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 Java. Le langage 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 gérer des nombres entiers de grande taille.
Il est possible d'utiliser cette classe pour effectuer des soustractions, des modulos, des multiplications sur les entiers. La classe BigInteger permet de représenter des entiers sans les limitations de taille des types primitifs entiers, ou des enveloppes des types primitifs. La classe BigInteger offre en plus des opérations habituelles sur les entiers, 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 des informations sur la façon dont il a été généré
- un nombre aléatoire indépendant des autres nombres aléatoires générés précédemment.
- Il doit également être 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.
En Java, nous pouvons utiliser la bibliothèque "SecureRandom" pour générer des nombres aléatoires cryptographiquement sûrs. Voici comment on peut utiliser cette bibliothèque pour générer un nombre aléatoire cryptographiquement sûr :

```java
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 $a \cdot u + p \cdot v = 1$. 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).

```java
/**
 * Version iterative de l'algorithme étendu d'Euclide (eviter recursion, run faster)
 * @param a
 * @param b
 * @return {a,u,v} avec a = gcd(a,b) et coefficients u et v tq au + bv = pgcd(a,b)
 */
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);
        t = u;
        u = u1;
        //u = u0 - q * u1
        u1 = t.subtract(q.multiply(u1));
        t = v;
        v = v1;
        //v = v0 - q * v1
        v1 = t.subtract(q.multiply(v1));
        t = b;
        b = a.subtract(q.multiply(b));
        a = t;
    }
    return (a.intValue()>0)? new BigInteger[]{a,u,v} : new BigInteger[]{a.negate(),u.negate(),v.negate()};
}
```

Voir la fonction de test ci-dessous :

```java
public void test10000Times(BigInteger p, SecureRandom random) throws EuclideException {
    BigInteger a, gcd_ap, bezout;
    BigInteger[] results;
    StringBuilder result = new StringBuilder();

    result.append("Test de la fonction Euclide()  : \n");
    for(int k=0; k < 10000; k++) {
        a = new BigInteger(1024, random);
        results = euclideCompute(a, p);
        gcd_ap = a.gcd(p).abs();
        bezout = a.multiply(results[1]).add(p.multiply(results[2]));
        assert(results[0].equals(gcd_ap)):"le premier element du tableau ne contient pas le gcd(a,p)";
        assert(gcd_ap.equals(bezout)):"gcd(a,p) != au + pv";
        assert(BigInteger.valueOf(1).equals(bezout)):"equation de bezout ne donne pas 1";

        //sortie que pour les 5 premieres occurrences
        if(k < 5){

            result.append("a = ").append(a).append("\t et \n");
            result.append("a.u + p.v = ").append(bezout).append("\n");
            //verifie que pgcd(a,p) == results[0]
            result.append("results[0] == pgcd(a,p) = ").append(results[0].equals(gcd_ap)).append("\n");
            // Vérifie que a * u + b * v = GCD(a, p)
            result.append("au + pv == pgcd(a,p) = ").append(gcd_ap.equals(bezout)).append("\n");
            result.append("gcd == 1 = ").append(BigInteger.valueOf(1).equals(bezout)).append("\n\n");

        }
    }

    try {
        bufferedWriter.write(result.toString());

    } catch (IOException e) {
        e.printStackTrace();
    }
    System.out.println(result);
    System.out.flush();
}
```

Question 4 :

Implémentez la fonction ExpMod(). Testez-la sur 10000 valeurs différentes.
Réponse :

Nous avons implémenté la fonction ExpMod() en utilisant l'algorithme de l'exponentiation rapide.

```java
public BigInteger expMod(BigInteger p, BigInteger g, BigInteger a){
    BigInteger x = BigInteger.ONE;

    while(a.compareTo(BigInteger.ZERO) > 0) {   //tant que a>0
        if(a.testBit(0)) {  //a pair
            x = (x.multiply(g)).mod(p);
        }

        g = (g.multiply(g)).mod(p); //g = g**2 mod p
        a = a.shiftRight(1);    //a/=2
    }
    return x;
}
```

Voir la fonction de test ci-dessous :

```java
public void test10000Times(BigInteger p, BigInteger g, SecureRandom sr) {
    BigInteger a, ourExpMod, modPowofBigint;
    StringBuilder sb = new StringBuilder();

    sb.append("Test de la fonction expMod() : \n");
    for(int k=0; k<10000; k++){
        a = new BigInteger(500,sr);
        ourExpMod = expMod(p,g,a);
        modPowofBigint = g.modPow(a,p) ;
        assert(ourExpMod.equals(modPowofBigint)):"((a^g) mod p != expMod(p,g,a))";

        //5 premieres occurrences
        if(k<5){
            sb.append("a = ").append(a).append("\t et \n");
            sb.append("expMod(p,g,a) = ").append(ourExpMod).append("\n");
            sb.append("((a^g) mod p == expMod(p,g,a)) = ").append(ourExpMod.equals(modPowofBigint)).append("\n\n");
        }

    }
    try{
        bufferedWriter.write(sb.toString());
    }catch(IOException e){
        e.printStackTrace();
    }
    System.out.println(sb);
    System.out.flush();

}
```

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 Encrypt(), Decrypt().

```java
public BigInteger[] keyGen(BigInteger p, BigInteger g, SecureRandom rand) {
    BigInteger x = new BigInteger(1024, rand); // tire x au hasard (private-key)
    BigInteger X = exponentiationModulaire.expMod(p,g,x); // calcule X = g^x mod p (public-key)
    return new BigInteger[] {x, X};
}
```

```java
public BigInteger[] encrypt(BigInteger p, BigInteger g, BigInteger publicKey, BigInteger m, SecureRandom rand){
    BigInteger r = new BigInteger(1024, rand);
    BigInteger B = exponentiationModulaire.expMod(p, g, r); // calcule B = g^r mod p
    BigInteger C = m.multiply(exponentiationModulaire.expMod(p, publicKey, r)).mod(p); // calcule C = m * X^r mod p
    return new BigInteger[]{ C, B };
}
```

```java
public BigInteger decrypt(BigInteger C, BigInteger B, BigInteger x, BigInteger p) throws EuclideException {
    return (euclide.modInv(exponentiationModulaire.expMod(p,B,x),p)).multiply(C).mod(p);
}
```

Voir la fonction de test ci-dessous :

```java
public void test100Times(BigInteger p, BigInteger g, SecureRandom random) throws EuclideException {
    BigInteger message, message2;
    BigInteger[] keys, encrypt;
    StringBuilder sb1 = new StringBuilder();

    sb1.append("Test du chiffrement ElGamal  : \n\n");

    for(int k=0; k<100; k++){
        message = BigInteger.valueOf(Math.abs(random.nextInt()));      You, 16 hours ago • opti ne pas ecrire a chaque boucle
        keys = keyGen(p,g, random);

        encrypt = encrypt(p,g, keys[1], message, random);
        message2 = decrypt(encrypt[0],encrypt[1], keys[0], p);
        assert(message.intValue() == message2.intValue()):"message déchiffré différent";

        //5 premieres occurrences
        if(k < 5){
            sb1.append("Le message est : ").append(message.intValue()).append("\n");
            sb1.append("Le message chiffré est : C  : ").append(encrypt[0].intValue()).append("    -    et B  : ").append(encrypt[1].intValue())

            sb1.append("Le message déchiffré est : ").append(message2.intValue()).append("\n");
            sb1.append("Message correctement déchiffré ?  ").append(message.intValue() == message2.intValue()).append("\n\n");
        }
    }
    try {
        bufferedWriter.write(sb1.toString());
    } catch (IOException e) {
        e.printStackTrace();
    }
    System.out.print(sb1);
    System.out.flush();
}
```

Nous avons également implanté une fonction permettant de calculer l'inverse modulaire d'un nombre. (utilisée dans la fonction Decrypt())

```java
public BigInteger modInv(BigInteger a, BigInteger b) throws EuclideException {
    BigInteger[] resEuclide = euclideEtendu2(a,b);
    if(!resEuclide[0].equals(BigInteger.ONE)){
        throw new EuclideException(s: "No modular inverse possible");
    }else{
        return resEuclide[1].mod(b);
    }
}
```

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 pro-

priété :

```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 =
    160673951202871820768522432264413824281379564922918785062825317898190593802109840755899
     et
a.u + p.v = 1
results[0] == pgcd(a,p) = true
au + pv == pgcd(a,p) = true
gcd == 1 = true

a =
    178692563691092062429232399272601670879140757709494040679319707602107499034757832117399
     et
a.u + p.v = 1
results[0] == pgcd(a,p) = true
au + pv == pgcd(a,p) = true
gcd == 1 = true

a =
    178686959315835014564667448615041465056097822529956931805211106178921228239734109306726
     et
```

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

a =
    157964703148738141022575258235549864420432938373980654168295926249857798008255841642666
     et
a.u + p.v = 1
results[0] == pgcd(a,p) = true
au + pv == pgcd(a,p) = true
gcd == 1 = true

a =
    132443799524215031501371190805309936511680265350275747540576235389840115523197368498346
     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 =
    254874639746731658352177469257117525963509306924936613981775403082444384438300100650674
     et
expMod(p,g,a) =
    820875554545498253074091697411687465580112782698908014502226617306701651671581718217496

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

a =
    291172166650839871081322552731146035618327533020005070608749047422957435612560140028643
     et
expMod(p,g,a) =
    733639055459631071050039147531372741129737036595726935097601267739290949358298447263800

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

a =
    138586791291729237858216608284917434956265164215571611817320244037336386183471504864411
     et
expMod(p,g,a) =
    593281797737826160611867093073404326000189692582784314807432551957320833465513919375840

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

a =
```

```
    31676785353018274187264689688221790133086803096356210932557895853716127818006498084646
     et
expMod(p,g,a) =
    93088910055134757287454774908382453895512671831752921616661557333240729985148131741223

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

a =
    93856950927371799499571108991201303246330981620793824520127488180924046257398138505778
     et
expMod(p,g,a) =
    10466520052871700500277434479940669794366437410135646355408568311772715459308119074428

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

Test du chiffrement ElGamal :

Le message est : 1369399431
Le message chiffre est : C : 632760402 - et B : -1865444536
Le message dechiffre est : 1369399431
Message correctement dechiffre ? true

Le message est : 1494697761
Le message chiffre est : C : 2005268047 - et B : 155466468
Le message dechiffre est : 1494697761
Message correctement dechiffre ? true

Le message est : 2125611792
Le message chiffre est : C : -1588473933 - et B : -1907232047
Le message dechiffre est : 2125611792
Message correctement dechiffre ? true

Le message est : 1308864376
Le message chiffre est : C : 676242185 - et B : -663133218
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 1: le fichier test.txt