomials $\overline{Y}_1, ..., \overline{Y}_{m/n}$.
   (b) **[Windowing]** For each batched plaintext polynomial $\overline{Y}$ computed during Step 5a, the receiver computes the component-wise $i \cdot 2^j$-th powers $\overline{Y}^{i \cdot 2^j}$, for $1 \leq i \leq 2^\ell - 1$ and $0 \leq j \leq \lfloor \log_2(B')/\ell \rfloor$.
   (c) **[Encrypt]** The receiver uses $\mathsf{FHE.Encrypt}$ to encrypt each such power, obtaining m/n collections of ciphertexts $\{c_{i,j}\}$. The receiver sends these ciphertexts to the sender.
6. **[Intersect]**
   (a) **[Homomorphically compute encryptions of all powers]** For each collection of ciphertexts $\{c_{i,j}\}$, the sender homomorphically computes a vector $\mathbf{c} = (c_0, \ldots, c_{B'})$, such that c_k is a homomorphic ciphertext encrypting $\overline{Y}^k$. In the end, the sender obtains m/n vectors $\mathbf{c}_1, \ldots, \mathbf{c}_{m/n}$.
   (b) **[Homomorphically evaluate the dot product]** The sender homomorphically evaluates

$$\mathbf{r}_{i,r} = \sum_{j=0}^{B'} \mathbf{c}_i[B' - j] \cdot S_{i,j}^{(r)}, \quad \text{for all } 1 \leq i \leq m/n, \text{ and } 1 \leq r \leq \alpha,$$

   optionally performs modulus switching on the ciphertexts $\mathbf{r}_{i,r}$ to reduce their sizes, and sends them back to the receiver.
7. **[Decrypt and get result]** For each $1 \leq r \leq \alpha$, the receiver decrypts all ciphertexts it receives and concatenates the resulting m/n vectors into one vector $\mathfrak{R}_r$ of length m. Finally, the receiver outputs

$$Y \cap X = \bigcup_{1 \leq r \leq \alpha} \{y \in Y : \mathfrak{R}_r[\mathtt{Loc}(y)] = 0\}.$$

Fig. 4: Full protocol.

Malicious behavior When considering malicious behavior our protocol faces several challenges. Most notable is the sender's ability to compute an arbitrary function on the receiver's homomorphically encrypted dataset. While the sender can not learn additional information directly from the ciphertexts, it is able to maliciously influence the correctness of the output, e.g. force the intersection/output to be the receiver's full

Name	n	q	t	DBC	κ
SEAL16384-1	16384	$2^{226} - 2^{26} + 1$	8519681	76	$\gg 128$ bits
SEAL16384-2	16384	$2^{226} - 2^{26} + 1$	8519681	46	$\gg 128$ bits
SEAL16384-3	16384	$2^{189} - 2^{21} + 9 \cdot 2^{15} + 1$	8519681	48	$\gg 128$ bits
SEAL8192-1	8192	$2^{226} - 2^{26} + 1$	8519681	46	≈ 120 bits
SEAL8192-2	8192	$2^{189} - 2^{21} + 9 \cdot 2^{15} + 1$	8519681	48	> 128 bits

Table 2: Encryption parameter sets for SEAL v2.1. Security estimates are based on [APS15,Alb17].

set, or more generally $f(X) \subseteq X$. Efficiently preventing such behavior by the sender appears to be extremely challenging.

For the case of a malicious receiver we need only to consider potential leakage which the receiver can induce (sender has no output). First, the receiver may provide a set of size greater than N_X due to its ability to fill vacant slots in the cuckoo hash table. Additionally, the argument that function privacy can easily be achieved through noise flooding no longer holds due to the receiver being possibly providing ciphertexts with more noise than expected. As such, the noise level of the response ciphertexts may depend on the sender's set, and thereby leak additional information. However, in general we believe that this protocol provides reasonable protection against a malicious receiver for most practical applications. We leave a more formal analysis of the malicious setting and potential countermeasures to future work.

When receiver holds the larger set So far we have made the assumption that the receiver's set size is much smaller than the sender's set size. Here we remark that our protocol can be slightly modified to handle the opposite case, where the receiver holds the larger set. The idea is that the two parties can perform our protocol with their roles switched until the last step. At this point, the receiver (who has now been playing the sender's role) holds an encryption of a vector v. It samples a random plaintext vector r, and sends back to the sender an encryption of $v + r$. The sender decrypts this value, and sends back the plaintext vector $v + r$ to the receiver, who can compute the final result v. This protocol is still secure in the semi-honest setting, and the communication remains linear in the smaller set and logarithmic in the larger set.

6 Implementation and Performance

6.1 Performance Results

We implemented our PSI protocol described in Figure 4. For fully homomorphic encryption we used SEAL v2.1 [LCP16], which implements the Fan-Vercauteren scheme [FV12] in C++. The parameters for SEAL that we used are given in Table 2, along with their computational security levels κ, estimated based on the best currently known attacks [APS15,Alb17]. The column labeled "DBC" refers to the `decomposition_bit_count` parameter in SEAL. We note that these parameters are highly optimized for the particular computations that we perform.

We give detailed computational performance results for our protocol in Table 3 for both single and multi-threaded execution with 4, 16, and 64 threads. As the receiver's computation is typically relatively small compared to the sender's, we restrict to single-threaded execution on the receiver's side. Still, it is worth pointing out that also the receiver's computation would benefit hugely from multi-threading, when available. Communication costs for our experiments are given in Table 4. We chose a statistical security level $\lambda = 40$, and a string length $\sigma = 32$ bits.

The benchmark machine has two 18-core Intel Xeon CPU E5-2699 v3 @ 2.3GHz and 256GB of RAM. We perform all tests using this single machine, and simulate network latency and bandwidth using the Linux `tc` command. Specifically, we consider a LAN setting, where the two parties are connected via local host with 10Gbps throughput, and a 0.2ms round-trip time (RTT). We also consider three WAN settings with 100Mbps, 10Mbps, and 1Mbps bandwidth, each with an 80ms RTT. All times are reported as the average of 10 trials.

Parameters			Optim.		Running time (seconds)									
N_X	N_Y	FHE parameters	α	ℓ	Sender pre-processing				Sender online				Receiver	
					$T=1$	4	16	64	1	4	16	64	Enc.	Dec.
2^{24}	11041	SEAL16384-2	256	1	72.2	18.0	6.2	3.0	42.2	14.4	7.1	5.6	0.3	10.3
			128	2	70.9	19.1	6.3	3.1	38.9	15.6	9.8	9.1	0.5	5.1
		SEAL16384-1	64	3	76.8	20.6	6.7	3.3	41.1	21.6	16.2	16.9	0.9	2.6
	5535	SEAL8192-1	256	1	64.1	17.9	5.5	2.7	36.0	11.8	6.3	5.5	0.2	4.9
			128	2	71.2	18.5	6.3	2.9	36.1	14.2	9.6	9.2	0.3	2.4
			64	3	80.4	21.5	6.7	3.2	41.9	21.5	17.7	17.7	0.5	1.2
2^{20}	11041	SEAL16384-1	128	1	9.1	2.5	1.0	0.5	8.0	2.6	1.2	1.1	0.2	5.1
			64	2	6.9	2.0	0.8	0.4	5.2	1.8	1.1	1.0	0.3	2.7
			32	3	6.4	1.7	0.9	0.6	4.5	2.1	1.3	1.5	0.7	1.3
	5535	SEAL8192-2	128	1	5.1	1.4	0.6	0.4	4.2	1.5	0.8	0.7	0.1	1.9
			64	2	4.4	1.2	0.6	0.3	3.4	1.7	0.7	1.0	0.2	1.0
			32	3	4.3	1.2	0.5	—	3.6	1.4	1.5	—	0.3	0.5
2^{16}	11041	SEAL16384-3	16	1	1.2	0.3	0.2	—	1.3	0.6	0.6	—	0.2	0.5
			8	2	1.0	0.3	0.2	—	1.5	1.2	1.3	—	0.3	0.3
			4	3	0.9	0.3	—	—	1.9	1.7	—	—	0.5	0.1
	5535	SEAL8192-2	32	1	0.9	0.3	0.2	—	0.9	0.4	0.3	—	0.1	0.5
			16	2	0.7	0.2	0.1	—	0.7	0.3	0.3	—	0.1	0.2
			8	3	0.6	0.2	—	—	0.7	0.5	—	—	0.3	0.1

Table 3: Running time in seconds for our protocol with $T \in \{1, 4, 16, 64\}$ threads; $\lambda = 40$, $\sigma = 32$, $h = 3$. Since we implemented multi-threading by dividing the α partitions evenly between threads, having $T > \alpha$ offers no performance benefit. These cases are denoted by "—" in the table.

Pre-processing The "Sender pre-processing" column in Table 3 measures the computational cost for the sender to prepare its coefficients of the polynomial $r \prod_{x \in X}(y - x)$, as mentioned in Section 4.3. More precisely, the sender's pre-processing work includes hashing and batching of its data, computing the coefficients in the right-hand side of (3), and sampling the random vectors. We also have the sender perform number theoretic transforms (NTT) to its plaintext polynomials to facilitate the underlying homomorphic multiplications in the second step described in Section 4.3.

We remark that our pre-processing can be done entirely offline without involving the receiver. Specifically, given an upper bound on the receiver's set size, the sender can locally choose parameters and perform the pre-processing. Upon learning the receiver's actual set size, the parameters selected by the sender are communicated to the receiver. We note that in order to achieve simulation-based security, the selected hash functions can only be used once. As such, each instance of the protocol must have an independent pre-processing phase, and in the event that a single pre-processing phase is used between several instances, an adversary with control of a party's set could force a hashing failure to occur. However, if such adversaries are not considered, then the pre-processing phase can be reused, resulting in significantly better performance.

PSI with longer items When implementing our PSI protocol, we restrict the item length to be 32 bits. The reason is, although we can accommodate arbitrary size items in principle as described in Section 4.2, doing so naively with our protocol would require the encryption parameters to be substantially increased, which has a large negative impact on performance. We leave the task of making our protocol efficient for arbitrary size items to future work.

6.2 Comparison to Pinkas *et al.* [PSZ16]

Our primary point of comparison is the Pinkas *et al.* PSI protocol [PSZ16], in which the authors consider both the case of symmetric set sizes, and the setting where the receiver's set is significantly smaller than the sender's. While our protocol can easily handle symmetric set sizes, our main advantage over [PSZ16] is in the asymmetric setting, which we now focus on. To make comparing the two protocols easier, we ran them on the same machine, and summarized the total running times side by side in Table 5. We chose to

Parameters			Optim.		Comm. size (MB)		Comm. time (seconds)			
N_X	N_Y	FHE parameters	α	ℓ	R $\rightarrow$ S	S $\rightarrow$ R	10 Gbps	100 Mbps	10 Mbps	1 Mbps
2^{24}	11041	SEAL16384-2	256	1	3.6	33.8	0.0	4.0	30.2	300.4
			128	2	6.3	16.9	0.0	2.4	19.0	186.7
		SEAL16384-1	64	3	12.7	8.4	0.0	2.2	17.4	169.4
	5535	SEAL8192-1	256	1	3.2	16.9	0.0	2.0	16.3	161.5
			128	2	4.1	8.4	0.0	1.3	10.3	101.0
			64	3	6.8	4.2	0.0	1.1	9.1	88.1
2^{20}	11041	SEAL16384-1	128	1	1.8	16.9	0.0	1.8	15.3	149.9
			64	2	3.6	8.4	0.0	1.3	9.9	98.0
			32	3	7.2	4.2	0.0	1.2	9.4	92.6
	5535	SEAL8192-2	128	1	1.1	8.4	0.0	1.0	7.8	77.0
			64	2	1.9	4.2	0.0	0.6	5.1	49.3
			32	3	3.4	2.2	0.0	0.6	4.7	45.0
2^{16}	11041	SEAL16384-3	16	1	2.3	2.1	0.0	0.5	3.6	35.5
			8	2	3.0	1.1	0.0	0.4	3.4	33.0
			4	3	6.0	0.5	0.0	0.6	5.4	52.9
	5535	SEAL8192-2	32	1	0.8	2.1	0.0	0.3	2.4	22.9
			16	2	1.5	1.1	0.0	0.3	2.2	20.7
			8	3	3.0	0.5	0.0	0.4	3.0	28.6

Table 4: Communication cost in MB for our protocol; $\lambda = 40$, $\sigma = 32$, $h = 3$. 10Gbps network assumes 0.2ms RTT, and the others use 80ms RTT. R $\rightarrow$ S and S $\rightarrow$ R denote the communications from receiver to sender, and from sender to receiver.

evaluate performance for the set sizes $N_Y \in \{5535, 11041\}, N_X \in \{2^{16}, 2^{20}, 2^{24}\}$ to maximize the utilization of ciphertext batching, described in Section 4.1. The sizes for N_Y were determined in Section 4.2 to be the largest that can guarantee a statistical security level of $\lambda \geq 40$. If a direct comparison to the running times reported in [PSZ16] is desired, the reader can feel free to round down our set sizes N_Y to match the sizes therein.

When comparing the two protocols, we find that our communication cost scales much better when the sender's set size is greater than 2^{16}. For instance, when considering strings of 32 bits, with $N_Y \leq 5535$ and $N_X = 2^{20}$, our protocol sends 5.6MB, while the same N_X, N_Y parameters applied to [PSZ16] result in 30.4MB of communication—a 5.4× improvement. Increasing N_X even further to $N_X = 2^{24}$, our protocol requires just 11.0MB of communication, whereas [PSZ16] requires over 480MB—a 43.7× improvement. Moreover, continuing to increase the sender's set size results in an even greater communication benefit.

When computing the intersection of sets of size $N_Y \leq 5535$ and $N_X = 2^{20}$ in a single-threaded LAN setting, our protocol requires 8.6 seconds. Evaluating the protocol of [PSZ16] using the same parameters results in an execution time of 3.1 seconds. While [PSZ16] is faster than our protocol in this particular setting, it also requires 5.4× more communication, and distributes the computational cost equally between the parties. That is, each party performs $O(N_X + N_Y)$ operations. In contrast, our protocol places very few requirements on the computational power of the receiver.

Since our protocol achieves a lower communication than [PSZ16] in the asymmetric set sizes setting, we obtain much better performance as we decrease the network bandwidth. To clearly demonstrate this, we consider several other network environments that model the WAN setting. In particular, we restrict the parties to a 100Mbps, 10Mbps, and 1Mbps networks with a 80ms round trip time. In these settings, our protocol outperforms [PSZ16] with few exceptions. Namely, the single-threaded 100Mbps setting, with $N_X = 2^{24}, N_Y \leq 5535$, our protocol requires 107.2 seconds, whereas [PSZ16] requires 87.9 seconds. However, our protocol receives a much greater speedup in the multi-threaded setting, reducing our running time to 36.7 seconds when the *sender* uses 4 threads. On the other hand, [PSZ16] requires 65.5 seconds for the same set sizes and with *both* parties using 4 threads—a nearly 1.8× slowdown compared to our protocol. As we further decrease the bandwidth, the difference becomes much more significant. In the 1Mbps single-threaded setting, with $N_X = 2^{24}, N_Y \leq 5535$, our protocol requires 211.1 seconds compared to [PSZ16] requiring 4080.6 seconds—a 19.3× improvement in running time. When utilizing 4 threads, our running time decreases to 132.7 seconds, while [PSZ16] requires 4064.3 seconds—a 30.6× improvement.

Parameters		Protocol	Comm.	Total time (seconds)							
N_X	N_Y		Size (MB)	10 Gbps		100 Mbps		10 Mbps		1 Mbps	
				$T=1$	4	1	4	1	4	1	4
2^{24}	11041	Us	23.2, †21.1	115.4	40.3	117.8	42.7	134.4	59.3	†290.8	†215.1
		[PSZ16]	480.9	40.5	23.3	88.0	66.4	449.5	427.5	4084.8	4067.2
		[KKRT16]	975.0	70.8	—	188.7	—	1269.1	—	12156.7	—
	5535	Us	20.1, †12.5, ‡11.0	105.2	34.8	107.2	36.7	†120.3	†45.8	†211.1	‡132.7
		[PSZ16]	480.4	40.1	23.1	87.9	65.5	449.2	427.3	4080.6	4064.3
		[KKRT16]	962.1	70.4	—	188.3	—	1263.5	—	12153.2	—
2^{20}	11041	Us	11.5	12.8	5.7	14.0	6.9	22.2	15.1	105.4	98.3
		[PSZ16]	30.9	3.3	2.1	7.0	5.6	29.8	28.3	263.7	262.1
		[KKRT16]	58.5	4.5	—	11.6	—	79.4	—	688.1	—
	5535	Us	5.6	8.6	3.3	9.2	3.9	13.3	8.0	53.6	48.3
		[PSZ16]	30.4	3.1	2.0	6.8	5.0	29.0	27.9	260.0	259.6
		[KKRT16]	57.3	4.4	—	11.5	—	79.3	—	686.0	—
2^{16}	11041	Us	4.1, †4.4	3.0	†1.7	3.4	†2.1	6.4	†5.3	36.0	35.0
		[PSZ16]	2.6	0.7	0.6	1.5	1.4	3.3	3.1	21.6	22.1
		[KKRT16]	4.5	0.4	—	1.4	—	5.6	—	48.2	—
	5535	Us	2.6	1.8	0.9	2.0	1.2	3.9	3.1	22.5	21.7
		[PSZ16]	2.1	0.7	0.6	1.4	1.3	2.9	2.8	19.8	21.3
		[KKRT16]	3.7	0.4	—	1.2	—	5.4	—	46.7	—

Table 5: Total communication cost in MB and running time in seconds comparing our protocol to [PSZ16] and to [KKRT16], with $T \in \{1, 4\}$ threads; $\lambda = 40$, $\sigma = 32$, $h = 3$. 10Gbps network assumes 0.2ms RTT, and others use 80ms RTT. Only single-threaded results are shown for [KKRT16] due to limitations of their implementation. The communication cost for [KKRT16] is based on the equation provided in their paper; empirical communication was observed to be ~ 1.5 times larger.

We also consider the running time of our protocol when more than 4 threads are used by the sender. When allowing 16 threads in the LAN setting, our running time decreases to 16.9 seconds for $N_X = 2^{24}, N_Y \leq 5535$. [PSZ16] on the other hand experiences less speedup over 4 threads, requiring just over 20 seconds for $N_X = 2^{24}$ when performed with 16 threads. This demonstrates that our protocol can outperform [PSZ16] even in the LAN setting, when at least 16 threads are used by the sender.

An important property of our protocol is the relatively small amount of work required by the receiver. In many applications the computations power of the receiver is significantly less than the sender. This is most notable in the contact discovery application where the receiver is likely a cellphone while the sender can be run at a large datacenter where computational power is inexpensive. For instance, Table 3 with parameters SEAL8192-1, $\alpha = 64, \ell = 3$ shows that for a intersection between 5535 and 2^{24} items, the receiver need only perform 1.7 seconds of computation while the server with 16 threads required 18 seconds with a total of 11MB of communication, less than half the size of the average 2012 iOS application download size [Res12] and a tenth of the average 2015 daily US smartphone mobile data usage [Eri16]. In contrast, [PSZ16] requires 480MB of communication—a 44× increase–and the computational load of the receiver is significantly higher requiring 50 million hash table queries and several thousand oblivious transfers.

6.3 Comparison to Kolesnikov *et al.* [KKRT16]

We also compare our protocol to that of Kolesnikov *et al.* [KKRT16], which optimizes the use of oblivious transfer. While their results do improve the running time for symmetric sets of large items, we found that when applied to our setting their improvement provides little benefit, and is outweighed by other optimizations employed by [PSZ16]. In particular, [PSZ16] considers a different oblivious transfer optimization which is more efficient on short strings, and also optimizes cuckoo hashing for the setting of asymmetric set sizes.

These design decisions result in [KKRT16] requiring 2× more communication than [PSZ16], and 87× more than our protocol, when intersecting 5535 and 2^{24} size sets with parameters SEAL8192-1, $\alpha = 64, \ell = 3$. When benchmarking [KKRT16], we found that the communication is actually ~ 1.5 larger than their theoretical

limit. The theoretical communication complexity of [KKRT16] is

$$N_{\text{X}} sv + k(1.2N_{\text{Y}} + s)\,,$$

where $s = 6$ is the stash size in cuckoo hashing, $k \approx 444$ is the width of the pseudorandom code, $v = \lambda + \log_2(N_{\text{X}}N_{\text{Y}})$ is the size of the OPRF output, and 1.2 is related to cuckoo hashing utilization. The communication complexity of [PSZ16] also follows same equation, but with a smaller k due to more optimized oblivious transfer sub-protocol. Our protocol on the other hand requires

$$1.5C\sigma N_{\text{Y}} \log_2 N_{\text{X}}$$

bits of communication, where C is a small constant for ciphertext expansion, $\sigma = 32$ is the string length, and 1.5 is related to the cuckoo hashing utilization with *no stash*. For example, when $N_{\text{X}} = 2^{24}$ and $N_{\text{Y}} = 5535$, our protocol requires only 12.5MB of communication, whereas the empirical communication of [KKRT16] in this setting is almost 115× larger.

This increase in communication translates into increased running times compared to [PSZ16] and our protocol in the WAN settings. For instance, when intersecting 5535 and 2^{24} items on a 10Mbps connection, our protocol is more than 57× faster, while [PSZ16] is only 3× faster. The total running times are summarized in Table 5 to make comparison to our protocol and to [PSZ16] easy. Since the implementation of [KKRT16] does not support multi-threading, we only present results for $T = 1$.

7 Conclusions

Although there has been huge progress in fully homomorphic encryption schemes since the groundbreaking work of Craig Gentry in 2009, it is still believed by many to be too expensive for practical use-cases. However, in this paper we have constructed a practical private set intersection protocol using the Fan-Vercauteren scheme, adopting and combining optimizations from both fully homomorphic encryption and cutting-edge work on PSI. We think our protocol is particularly interesting for the private contact discovery use-case, where it achieves a very low communication overhead: about 12MB to intersect a set of 5 thousand items with a set of 16 million items, which is significantly lower than in the previous state-of-the-art protocols. We regard our work as a first step to explore the possibilities of applying fully homomorphic encryption to private set intersection, and look forward to further discussions and optimizations.

References

ABC+15. Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A Reuter, and Martin Strand. A guide to fully homomorphic encryption. Technical report, IACR Cryptology ePrint Archive (2015/1192), 2015.

ACT11. Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (if) size matters: Size-hiding private set intersection. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, Italy, March 6-9, 2011. Proceedings*, volume 6571 of *Lecture Notes in Computer Science*, pages 156–173. Springer, 2011.

AJLA+12. Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 483–501. Springer, 2012.

Alb17. Martin R Albrecht. On dual lattice attacks against small-secret lwe and parameter choices in helib and seal. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 103–129. Springer, 2017.

ANS10. Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 787–796. IEEE, 2010.

APS15. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015.

BdM94. Josh Benaloh and Michael de Mare. *One-Way Accumulators: A Decentralized Alternative to Digital Signatures*, pages 274–285. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

BFT16a. Tatiana Bradley, Sky Faber, and Gene Tsudik. Bounded size-hiding private set intersection. In *International Conference on Security and Cryptography for Networks*, pages 449–467. Springer, 2016.

BFT16b. Tatiana Bradley, Sky Faber, and Gene Tsudik. Bounded size-hiding private set intersection. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings*, volume 9841 of *Lecture Notes in Computer Science*, pages 449–467. Springer, 2016.

BGH13. Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in LWE-based homomorphic encryption. In *Public-Key Cryptography–PKC 2013*, pages 1–13. Springer, 2013.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

BLLN13. Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding*, pages 45–64. Springer, 2013.

BV14. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM Journal on Computing*, 43(2):831–871, 2014.

CS16. Ana Costache and Nigel P Smart. Which ring based somewhat homomorphic encryption scheme is best? In *Cryptographers' Track at the RSA Conference*, pages 325–340. Springer, 2016.

DCW13. Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 789–800. ACM, 2013.

DGM+10. Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *International Colloquium on Automata, Languages, and Programming*, pages 213–225. Springer, 2010.

DM03. Luc Devroye and Pat Morin. Cuckoo hashing: further analysis. *Information Processing Letters*, 86(4):215–219, 2003.

DS16. Léo Ducas and Damien Stehlé. Sanitization of FHE ciphertexts. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–310. Springer, 2016.

DSMRY09. Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Efficient robust private set intersection. In *Proceedings of the 7th International Conference on Applied Cryptography and Network Security*, ACNS '09, pages 125–142, Berlin, Heidelberg, 2009. Springer-Verlag.

Eri16. Ericsson. Ericsson mobility report: On the pulse of the networked society. *Stockholm, Sweden*, 2016.

FMM09. Alan Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 490–503. Springer, 2009.

FNP04. Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004.

FPSS03. Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 271–282. Springer, 2003.

FV12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/.

GBDL+16. Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 201–210, 2016.

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

GHS12. Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

GHV10. Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-hop homomorphic encryption and rerandomizable yao circuits. In *Annual Cryptology Conference*, pages 155–172. Springer, 2010.

GSW13. Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

HEK12. Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

HFH99. Bernardo A. Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *In Proc. of the 1st ACM Conference on Electronic Commerce*, pages 78–86. ACM Press, 1999.

HL08. Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography Conference*, pages 155–175. Springer, 2008.

HN10. Carmit Hazay and Kobbi Nissim. Efficient set operations in the presence of malicious adversaries. In *International Workshop on Public Key Cryptography*, pages 312–331. Springer, 2010.

HN12. Carmit Hazay and Kobbi Nissim. Efficient set operations in the presence of malicious adversaries. *Journal of cryptology*, 25(3):383–433, 2012.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*, pages 145–161. Springer, 2003.

KKRT16. Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. Cryptology ePrint Archive, Report 2016/799, 2016. http://eprint.iacr.org/2016/799.

KLS+17. Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(4):97–117, 2017. To appear. Full version: http://ia.cr/2017/670. Code: http://encrypto.de/code/MobilePSI.

KMRS14. Seny Kamara, Payman Mohassel, Mariana Raykova, and Seyed Saeed Sadeghian. Scaling private set intersection to billion-element sets. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *Lecture Notes in Computer Science*, pages 195–215. Springer, 2014.

Lam16. Mikkel Lambaek. Breaking and fixing private set intersection protocols. Cryptology ePrint Archive, Report 2016/665, 2016. http://eprint.iacr.org/2016/665.

LATV12. Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.

LCP16. Kim Laine, Hao Chen, and Rachel Player. Simple encrypted arithmetic library - SEAL (v2.1). Technical report, Microsoft Research, September 2016.

Lin16. Yehuda Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. http://eprint.iacr.org/2016/046.

LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

Mar14. Moxie Marlinspike. The difficulty of private contact discovery. A company sponsored blog post, 2014. https://whispersystems.org/blog/contact-discovery/.

Mea86. C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134, April 1986.

NLV11. Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In Christian Cachin and Thomas Ristenpart, editors, *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 113–124. ACM, 2011.

OOS17. Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-n ot extension with application to private set intersection. In *Cryptographers' Track at the RSA Conference*, pages 381–396. Springer, 2017.

PR01. Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

PSSZ15. Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 515–530, 2015.

PSZ14. Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *Usenix Security*, volume 14, pages 797–812, 2014.

PSZ16. Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. Cryptology ePrint Archive, Report 2016/930, 2016. http://eprint.iacr.org/2016/930.

RAD78. Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

Res12. ABI Research. Average size of mobile games for ios increased by a whopping 42% between march and september. *London, United Kingdom*, 2012.

RR16. Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. Cryptology ePrint Archive, Report 2016/746, 2016. http://eprint.iacr.org/2016/746.

RS98. Martin Raab and Angelika Steger. "Balls into Bins" – a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

SV14. Nigel P Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.