Introduction à la cryptographie : Implémentation algo de chiffrement d'El Gamal

Romain Duc, Alexandre Serratore

6 mars 2023

Question 1 :
Quel langage de programmation avez-vous choisi ? Quelle bibliothèque permettant de gérer des nombres entiers de grande taille allez-vous utiliser ? Quelles sont les opérations implémentées dans cette bibliothèque (multiplication, addition,...) ?

Réponse :
Nous avons choisi d'utiliser le langage Java. Java nous a paru l'option la plus adaptée, car il dispose de bibliothèques mathématiques intégrées permettant d'effectuer des opérations complexes, e.g. l'exponentiation modulaire.
La classe *java.math.BigInteger* permet de représenter et de manipuler des nombres entiers de grande taille sans être contraint par les limitations de taille des types primitifs ou leurs enveloppes. Il est possible d'utiliser cette classe pour effectuer des soustractions, des modulos, des multiplications. En plus des opérations habituelles sur les entiers, elle offre des opérations de calcul en arithmétique modulaire, calcul du pgcd, génération de nombre premier, test pour savoir si un entier est premier, etc...

Question 2 :
De plus, pour générer les valeurs (très grandes elles aussi), x et r, il faut être capable de générer des nombres aléatoires de très bonne qualité et dits cryptographiquement sûrs.
En vous aidant d'internet, donnez la définition d'un nombre aléatoire cryptographiquement sûr. Selon le langage de programmation choisi, donnez le nom de la bibliothèque qui va vous permettre de générer ces nombres aléatoires.

Réponse :
Un nombre aléatoire cryptographiquement sûr est un nombre:
- Généré aléatoirement qui est difficile à prédire ou à reproduire, même en utilisant les informations sur la façon dont il a été généré

- Aléatoire indépendant des autres nombres aléatoires générés précédemment.
- Très difficile, étant donné les k premiers bits d'une séquence, de trouver le $(k+1)$-ème bit à l'aide d'un algorithme polynomial avec un taux de succès de plus de 50%.

Ces nombres sont souvent utilisés pour sécuriser les communications cryptographiques en générant des clés de chiffrement ou en ajoutant du bruit à des communications pour empêcher qu'elles ne soient déchiffrées.
Nous utiliserons la bibliothèque "SecureRandom" de Java pour générer des nombres aléatoires cryptographiquement sûrs. Voici une exemple d'utilisation de cette bibliothèque:

```
import java.security.SecureRandom;

SecureRandom random = new SecureRandom().getInstanceStrong();
int randomInt = random.nextInt();
```

- https://miashs-www.u-ga.fr/prevert/Prog/Java/CoursJava/lesBig.html
- https://stackoverflow.com/questions/11051205/difference-between-java-util-rando

Question 3 :
Implémentez la fonction Euclide(). Testez là en vérifiant sur 10000 valeurs différentes de a que a · u + p · v = 1. Vous pouvez également vérifier que si une fonction existe déjà dans votre librairie, vous retournez bien les mêmes valeurs.
Réponse :
Nous avons implémenté la fonction Euclide() en utilisant l'algorithme d'Euclide étendu (solution itérative). Cette fonction prends donc en paramètre 2 *BigInteger* puis retourne un tableau de *BigInteger* contenant PGCD(a,b) ainsi que les coefficient u et v aux indices respectifs [0], [1] et [2].

```
public BigInteger[] euclideEtendu2(BigInteger a, BigInteger b)
{
    final BigInteger UN = BigInteger.ONE, ZERO = BigInteger.ZERO;
    BigInteger u = UN,
            v = ZERO,
            u1 = ZERO,
            v1 = UN,
            t;

    // tant que BigInteger.valueOf(b) > 0
    while(b.compareTo(ZERO) > 0){
        //q = a / b;
        BigInteger q = a.divide(b);
```

```
14          t = u;
15          u = u1;
16          //u = u0 − q ∗ u1
17          u1 = t.subtract(q.multiply(u1));
18          t = v;
19          v = v1;
20          //v = v0 − q ∗ v1
21          v1 = t.subtract(q.multiply(v1));
22          t = b;
23          b = a.subtract(q.multiply(b));
24          a = t;
25      }
26      return (a.intValue()>0)? new BigInteger[]{a,u,v} : new
            BigInteger[]{a.negate(),u.negate(),v.negate()};
27  }
```

Listing 1: Euclide étendu itératif

Nous avons testé cette implémentation à l'aide de la fonction *void test10000Times(BigInteger p, SecureRandom random)* (Voir ci-après).
Pour le même entier p nous générons 10000 valeurs de a à l'aide de SecureRandom et nous appellons la fonction Euclide() définie précédement. Puis nous vérifions à l'aide de 3 assert que:
- le tableau retourné contient bien le gcd(a,p) (A l'aide de la fonction *gcd(p)* de la classe *BigInteger*)
- le gcd(a,p) est bien égale à au + pv
- l'équation de bezout donne bien 1

Nous affichons les 5 derniers résultat dans le fichier "test.txt".

```
1   public void test10000Times(BigInteger p, SecureRandom random) throws
        EuclideException {
2       BigInteger a, gcd_ap, bezout;
3       BigInteger[] results;
4       StringBuilder result = new StringBuilder();
5
6       result.append("Test de la fonction Euclide() : \n");
7       for(int k=0; k < 10000; k++) {
8           a = new BigInteger(1024, random);
9           results = euclideCompute(a, p);
10          gcd_ap = a.gcd(p).abs();
11          bezout = a.multiply(results[1]).add(p.multiply(results[2]));
12          assert(results[0].equals(gcd_ap)):"le premier element du
                tableau ne contient pas le gcd(a,p)";
13          assert(gcd_ap.equals(bezout)):"gcd(a,p) != au + pv";
14          assert(BigInteger.valueOf(1).equals(bezout)):"equation de
                bezout ne donne pas 1";
```

```
15
16          //sortie que pour les 5 premieres occurrences
17          if(k < 5){
18
19              result.append("a = ").append(a).append("\t et \n");
20              result.append("a.u + p.v = ").append(bezout).append("\n")
                    ;
21              // verifie que pgcd(a,p) == results[0]
22              result.append("results[0] == pgcd(a,p) = ").append(
                    results[0].equals(gcd_ap)).append("\n");
23              // Verifie que a * u + b * v = GCD(a, p)
24              result.append("au + pv == pgcd(a,p) = ").append(gcd_ap.
                    equals(bezout)).append("\n");
25              result.append("gcd == 1 = ").append(BigInteger.valueOf(1)
                    .equals(bezout)).append("\n\n");
26
27          }
28      }
29
30  }
```

Listing 2: testEuclide

Question 4 :
Implémentez la fonction ExpMod(). Testez-la sur 10000 valeurs différentes.
Réponse :
Nous avons implémenté cette fonction en utilisant l'algorithme de l'exponentiation rapide avec *BigInteger expMod(BigInteger p, BigInteger g, BigInteger a)* qui se situe dans la classe *ExponentiationModulaire.java*. Cette fonction prends donc en paramètre 3 *BigInteger* puis retourne un *BigInteger* $A = (g^a \equiv p)$.

```
1   public BigInteger expMod(BigInteger p, BigInteger g, BigInteger a){
2       BigInteger x = BigInteger.ONE;
3
4       while(a.compareTo(BigInteger.ZERO) > 0) { //tant que a>0
5           if(a.testBit(0)) { //a pair
6               x = (x.multiply(g)).mod(p);
7           }
8
9           g = (g.multiply(g)).mod(p); //g = g**2 mod p
10          a = a.shiftRight(1);  //a/=2
11      }
12      return x;
13  }
```

Listing 3: Exponentiation rapide

Nous avons testé cette implémentation à l'aide de la fonction *void test10000Times(BigInteger p, BigInteger g, SecureRandom sr) (Voir ci-après.*

Pour le même entier p nous générons 10000 valeurs de a à l'aide de SecureRandom et nous appellons la fonction Euclide() définie précédement. Puis nous vérifions à l'aide de 3 assert que:
- le tableau retourné contient bien le gcd(a,p) (A l'aide de la fonction *gcd(p)* de la classe *BigInteger*)
- le gcd(a,p) est bien égale à au + pv
- l'équation de bezout donne bien 1
Nous affichons les 5 derniers résultat dans le fichier "test.txt".

```
1  public void test10000Times(BigInteger p, BigInteger g, SecureRandom
       sr) {
2      BigInteger a, ourExpMod, modPowofBigint;
3      StringBuilder sb = new StringBuilder();
4
5      sb.append("Test de la fonction expMod() : \n");
6      for(int k=0; k<10000; k++){
7          a = new BigInteger(500,sr);
8          ourExpMod = expMod(p,g,a);
9          modPowofBigint = g.modPow(a,p) ;
10         assert(ourExpMod.equals(modPowofBigint)):"((a^g) mod p !=
               expMod(p,g,a))";
11
12         //5 premieres  occurrences
13         if(k<5){
14             sb.append("a = ").append(a).append("\t et \n");
15             sb.append("expMod(p,g,a) = ").append(ourExpMod).append("\
                   n");
16             sb.append("((a^g) mod p == expMod(p,g,a)) = ").append(
                   ourExpMod.equals(modPowofBigint)).append("\n\n");
17         }
18
19     }
20     try{
21         bufferedWriter.write(sb.toString());
22     }catch(IOException e){
23         e.printStackTrace();
24     }
25     System.out.println(sb);
26     System.out.flush();
27
28 }
```

Listing 4: testExpMod

Question 5 :
Implémentez ces trois fonctions. Testez-la en vérifiant que 100 valeurs

différentes de m donne bien les déchiffrés attendus (c'est-à-dire de nouveau les m!). Vérifiez également que les 100 r tirés au hasard sont bien différents.
Réponse :
Nous avons implémenté les fonctions de la manière suivantes:

- *BigInteger[] keyGen(BigInteger p, BigInteger g, SecureRandom rand).* Cette fonction retourne donc un couple de valeur en générant un nombre x aléatoire puis en calculant X a l'aide de l'exponentiation modulaire et des deux entier (p et g) passés en paramètre.

```
1  /**
2  * Generation des cles
3  * @param p valeur du grand nombre premier
4  * @param g valeur du grand nombre premier
5  * @return BigInteger[2] contenant k_p et k_s
6  */
7  public BigInteger[] keyGen(BigInteger p, BigInteger g, SecureRandom
       rand) {
8      BigInteger x = new BigInteger(1024, rand); // tire x au hasard (
           private-key)
9      BigInteger X = exponentiationModulaire.expMod(p,g,x); // calcule X
           = g^x mod p (public-key)
10     return new BigInteger[] {x, X};
11 }
```

Listing 5: keyGen

- *BigInteger[] encrypt(BigInteger p, BigInteger g, BigInteger publicKey, BigInteger m, SecureRandom rand).* Cette fonction retourne un couple (C, B) qui est calculé de la manière suivante: $B = expMod(p, g, r)$ avec r un nombre aléatoire et $C = m \times expMod(p, publicKey, r) \equiv p$ avec m, p, $publicKey$, r et p passé en paramètre.

```
1  /**
2  * Fonction de chiffrement : prend en entree
3  * la cle publique de Bob K_p = (p,g,X) et un message m
4  * @param p
5  * @param g
6  * @param publicKey
7  * @param m
8  * @return couple chiffre {C,B} avec C = m.y mod p et B = g^r mod p
9  */
10 public BigInteger[] encrypt(BigInteger p, BigInteger g, BigInteger
       publicKey, BigInteger m, SecureRandom rand){
11     BigInteger r = new BigInteger(1024, rand);
12     BigInteger B = exponentiationModulaire.expMod(p, g, r); // calcule
           B = g^r mod p
```

```
13      BigInteger C = m.multiply(exponentiationModulaire.expMod(p,
            publicKey, r)).mod(p); // calcule C = m * X^r mod p
14      return new BigInteger[]{ C, B };
15  }
```

Listing 6: encrypt

- *BigInteger decrypt(BigInteger C, BigInteger B, BigInteger x, BigInteger p)* Cette fonction retourne le message m déchiffré a l'aide du couble $(C, B$ et de la clé secrète x en utilisant un inverse modulaire. L'implémentation de la fonction d'inverse modulaire est décrite plus bas.

```
1   /**
2   * Fonction de dechiffrement
3   * @param C 1e partie du chiffre
4   * @param B 2e partie du chiffre
5   * @param x cle secrete de Bob
6   * @param p grand entier : modulo
7   * @return
8   */
9   public BigInteger decrypt(BigInteger C, BigInteger B, BigInteger x,
        BigInteger p) throws EuclideException {
10      return (euclide.modInv(exponentiationModulaire.expMod(p,B,x),p))
            .multiply(C).mod(p);
11  }
```

Listing 7: decrypt

Nous avons testé leur implémentation (en générant 100 valeurs différentes) à l'aide de la fonction de test ci-dessous :

```
1   public void test100Times(BigInteger p, BigInteger g, SecureRandom
        random) throws EuclideException {
2       BigInteger message, message2;
3       BigInteger[] keys, encrypt;
4       StringBuilder sb1 = new StringBuilder();
5
6       sb1.append("Test du chiffrement ElGamal : \n\n");
7
8       for(int k=0; k<100; k++){
9           message = BigInteger.valueOf(Math.abs(random.nextInt()));
10          keys = keyGen(p,g, random);
11
12          encrypt = encrypt(p,g, keys[1], message, random);
13          message2 = decrypt(encrypt[0],encrypt[1], keys[0], p);
14          assert(message.intValue() == message2.intValue()):"message dé
                chiffré différent";
15
16          //5 premieres occurrences
17          if(k < 5){
```

```
18              sb1.append("Le message est : ").append(message.intValue()
                    ).append("\n");
19              sb1.append("Le message chiffré est : C : ").append(
                    encrypt[0].intValue()).append(" - et B : ").append(
                    encrypt[1].intValue()).append("\n");
20
21              sb1.append("Le message déchiffré est : ").append(message2
                    .intValue()).append("\n");
22              sb1.append("Message correctement déchiffré ? ").append(
                    message.intValue() == message2.intValue()).append("\n
                    \n");
23          }
24      }
25      try {
26          bufferedWriter.write(sb1.toString());
27      } catch (IOException e) {
28          e.printStackTrace();
29      }
30      System.out.print(sb1);
31      System.out.flush();
32  }
```

Listing 8: Test chiffrement El-Gamal

La fonction ci-après est notre implémentation du calcul de l'inverse modulaire d'un nombre. (utilisée dans la fonction Decrypt())

```
1   public BigInteger modInv(BigInteger a, BigInteger b) throws
        EuclideException {
2       BigInteger[] resEuclide = euclideEtendu2(a,b);
3       if(!resEuclide[0].equals(BigInteger.ONE)){
4           throw new EuclideException("No modular inverse possible");
5       }else{
6           return resEuclide[1].mod(b);
7       }
8   }
```

Listing 9: Inverse modulaire

Question 6 :
Vérifiez cette propriété en chiffrant deux messages différents m1 et m2 avec la fonction Encrypt(), puis en calculant C = C1 . C2 mod p et B = B1 . B2 mod p et enfin en déchiffrant via la fonction Decrypt() le couple (C, B) pour récupérer m. Vérifier ensuite que m = m1.m2 mod p.
Réponse :

Nous avons implémenté la fonction de test ci-dessous qui test 5 fois la propriété :

```java
public void homomorphic_property(BigInteger p, BigInteger g,
    SecureRandom random) throws EuclideException{
    BigInteger message, message2;
    BigInteger[] keys, encrypt, encrypt2;
    BigInteger decryptTotal, productm1m2;
    StringBuilder sb1 = new StringBuilder();

    sb1.append("Test de la propriété homomorphe : \n\n");
    for(int k=0;k<5;k++) {
        message = BigInteger.valueOf(Math.abs(random.nextInt()));
        message2 = BigInteger.valueOf(Math.abs(random.nextInt()));
        keys = keyGen(p, g, random);

        encrypt = encrypt(p, g, keys[1], message, random);
        encrypt2 = encrypt(p, g, keys[1], message2, random);

        decryptTotal = decrypt(encrypt[0].multiply(encrypt2[0]).mod(p
            ), encrypt[1].multiply(encrypt2[1]).mod(p), keys[0], p);
        productm1m2 = message.multiply(message2).mod(p);

        assert (productm1m2.equals(decryptTotal)) : "homomorphic
            property not checked";

        sb1.append("m1 = ").append(message).append("\tm2 = ").append(
            message2).append("\n");
        sb1.append("C and B decrypting gives : ").append(decryptTotal
            ).append("\n");
        sb1.append("(m1 * m2).mod(p) = ").append(productm1m2).append(
            "\n");
        sb1.append("Homomorphic property is checked : ").append(
            productm1m2.equals(decryptTotal)).append("\n\n");
    }
    try {
        bufferedWriter.write(sb1.toString());
    } catch (IOException e) {
        e.printStackTrace();
    }
    System.out.println(sb1);
    System.out.println("L'ensemble des tests se trouvent dans le
        fichier test.txt \n");
    System.out.flush();
}
```

Fichier test.txt :

```
Test de la fonction Euclide() :
a =
    160673951202871820768522432264413824281379564922918785062825317898190593802109840755896
     et
```

```
a.u + p.v = 1
results[0] == pgcd(a,p) = true
au + pv == pgcd(a,p) = true
gcd == 1 = true

a =
    178692563691092062429232399272601670879140757709494040679319707602107499034757832117396
     et
a.u + p.v = 1
results[0] == pgcd(a,p) = true
au + pv == pgcd(a,p) = true
gcd == 1 = true

a =
    178686959315835014564667448615041465056097822529956931805211106178921228239734109306721
     et
a.u + p.v = 1
results[0] == pgcd(a,p) = true
au + pv == pgcd(a,p) = true
gcd == 1 = true

a =
    157964703148738141022575258235549864420432938373980654168295926249857798008255841642664
     et
a.u + p.v = 1
results[0] == pgcd(a,p) = true
au + pv == pgcd(a,p) = true
gcd == 1 = true

a =
    132443799524215031501371190805309936511680265350275747540576235389840115523197368498347
     et
a.u + p.v = 1
results[0] == pgcd(a,p) = true
au + pv == pgcd(a,p) = true
gcd == 1 = true

Test de la fonction expMod() :
a =
    254874639746731658352177469257117525963509306924936613981775403082444384438300100650678
     et
expMod(p,g,a) =
    820875554545498253074091697411687465580112782698908014502226617306701651671581718217498

((a^g) mod p == expMod(p,g,a)) = true

a =
    291172166650839871081322552731146035618327533020005070608749047422957435612560140028643
     et
```

```
38 expMod(p,g,a) =
       7336390554596310710500391475313727411297370365957269350976012677392909493582984472638023

39 ((a^g) mod p == expMod(p,g,a)) = true
40
41 a =
       13858679129172923785821660828491743495626516421557161181732024403733638618347150486441961
        et
42 expMod(p,g,a) =
       5932817977378261606118670930734043260001896925827843148074325519573208334655139193758401

43 ((a^g) mod p == expMod(p,g,a)) = true
44
45 a =
       3167678535301827418726468968822179013308680309635621093255789585371612781800649808464681
        et
46 expMod(p,g,a) =
       9308891005513475728745477490838245389551267183175292161666155733324072998514813174122370

47 ((a^g) mod p == expMod(p,g,a)) = true
48
49 a =
       9385695092737179949957110899120130324633098162079382452012748818092404625739813850577868
        et
50 expMod(p,g,a) =
       1046652005287170050027743447994066979436643741013564635540856831177271545930811907442878

51 ((a^g) mod p == expMod(p,g,a)) = true
52
53 Test du chiffrement ElGamal :
54
55 Le message est : 1369399431
56 Le message chiffre est : C : 632760402 - et B : -1865444536
57 Le message dechiffre est : 1369399431
58 Message correctement dechiffre ? true
59
60 Le message est : 1494697761
61 Le message chiffre est : C : 2005268047 - et B : 155466468
62 Le message dechiffre est : 1494697761
63 Message correctement dechiffre ? true
64
65 Le message est : 2125611792
66 Le message chiffre est : C : -1588473933 - et B : -1907232047
67 Le message dechiffre est : 2125611792
68 Message correctement dechiffre ? true
69
70 Le message est : 1308864376
71 Le message chiffre est : C : 676242185 - et B : -663133218
72 Le message dechiffre est : 1308864376
```

```
Message correctement dechiffre ? true

Le message est : 934934231
Le message chiffre est : C : 493673307 - et B : -1753947215
Le message dechiffre est : 934934231
Message correctement dechiffre ? true

Test de la propriete homomorphe :

m1 = 815806361 m2 = 1303917819
C and B decrypting gives : 1063744450961446659
(m1 * m2).mod(p) = 1063744450961446659
Homomorphic property is checked : true

m1 = 2080500483 m2 = 587353701
C and B decrypting gives : 1221989658622337583
(m1 * m2).mod(p) = 1221989658622337583
Homomorphic property is checked : true

m1 = 1661024313 m2 = 649974597
C and B decrypting gives : 1079623608449376861
(m1 * m2).mod(p) = 1079623608449376861
Homomorphic property is checked : true

m1 = 1772248928 m2 = 1560486088
C and B decrypting gives : 2765569796616913664
(m1 * m2).mod(p) = 2765569796616913664
Homomorphic property is checked : true

m1 = 700002507 m2 = 1483795092
C and B decrypting gives : 1038660284274295644
(m1 * m2).mod(p) = 1038660284274295644
Homomorphic property is checked : true
```

Listing 10: le fichier test.txt