Network and System Defense Report
PSI Scheme implementation

Caliandro Pierciro - 0299815

April 27, 2022

Contents

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 and the scheme built is made of two parties, a **sender** and a **receiver**.

2 PSI scheme overview

The assumption made are the following:

- the receiver and the sender have their private sets D_r and D_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, sender and receiver agree on a fully homomorphic scheme.
Inside the code, there is not a "real" agreement using the network or something else, indeed the agreement is made in the `src/main/main.cpp` file where the parameters for the scheme are decided. In particular, the scheme used in this implementation is BFV, which allows addition and multiplication over integers.
This scheme cannot perform arbitrary computations on encrypted data: in fact, each ciphertext has a specific quantity that is called **invariant noise budget**, measured in bits, which depends on the choice of the scheme's parameters and is consumed in homomorphic operations.
Once this noise budget reaches 0, the resulting ciphertext is corrupted and so it is not possible to obtain the correct result in decryption.
The 3 main parameters that one needs to set up in the `EncryptionParameters` class for the scheme are:

- **degree of the polynomial modulus**: 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**: this value is a product of distinct prime numbers, each up to 60 bits. The choice of this parameter implies larger noise budget, but there is an upper bound determined by the polynomial modulus degree.
  To choose this parameter, it is possible to use a facility of the library that sets a suited value, by relying on

  ```
  params.set_coeff_modulus(CoeffModulus::BFVDefault(poly_mod_degree));
  ```

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

- **plaintext modulus**, specific for the BFV scheme. This parameter determines the size of the plaintext data type and the consumption of noise budget, so it is essential trying to keep the data type for the plaintext smaller for better performance.
  The noise budget in a freshly encrypted ciphertext is:

  $$\approx log_2(\frac{coefficient\_modulus}{plaintext\_modulus}) \tag{1}$$

  and the noise consumption in a homomorphic multiplication is

  $$log_2(plaintext\_modulus) + \text{other terms} \tag{2}$$

  The choice was to use Batch Encoding, that allows to represent data in matrix form, resulting in better performance since the operations are made on each slot of the matrix.
  For batch mode, the plaintext modulus can be set by:

  ```
  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; these keys must remain secret.
Since the keys will be required for the receiver in following steps, they have been saved in a suited class:

```
class Receiver
{
public:
  void setRecvDataset(vector<string> dataset){
    this->recv_dataset = dataset;
    if (dataset.size() > 0)
      setBitsSize(dataset[0].length());
  }
  void setRecvSk(SecretKey sk){ this->recv_sk = sk; }
  void setRecvPk(PublicKey pk){ this->recv_pk = pk; }
    void setRelinKeys(RelinKeys relin_keys) { this->relin_keys = relin_keys; }
  void setBitsSize(long size) { this->bits_size = size; }

  SecretKey getRecvSk(){ return this->recv_sk; }
  PublicKey getRecvPk(){ return this->recv_pk; }
    RelinKeys getRelinKeys() { return this->relin_keys; }
  vector<string> getRecvDataset(){ return this->recv_dataset; }
  long getDatasetSize(){ return this->bits_size; }

private:
  SecretKey recv_sk;
  PublicKey recv_pk;
    RelinKeys relin_keys;
  vector<string> recv_dataset;
  long bits_size;
};
```

this class also keeps the receiver dataset, that will be accessed to check which strings belong to the intersection.
After generating the keys, 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`, which will be interpreted as a matrix. This matrix is encrypted 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:

$$d_i = r_i \cdot \sum_{n_s \in D_s} (c_i - n_s) \tag{3}$$

  where D_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:

```
  Evaluator send_evaluator(send_context);
  KeyGenerator send_keygen(send_context);


  /* Evaluation of the polynomial expressed at the top of this file. It is the PSI scheme polynomial,
     * that has to be computed for each element of the received ciphertext
   * */
  BatchEncoder encoder(send_context);
  size_t index;
  size_t slot_count = encoder.slot_count();
  size_t row_size = slot_count/2;

  // Compute the first subtraction
  vector<uint64_t> first_val_matrix(slot_count, longint_sender_dataset[0]);
  Plaintext first_plain;
  encoder.encode(first_val_matrix, first_plain);

  send_evaluator.sub_plain(recv_ct, first_plain, d);  // homomorphic computation of c_i - s_j

  /* For each value of the sender dataset, compute the difference between the matrices.
   * Then, multiply with the previous value to keep up with the polynomial computation
     * */
  for(long index = 1; index < sender_dataset.size(); index++){
    vector<uint64_t> prod_matrix(slot_count, longint_sender_dataset[index]);
    Plaintext sub_plain;
    Ciphertext sub_encrypted;
    encoder.encode(prod_matrix, sub_plain);

        // Subtract, multiply and relinearize the result to keep the size of the ciphertext = 2
    send_evaluator.sub_plain(recv_ct, sub_plain, sub_encrypted);
        send_evaluator.multiply_inplace(d, sub_encrypted);
#ifdef RELIN
      send_evaluator.relinearize_inplace(d, send_relin_keys);
#endif
    }

  // Finally, multiply for the random value
  Plaintext rand_plain;
  encoder.encode(gen_rand(slot_count, sender_dataset.size()), rand_plain);

  send_evaluator.multiply_plain_inplace(d, rand_plain);
#ifdef RELIN
  send_evaluator.relinearize_inplace(d, send_relin_keys);
#endif
```

An important aspect of the performance is in the use of **relinearization keys**: these keys are required to reduce the size of the resulting ciphertext, so that applying these after each multiplication the size can remain equals to 2.

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:

$$I = D_r \cap D_s = \{n_r : \mathbf{Decrypt}(\mathbf{sk}, d_i) = 0\} \tag{4}$$

So, it can know which element belongs 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. To simplify, the size of both dataset is the same.
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	2,01757	25
8192	24	6	3,71375	0
8192	24	8	6,04167	0
8192	24	10	8,95375	0

Table 1: Test result for polynomial modulus degree of 8192 bit, with no relinearization

Modulus length	Bitstring size	Dataset size	Computation Time	Remaining noise
16384	24	4	8,62892	237
16384	24	6	15,7851	170
16384	24	8	25,1823	103
16384	24	10	36,131	37

Table 2: Test result for polynomial modulus degree of 16384 bit, with no relinearization

The same test has been run using relinerization keys, showing an important advantage in terms of the time required to compute the whole scheme, as the size of the dataset grows:

Modulus length	Bitstring size	Dataset size	Computation Time	Remaining noise
8192	24	4	1,82994	24
8192	24	6	2,78646	0
8192	24	8	3,77176	0
8192	24	10	4,92907	0

Table 3: Test result for polynomial modulus degree of 8192 bit, with relinearization

The clear limitation shown by the data result is in the invariant noise: even with a small 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 whole implementation was derived on the basis of the examples offered by the library itself, which unfortunately does not have any further documentation, for example to choose a more suited value for the coefficient module or the plaintext modulus manually, so it would be necessary a deeper knowledge of the source code.

Modulus length	Bitstring size	Dataset size	Computation Time	Remaining noise
16384	24	4	8,56984	237
16384	24	6	13,1483	170
16384	24	8	17,7195	103
16384	24	10	22,0732	36

Table 4: Test result for polynomial modulus degree of 16384 bit, with relinearization