

Network and System Defense Report

PSI Scheme implementation

Caliandro Piercero - 0299815

April 24, 2022

Contents

1	Introduction	2
2	PSI scheme overview	2
2.1	Setup phase	2
2.2	Element encryption	4
2.3	Intersection computation	5
3	Results and limitations	5

1 Introduction

The following document summarizes the implementation choices made to build a Private Set Intersection (PSI) scheme using homomorphic encryption. PSI is a computational problem in which two or more parties want to know the intersection between a certain set of elements that each one of them has, without revealing the content of the whole set each other.

To perform the homomorphic computation, the Microsoft SEAL library was used (link alla libreria), and the scheme built uses two parties, a **sender** and a **receiver**.

2 PSI scheme overview

The assumption made are the following:

- the receiver and the sender has their private sets S_r and S_s , of sizes respectively N_r and N_s ;
- each set is composed of bitstrings of length σ ;
- the values N_r , N_s and σ are public and known;

PSI scheme is composed of 3 main steps:

1. **setup**
2. **element encryption**
3. **intersection computation**

2.1 Setup phase

In this step, the sender and the receiver agrees on a fully homomorphic scheme.

Inside the code, there is not a "real" agreement using the network or something else, the agreement is made in the `src/main/main.cpp` file where the parameters for the scheme are decided. In particular, SEAL offers two main schemes:

- BFV, that can perform addition and multiplication on integers;

- CKKS, used mainly to perform multiplication on real numbers;

The choice in this case is to use BFV scheme, which can be "summarized" into 3 main parameters that one needs to set up for the `EncryptionParameters` class:

- degree of the polynomial modulus (spiegare cos'è): this parameter must be a power of 2, and represents the degree of a power-of-two cyclotomic polynomial. The larger this value is, the more complicated the encryption operations that can be performed, but also the size of the resulting ciphertext will be higher;
- ciphertext coefficient modulus (anche qui spiegare): this value is a product of distinct prime numbers, each up to 60 bits. The choice of this parameter implies larger noise.
To choose this parameter, it is possible to use a facility of the library that sets a suited value, by relying on

```
1 CoeffModulus::BFVDefault(poly_modulus_degree);
```

where the `poly_modulus_degree` is the value of [1];

- plaintext modulus, specific for the BFV scheme. In this case, it is set using

```
1 params.set_plain_modulus(PlainModulus::Batching(poly_mod_degree, 20));
```

After choosing these parameters, the receiver proceeds to generate a pair of public /private keys that will be used to encrypt and decrypt the data. Since the keys will be required for the receiver in following steps, the keys (as the receiver dataset) have been saved in a suited class:

```

1 class Receiver
2 {
3 public:
4     void setRecvDataset(vector<string> dataset){
5         this->recv_dataset = dataset;
6         if (dataset.size() > 0)
7             setBitsSize(dataset[0].length());
8     }
9     void setRecvSk(SecretKey sk){ this->recv_sk = sk; }
10    void setRecvPk(PublicKey pk){ this->recv_pk = pk; }
11    void setBitsSize(long size) { this->bits_size = size; }
12
13    SecretKey getRecvSk(){ return this->recv_sk; }
14    PublicKey getRecvPk(){ return this->recv_pk; }
15    vector<string> getRecvDataset(){ return this->recv_dataset; }
16    long getDatasetSize(){ return this->bits_size; }
17
18 private:
19     SecretKey recv_sk;
20     PublicKey recv_pk;
21     vector<string> recv_dataset;
22     long bits_size;
23 };

```

After the key generation, the receiver dataset is opened from the file (src/dataset/receiver.csv) and encrypted by the receiver.

To do so, the dataset is treated as a vector of `uint64_t`, using the **batch encoding** technique offered by the library.

So, encryption operation is performed on the whole matrix, and the resulting ciphertext is passed to the sender

2.2 Element encryption

In this phase, suppose that the sender receives the encrypted ciphertext from the receiver, which is a matrix where each entry is one of the encrypted values, c_i .

What the sender has to do is computing the intersection between its dataset and the receiver's one, as follows:

- generate a random, non-zero, value r_i ;
- homomorphically computes:

$$d_i = r_i \cdot \sum_{s' \in S'} (c_i - s') \quad (1)$$

where S' is sender's dataset

so in the end, the sender will produce another encrypted matrix d_i , the code to do so is the following:

```

1 BatchEncoder encoder(send_context);
2 size_t index;
3 size_t slot_count = encoder.slot_count();
4 size_t row_size = slot_count/2;
5
6 // Compute the first subtraction
7 vector<uint64_t> first_val_matrix(slot_count, longint_sender_dataset[0]);
8 Plaintext first_plain;
9 encoder.encode(first_val_matrix, first_plain);
10
11 send_evaluator.sub_plain(recv_ct, first_plain, d_i); // homomorphic computation of c_i - s_j
12
13 /* For each value of the sender dataset, compute the difference between the matrices.
14 * Then, multiply with the previous value to keep up with the polynomial computation */
15 for(long index = 1; index < sender_dataset.size(); index++){
16     vector<uint64_t> prod_matrix(slot_count, longint_sender_dataset[index]);
17     Plaintext sub_plain;
18     Ciphertext sub_encrypted;
19     Ciphertext prod_encrypted;
20     encoder.encode(prod_matrix, sub_plain);
21     send_evaluator.sub_plain(recv_ct, sub_plain, sub_encrypted);
22     send_evaluator.multiply(d_i, sub_encrypted, d_i);
23 }
24
25 // Finally, multiply for the random value
26 Plaintext rand_plain;
27 encoder.encode(gen_rand(slot_count, sender_dataset.size()), rand_plain);

```

2.3 Intersection computation

In the last step, the receiver gets the homomorphic computation of the sender, and can compute the intersection between the dataset:

$$I = N_r \cap N_s = \{n_r : \mathbf{Decrypt}(\mathbf{sk}, d_i) = 0\} \quad (2)$$

So, it can know which elements belong to the intersection without knowing the full set of the sender.

The multiplication for a random value will avoid any hint about the element not belonging to that intersection.

3 Results and limitations

Tests for the scheme are in the file `src/test/test.cpp`: in particular, the aim of the test is:

- verify that the scheme is working properly. To do so, the dataset for sender and receiver are created inside the file, and the intersection is computed and stored. So, the test will assert that the intersection computed by the receiver and the saved one are the same;
- furthermore, some data for the test as gathered, such as:

- the time needed to complete the full scheme;
- the resulting noise after the computation;
- the parameter of the scheme, in terms of the `poly_modulus_degree`

data obtained are summarized in the following table:

Modulus length	Bitstring size	Dataset size	Computation Time	Remaining noise
8192	24	4	2,56265	24
8192	24	6	4,62703	0
8192	24	8	7,2227	0
8192	24	10	10,2784	0

Table 1: Test result for polynomial modulus degree of 8192 bit

Modulus length	Bitstring size	Dataset size	Computation Time	Remaining noise
16384	24	4	10,7762	237
16384	24	6	18,7084	170
16384	24	8	29,1682	103
16384	24	10	41,6273	35

Table 2: Test result for polynomial modulus degree of 16384 bit

The clear limitation shown by the data result is in the invariant noise: even with a smaller dataset, for a modulus of size 8192, the noise goes soon to 0 meaning that it will be not possible for the receiver to decrypt the result correctly.

Increasing the modulus size allows to treat a larger dataset, but the performance are badly affected.

The library presents different examples that combines also other homomorphic schemes (such as the BGV one) but are much more complicated and requires a deeper knowledge of the library code, for example to choose a more suited value for the coefficient module manually.