Mathemathics with a Computer
Final Report

Dmitar Zvonimir Mitev

22 May 2024

1 Introduction

As part of the course "Mathematics with a Computer", I am implementing the functional encryption (FE) scheme for computing inner products by Abdalla et. al. that is based on the decisional Diffie-Hellman (DDH) assumption. f All the definitions concerning FE and the scheme itself can be found on [Abd+15]. I implemented all four algorithms of the scheme: Setup, Encrypt, KeyDer and Decrypt. The programming language of choice is C++ with the Crypto++ library which supports arbitrary-precision arithmetic. In this document I explain my way of implementing the scheme as well as the plans until the final defence. I suggest reading the intermediate report again before continuing onwards.

2 Helper functions and classes

For correct functioning of the scheme I implemented more functions and three classes. I will explain all of them one by one.

2.1 my_utilities.h and my_utilities.cpp

In these two files I implemented the functions I needed for the implementation of the scheme. As I tried to follow the good practices of programming in C++, the functions' names are in a header file which I include in all other files, while the actual implementation is in the .cpp file. By checking the .h file one can deduce the purpose of all the functions, as above each one there is a comment saying what the function is for. Two functions will most likely be removed before the final upload since I am not using them anywhere. Those are:

1. `isProbablePrime`: A function that utilizes the Miller-Rabin test to determine whether a given number is a probable prime. I figured out I can trust Crypto++ when it tells me that a generated number is a (safe) prime;
2. `generateCryptoSecureRandomInteger`: AutoSeededRandomPool from Crypto++ is already cryptographically secure according to their documentation. I use it instead.

Since most other functions have a straight-forward implementation, here I focus just on two others:

1. `generatorOfQuadraticResidues`: the idea here is that by generating any non-identity element, by squaring it one gets a generator of the quadratic residues group. This is a consequence of the fact that the quadratic residues (sub)group is of prime order. Since we are working with very big numbers, I bounded the random generated element with the maximum value of int in C++. It holds that in $\mathbb{Z}_p^*$ the hardness of the discrete logarithm problem is independent of the generator. Hence we do not need the generated element to be very big as the operations will be slower.

2. `babyStepGiantStep`: here I implemented Shanks' baby-step giant-step algorithm for the final discrete logarithm computation. For it I used a map data structure together with in-built functions of the Integer class. Since we want the discrete logarithm to be found in reasonable time, we are bounding the search from above with the `bound` parameter. Every important step is commented.

2.2 MasterSecretKey.h and MasterSecretKey.cpp

As the name suggests, this is the class for generating master secret keys i.e. keys that are later used for generating (ordinary) secret keys. The master secret key in [Abd+15] is a fixed-length array of integers – in my implementation the class has two attributes: a vector of Integers named `privateKey` and a length parameter ℓ. Since there is a mathematical correspondence between the elements of the pair (master secret key, master public key), I set the constructors of both MasterSecretKey and MasterPublicKey (explained in the follow-up) objects to private, with a third class being responsible for making sure that the mathematical relations between the keys hold, i.e. the keys correspond to one another.

2.3 MasterPublicKey.h and MasterPublicKey.cpp

This is the class for the objects representing the master public keys. The objects have 5 attributes: the key length ℓ, a safe prime p, a subprime $q = \frac{p-1}{2}$, generator of the quadratic residues subgroup g and a vector of Integers `publicKey`. The class has public getters for all attributes, but its constructors are set to private.

2.4 KeyPair.h and KeyPair.cpp

This is the class responsible for creating corresponding keys. It is a friend class [Wik] of both MasterPublicKey and MasterPrivateKey classes and is therefore able to access their private attributes. For a better illustration of the connection between the classes, see 1.

For all the elements of the `privateKey` and `publicKey` vectors it must hold $\texttt{publicKey}[i] = g^{\texttt{privateKey[i]}} \pmod p$ where g is the generator of the quadratic

residues group. We create these two objects simultaneously, thus preventing the user of creating a non-corresponding pair of keys. In other words, we force the master public and master private key to have the desired mathematical connection.

The object KeyPair has two different constructors. One of them takes two parameters: `length`, the length of the key vectors, and `bits`, the number of bits of the prime number. The second constructor takes a third parameter p in addition to the previous two. This is a safe prime of size `bits`. Since primes may be reused, if a safe prime is already generated we can use the second constructor. This does not compromise the safety of the scheme, as the prime is part of the public key and is in any case known to everyone.

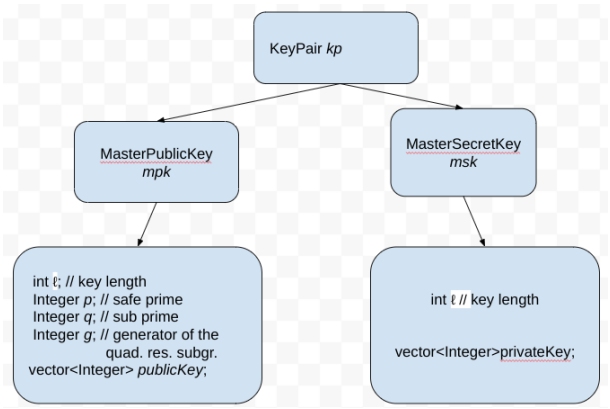

Figure 1: Connections between the classes.

3 Scheme.cpp

The implementation of the four algorithms of the scheme: Setup, Encrypt, Keygen and Decrypt, is in the file Scheme.cpp. With all of the points from the previous section the implementation is relatively straight-forward. Nonetheless, I will touch on a couple of points on each one of the four algorithms.

- Setup: takes three mandatory arguments: dimension of the key vector ℓ, number of bits of the safe prime `bits`, an upper bound of the elements of the vector `bound` and an optional fourth argument p, a safe prime. Then a KeyPair object is instantiated with the first or the second constructor depending on whether a fourth argument was provided. There are a lot of possible scenarios in which the inputted arguments are not compatible with the purpose of the scheme. In all such cases an exception is thrown. The most interesting is when $\ell B^2 \geq q$. In that case an overflow may happen, hence the solution of the final discrete logarithm is not necessarily unique. In [Abd+15] it is explicitly said that for the scheme to work it

must hold that $\ell B^2 < q$. I took care of this by potentially throwing an `invalid_argument` exception.

- Encrypt: takes three arguments MasterPublicKey `mpk`, vector⟨Integer⟩ `plaintext` and an upper bound `bound`. Then [Abd+15] is followed and the ciphertext is produced and returned.
- Keygen: takes three arguments: KeyPair `key`, vector⟨Integer⟩ y – the vector for which the key is to be generated, and an upper bound `bound`. For the generation of the secret keys one needs both the master secret key and the master public key. We give the function access to both keys via the KeyPair object.
- Decrypt: takes five arguments:
  1. the MasterPublicKey `mpk`;
  2. vector⟨Integer⟩ `ciphertext`;
  3. vector⟨Integer⟩ y – the vector for which one has the corresponding secret key;
  4. Integer `sk_y` – the secret key for vector y;
  5. B – the upper bound for the discrete logarithm computation.

  Following [Abd+15][page 8], first the inverse of the denominator is computed ($\mathtt{ct}_0^{-\mathtt{sk\_y}}$) followed by the product in the numerator ($\prod_{i=1}^{\ell} \mathtt{ct}_i^{y_i}$). After multiplying those two terms a discrete logarithm is computed with the baby-step giant-step algorithm, from which one recovers the inner product $\langle \mathtt{plaintext}, y \rangle$. The discrete logarithm is searched in the set $\{0, 1, \ldots, \ell B^2\}$. Since we put additional restrictions in the Encrypt function on the individual elements of the plaintext, the discrete logarithm must be in the above set.

4 Findings so far

As expected, two things are problematic. The first is the decryption function. The computation of the discrete logarithm takes a lot of resources and it already happened to me that my computer went out of RAM after setting the bound too high.

The second is the generation of the safe primes. The time needed for generating a large safe prime is not that short and varies a lot. For parameters $B = 10\,000$, `bits` $= 1024$ and $\ell = 100$, for a randomly generated vector my computer needs anywhere between 0.3 and 2 seconds to execute all four algorithms on my computer. By setting `bits` to 2048 the time varies between 3 and 20 seconds. This motivates the final topic.

5 Future plans

Since everything is working properly, I do not plan on doing any major changes. Before the defence I plan on performing tests to see the time needed for different parameters B, `bits` and ℓ. I plan on conducting this in two steps as follows:

1. measure the time needed for generating safe primes of different bit sizes;
2. use precomputed safe prime(s) to measure the execution time of the four algorithms, i.e. use the second constructor of KeyPair.

From that data, I hope to understand the maximum parameters the scheme supports and identify its "breaking point". I will post the measured times of execution on GitHub once I compute them.

References

[Abd+15] M. Abdalla et al. "Simple functional encryption scheme for inner products". In: *Public-Key Cryptography – PKC 2015*. Springer. 2015, pp. 733–751.

[Wik] Wikipedia. *Friend class — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Friend_class#cite_note-1. [Online; accessed 22-May-2024].