Network and System Defense Report
PSI Scheme implementation

Caliandro Pierciro - 0299815

May 2, 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, Microsoft SEAL library[**1**] 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**

All code is in the `src` directory, in particular the core files are in `src/lib`, which contains the file for the logic of receiver and sender, plus utility functions inside `utils.cpp`.

2.1 Setup phase and element encryption

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 there is a function inside the `utils.cpp` file where the parameters for the scheme are decided.
In particular, the homomorphic 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;

- **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:

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

  and the noise consumption in a homomorphic multiplication is

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

The choice of parameters is made as follows:

```
EncryptionParameters get_params(size_t poly_mode_degree)
{
    EncryptionParameters params(scheme_type::bfv);
  params.set_poly_modulus_degree(poly_mode_degree);
  params.set_coeff_modulus(CoeffModulus::BFVDefault(poly_mode_degree));
  params.set_plain_modulus(PlainModulus::Batching(poly_mode_degree, 20));

    return params;
}
```

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, which is `Receiver`, implemented in file `src/lib/utils.h` so that it can easily be imported where it is needed.
This class also keeps the receiver dataset, that will be accessed to check which strings belong to the intersection.
After generating the keys, in function `crypt_dataset`, the receiver dataset is opened from the file 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 first encoded (using `BatchEncoder`), encrypted and the resulting ciphertext is passed to the sender.

2.2 Intersection computation

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 compute, for each element i of the received encrypted set, what follows:

- generate a random, non-zero, value r_i which happens in `gen_rand` function;
- 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

all this computation is made by the function `homomorphic_computation`, so in the end, the sender will produce another encrypted matrix d_i.
An important aspect of performance in this computation is 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 Response extraction

In the last step, the receiver gets the homomorphic computation of the sender, and can extract the intersection between the dataset, considering each element i of the sender's encrypted computation:

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

where

- D_r and D_s are, respectively, receiver's and sender's private datasets;
- I will be the resulting intersection;

So, receiver 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

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

There are some limitation that have been found while developing the scheme and that have been confirmed by the results:

1. the first limitation shown by the data result is in the invariant noise: even with a small dataset, for a modulus of size 8192 bits, 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;

2. 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 finer grained value for the coefficient module or the plaintext modulus manually.
   To do so it would be necessary a deeper knowledge of the source code;

3. the choice of `uint64_t` as datatype is such that when the bitstring exceeds 24 bit of size, the multiplication between values can result in an overflow, so even if decryption can be performed, because the noise is greater than 0, the resulting intersection is not the right one (expected one).

References

[1] Microsoft SEAL library, see https://github.com/Microsoft/SEAL