TP 2 – Cryptage

(DES – DSA – BlowFish – PBE – RSA – SHA – SHAMD5)

Dans ce TP, nous allons mettre en pratique quelques algorithmes connus de cryptage et décryptage de données.

2. Cryptage BlowFish (coup de poisson) :

Blowfish est un algorithme de chiffrement symétrique, c'est-à-dire à clef secrète, par blocs.

Il a été conçu par Bruce Schneier en 1993. Le concept est basé sur l'idée qu'une bonne sécurité contre les attaques de cryptanalyse peut être obtenue en utilisant de très grandes clés pseudo-aléatoires.

Blowfish présente une bonne rapidité d'exécution excepté lors d'un changement de clé, il est environ 5 fois plus rapide que Triple DES et deux fois plus rapide que IDEA. Malgré son âge, il demeure encore solide du point de vue cryptographique avec relativement peu d'attaques efficaces sur les versions avec moins de tours. La version complète avec 16 tours est à ce jour entièrement fiable et la recherche exhaustive reste le seul moyen pour l'attaquer.

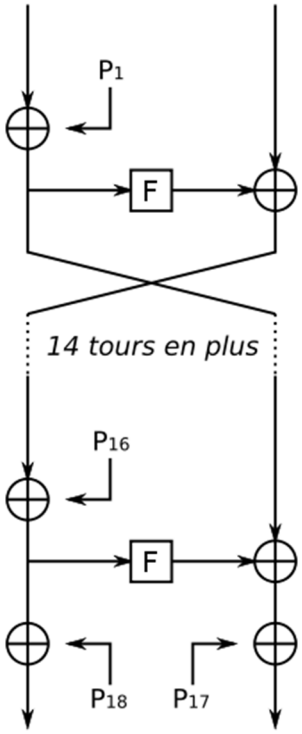
C'est l'un des premiers algorithmes de chiffrement dont l'utilisation était libre. Il est utilisé dans de nombreux logiciels propriétaires et libres (dont GnuPG et OpenSSH).

2.1. Mécanisme :

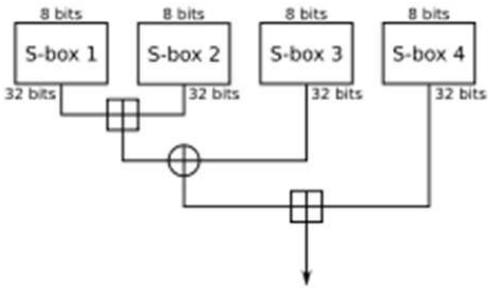
Il utilise une taille de bloc de 64 bits et la clé, de longueur variable, qui peut aller de 32 à 448 bits. Il est basé sur un schéma de *Feistel* avec 16 tours et utilise des *S-Boxes* de grande taille qui dépendent de la clé.

S-Boxes : Table de substitution utilisée dans un algorithme de chiffrement symétrique (permet de casser la linéarité de la structure de chiffrement et leur nombre varie selon les algorithmes.)

Voici le schéma de *Feistel* dans Blowfish et la fonction associée :



F : Fonction de BlowFish



2.2. Application et intégration dans le projet :

1) Créez le package **fr.doranco.cryptage.blowfish** puis créez dans ce package une classe abstraite nommée **`CryptageBlowFish`** avec :
   - Un constructeur sans paramètres
   - une constante statique finale `KEY_SIZE = 128`
   - un paramètre privé secretKey avec ses getter/setter

2) Créez les méthodes suivantes qui permettent de :

```java
/**
 * Retourne toutes les informations de la cl sous forme d'un tableau de bytes.
 * Elle peut ainsi être stockée puis reconstruite ultérieurement en utilisant la
 * méthode setSecretKey(byte[] keyData)
 */
public byte[] getSecretKeyInBytes() {
    return secretKey.getEncoded();
}
```

```java
/**
 * Permet de reconstruire la clé secète à partir de ses données, stockées dans
 * un tableau de bytes.
 */
public void setSecretKeyToBytes(byte[] keyData) {
    secretKey = new SecretKeySpec(keyData, "Blowfish");
}
```

3) Créez la méthode suivante qui permet de :

```java
/**
 * Permet de générer la clé à partir de l'algorithme BlowFish.
 */
public void generateKey() {
    try {
        KeyGenerator keyGen = KeyGenerator.getInstance("Blowfish");
        keyGen.init(KEY_SIZE);
        secretKey = keyGen.generateKey();
    } catch (Exception e) {
        System.out.println(e);
    }
}
```

4) Créez les méthodes de cryptage et de décryptage suivantes :

```
public byte[] crypt(String message) {
    try {
        byte[] messageInBytes = message.getBytes();
        Cipher cipher = Cipher.getInstance("Blowfish");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        return cipher.doFinal(messageInBytes);
    } catch (Exception e) {
        System.out.println(e);
    }
    return null;
}
```

```
public String decrypt(byte[] cipherText) {
    try {
        Cipher cipher = Cipher.getInstance("Blowfish");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        byte[] retourBytes = cipher.doFinal(cipherText);
        return (new String(retourBytes));
    } catch (Exception e) {
        System.out.println(e);
    }
    return null;
}
```

5) Créez une classe **`CryptageBlowFish_Main`** avec une méthode *`main()`* dans le package **fr.doranco.cryptage.blowfish** puis écrivez le code suivant qui permet de tester l'algorithme de cryptage BlowFish.

Il faudra adapter quelques lignes de ce main pour appeler correctement les méthodes de la classe **`CryptageBlowFish`** (modifier le constructeur en private) **.**

Il faudra également éventuellement adapter les valeurs de retour des méthodes de la classe **`CryptageBlowFish`.**

```
public static void main(String[] args) {

    String message = "Ceci est un texte d'exemple";
    System.out.println("message d'origine = " + message);
    CryptageBlowfish bf = new CryptageBlowfish();
    bf.generateKey();
    byte[] secretKey = bf.getSecretKeyInBytes();
    byte[] messageCrypte = bf.crypt(message);
    System.out.println("Message crypté = " + new BigInteger(messageCrypte));

    bf.setSecretKeyToBytes(secretKey);
    String messageDecrypte = bf.decrypt(messageCrypte);
    System.out.println("Message décrypté = " + messageDecrypte);
    if (!messageDecrypte.equals(message))
        System.out.println("Error: Message crypté != Message décrypté");
}
```

3. Signature DSA (Digital Signature Algorithm) :

Le DSA est un algorithme de signature numérique standardisé par le NIST aux États-Unis (*National Institute of Standards and Technology*), du temps où le RSA était encore breveté.

Cet algorithme fait partie de la spécification DSS (*Digital Signature Standard*) adoptée en 1993 (FIPS 186 : *Federal Information Processing Standards*). Une révision mineure a été publiée en 1996 (FIPS 186-1) et le standard a été amélioré en 2002 dans FIPS 186-2. Il est couvert par le brevet n° 5 231 668 aux USA (26 juin 1991) attribué à David Kravitz, ancien employé de la NSA. Actuellement, il peut être utilisé gratuitement.

3.1. Mécanisme :

Le processus se fait en trois étapes :

1. génération des clés ;
2. signature du document ;
3. vérification du document signé.

3.2. Application et intégration dans le projet :

1) Créez le package **fr.doranco.signature.dsa** puis créez dans ce package une classe abstraite nommée **`CryptageDSA`** avec :

   - Un constructeur privé sans paramètres
   - des constantes statiques finales suivantes :

     `KEY_SIZE = 1024 / CHEAT_TEXT = false / CHEAT_SIGNATURE = false`

2

```
try {
    String message = "Transfer $2000 to account S314542.0";
    byte[] text = message.getBytes();
    System.out.println("\nGenerating a pair of DSA keys...");
    KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
    keyPairGen.initialize(KEY_SIZE, new SecureRandom());
    KeyPair kp = keyPairGen.generateKeyPair();
    System.out.println("Signing the text...");
    Signature signature = Signature.getInstance("DSA");
    signature.initSign(kp.getPrivate());
    signature.update(text);
    byte[] sig = signature.sign();

    if (CHEAT_TEXT)
        text[0]++;
    if (CHEAT_SIGNATURE)
        sig[4]++; // changing sig[0] produces an exception

    System.out.println("\nVerifying the signature...");
    signature.initVerify(kp.getPublic());
    signature.update(text);
    boolean ok = signature.verify(sig);
    System.out.println("Signature is " + (ok ? "OK" : "NOT OK") + " !\n");
} catch (Exception e) {
    System.out.println(e);
}
```

Cette méthode main permet de signer un texte puis vérifie la signature. Les constantes `CHEAT_TEXT` et `CHEAT_SIGNATURE` permettent de modifier respectivement le texte et la signature (après l'opération de signature) afin de s'assurer que toute tentative de fraude soit détectée.

Note: Le message peut-être de longueur quelconque; il n'y a pas besoin de faire un hash avant d'appliquer le processus de signature.

3) Créez la classe **`SignatureDSA_Main`** avec la méthode *`main()`* précédente et réorganisez votre code dans la classe **`SignatureDSA`** de telle manière que cette dernière expose trois méthodes permettant de réaliser la signature digitale utilisant l'algorithme DSA (*`generateKey`*, *`signDocument`*, *`verifySignedDocument`*). Vous devez définir les bons paramètres des méthodes ainsi que les bonnes valeurs de retour.

   Réorganisez maintenant votre code du *`main()`* de la classe **`SignatureDSA_Main`** pour faire appel à ces trois méthodes dans l'ordre puis testez l'ensemble.

4. Cryptage PBE (Password Base Encryption) :

Ce type de cryptage permet d'encrypter un texte en utilisant l'algorithme PBE.

Le risque d'utiliser un mot de passe directement comme clé est d'obtenir des motifs dans le ciphertext.

L'algorithme PBE résout le problème en "salant" (salting) le mot de passe (c'est-à-dire en ajoutant des données aléatoires), puis en lui appliquant successivement un certain nombre de fois (nombre d'itérations) une fonction de hachage (typiquement MD5).

Les deux paramètres utilisés, "salt" (un tableau de bytes aléatoires) et "iterations" (le nombre de fois qu'on applique MD5) doivent être enregistrés comme paramètres du Cipher afin de pouvoir être utilisés lors du décodage.

Dans ce programme le salt est généré de manière aléatoire à chaque exécution, ce qui explique que le ciphertext est différent à chaque fois même si on encrypte le même message avec le même mot de passe.

4.1. Application et intégration dans le projet :

1) Créez le package **fr.doranco.cryptage.pbe** puis créez dans ce package une classe abstraite nommée **`CryptagePBE`** avec un constructeur privé sans paramètres.
2) Créez la méthode *`main()`* qui permet de développer le code suivant et le tester :

```java
String message = "Vive les cours Java";
System.out.println("message à crypter = " + message);

char[] password = {'t', 'h', 'e', ' ', 'p', 'a', 's', 's'};

// le salt est choisis aléatoirement et le nombre d'itérations par défaut vaut 10
PBEKeySpec keySpec = new PBEKeySpec(password);

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
SecretKey secretKey = keyFactory.generateSecret(keySpec);

Cipher cipher = Cipher.getInstance("PBEWithMD5AndDES");
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
byte[] messageInByte = message.getBytes();
byte[] ciphertext = cipher.doFinal(messageInByte);
System.out.println("message crypté = " + new BigInteger(ciphertext));

// on récupère les paramétres, comme le salt et le nombre d'itérations
AlgorithmParameters params = cipher.getParameters();
cipher.init(Cipher.DECRYPT_MODE, secretKey, params);
byte[] messageDecrypteInByte = cipher.doFinal(ciphertext);
String messageDecrypte = new String(messageDecrypteInByte);
System.out.println("message décrypté =  " + messageDecrypte);
```

3) Créez la classe **`CryptagePBE_Main`** avec la méthode *`main()`* précédente et réorganisez votre code dans la classe **`CryptagePBE`** de telle manière que cette dernière expose les deux méthodes permettant de réaliser le cryptage et décryptage utilisant l'algorithme PBE (*`crypt, decrypt`*). Vous devez définir les bons paramètres de ces méthodes ainsi que les bonnes valeurs de retour.

   Réorganisez maintenant votre code du *`main()`* de la classe **`CryptagePBE_Main`** pour faire appel à ces deux méthodes dans l'ordre puis testez l'ensemble.