Securité Informatique : Cryptographie

Allan BOOZ, Yanis BOUDEMIA ETTOUIL

16 avril 2020

Résumé

Il s'agit de réaliser une application mobile Android qui permet de crypter/décrypter par des méthodes différentes des messages saisis par l'utilisateur.

1 Introduction

L'objectif de notre application est de permettre à son utilisateur de crypter et décrypter des messages des chiffrements suivants : Atbash, César, Vigenère, Homophone de Polybe, Hill, DES et RSA.

2 Le corps du projet

2.1 Atbash

2.1.1 Cryptage

Atbash est un chiffrement qui écrit l'alphabet en sens contraire. C'est-à-dire qu'avec notre message, par exemple : "Hello", on recherche dans l'alphabet l'indice de "H", et dans l'alphabet inversé on regarde à cet indice l'alphabet correspondant.

En java, on va créer une liste de String représentant l'alphabet et une autre liste de String représentant l'alphabet inverse.

Ensuite, le constructeur de notre classe va prendre en argument le message qu'on souhaite crypter/décrypter et efin à travers une boucle chercher le caractère inverse pour chaque caractère dans notre message.

```
            for (int i = 0; i < mot.length(); i++) {
            res += reverse_alpha[indexOf(alphabets, String.valueOf(mot.charAt(i)))];
        }
```

```
 public int indexOf(final ArrayList<String> tab, final String val) {...}
```

est une fonction qui recherche dans une liste l'indice du mot passer en paramètre dans cette liste.

2.1.2 Decryptage

Le principe de déchiffrement d'Atbash est le même que son chiffrement. Par exemple : si on a chiffré que A était égale à Z alors l'inverse de Z c'est A.(crypté : A -> Z ; décrypté : Z -> A)

2.2 César

2.2.1 Cryptage

César est un chiffrement qui à l'origine déplaçait les lettres de 3 positions dans l'alphabet. Par exemple pour un message : "A" avec un décalage de 3, son cryptage serait : "D".
Ainsi en Java, nous allons parcourir notre liste d'alphabet et rechercher grâce à notre fonction indexOf() l'indice de chaque lettre qui compose notre mot, rajouter notre décalage et concatener tous les nouvelles lettres dans un type String.

```
mess_crypt += alphabets.get(indexOf(alphabets, String.valueOf(mot.charAt(i))) + decalage);
```

2.2.2 Decryptage

Le principe de déchiffrement de César est le même que son chiffrement mais au lieu de décaler de N-nombre dans l'alphabet, on recule de N-nombre.

```
mess_decrypt += alphabets.get(indexOf(alphabets, String.valueOf(mot.charAt(i))) - decalage);
```

2.3 Vigenère

2.3.1 Cryptage

Le chiffrement de Vigenère consiste à additionner la clé au message. C'est-à-dire que l'addition de lettre est réalisée par des nombres, les valeurs des lettres sont l'indice de l'alphabet + ascii extended. Le résultat est donné modulo 256 : si le résultat est supérieur ou égal à 256, on soustrait 256 au résultat. (256 est la longueur de l'alphabet + ascii extended).

Exemple : message = "ABCD", la première lettre A (=65), la clé = "DCBA" la première lettre D (=68) et les ajouter 65+68=133. On conserve la valeur et on continue avec la lettre suivante du message et de la lettre suivante de la clé. Arrivé à la fin de la clé, on recommence au début de celle-ci.

```
poids.add(indexOf(alphabets, String.valueOf(mot.charAt(i))) +
          indexOf(alphabets, String.valueOf(clef.charAt(x))));
```

Pour chaque nombre obtenu (qui doit avoir une valeur entre 0 et 255), on fait correspondre la lettre ayant le même rang dans l'alphabet + ascii extended.

```
for (int y = 0; y < poids.size(); y++) {
    mess_crypt += alphabets.get(poids.get(y));
}
```

Pour faire correspondre la longueur du texte à la clé, celle-ci est répétée à l'infini : DCBADC-BADCBA...

```
int x;
if (x > clef.length() - 1) {
    x = 0;
}
```

2.3.2 Decryptage

Pour déchiffrer Vigenère, on prend la première lettre du message et la première lettre de la clé, et on soustrait leurs valeurs. Si le résultat est négatif, on ajouter 256 au résultat (où 256 est la longueur de l'alphabet + ascii extended), le résultat correspond au rang dans l'alphabet de la lettre claire.

2.4 Homophone de Polybe

2.4.1 Cryptage

Afin de chiffrer l'homophone de Polybe, nous devons créer une matrice composée des 26 lettres de l'alphabet + 10 chiffres allant de 0 à 9 soit 36 caractères.

```
public String[][] init_matrice(String mot) {
    String[][] matrice = new String[LIGNE][COLONNE];
    String l = mot + alphabet;
    l = liste_sans_doublon(l);
    int id = 0;
    for (int x = 0; x < LIGNE; x++) {
        for (int y = 0; y < COLONNE; y++) {
            matrice[x][y] = String.valueOf(l.charAt(id));
            id++;
        }
    }
    return matrice;
}
```

On décide de remplir cette matrice tout d'abord avec notre message sans doublon, puis par ordre alphabétique. En java nous créerons une fonction qui enleve les doublons d'un mot avant de l'injecter dans notre matrice.

```
public String liste_sans_doublon(String text) {
    String res = "";
    for (int i = 0; i < text.length(); i++) {
        res += (res.contains(text.charAt(i) + "") ? "" : text.charAt(i) + "");
    }
    return res;
}
```

Puis, pour chiffrer le mot on remplace une lettre par un couple de lettres dont la première se situe sur la ligne de la lettre initiale et la seconde se situe sur la colonne de la lettre initiale.
Exemple :

```
Message : Hello
Matrice :
    H E L O A B
    C D F G I J
    K M N P Q R
    S T U V W X
    Y Z 0 1 2 3
    4 5 6 7 8 9

couple de la lettre "H" : H : [(C,E), (B,Y), (L,4), ...]
```

Ainsi en Java, on va créer une fonction couple qui va rechercher pour un caractère donné une liste de couple possible. Pour cela, on localise notre caractère dans la matrice et on mémorise la ligne et la colonne.

```
public ArrayList<List> couple(String[][] matrice, String l) {
    ...

    for (int x = 0; x < LIGNE; x++) {
        for (int y = 0; y < COLONNE; y++) {
            if (matrice[x][y].equals(String.valueOf(lettre))) {
                line = x;
                col = y;
            }
        }
    }
```

Puis on stocke dans une liste tous les caractères se trouvant sur la même ligne que le caractère donné et de même pour ceux se trouvant sur la même colonne que le caractère donné.

```
    for (String i : matrice[line]) {
        if (!i.equals(lettre)) {
            list_line.add(i);
        }
    }
    for (int x = 0; x < LIGNE; x++) {
        String i = matrice[x][col];
        if (!i.equals(lettre)) {
            list_col.add(i);
        }
    }
```

Enfin on renvoie une liste de couple :

```
    ...

    for (String x : list_line) {
        for (String y : list_col) {
            liste = new ArrayList<String>();
            liste.add(x);
            liste.add(y);
            bloc.add(liste);
        }
    }
    return bloc;
}
```

Ayant maintenant toutes les fonctions utiles, nous pouvons maintenant écrire une fonction crypter qui va simplement renvoyer la matrice elle-même ainsi que les coordonnées de chaque caractère qui compose le mot. Etant donné que la fonction couple renvoie une liste de couple, il s'agira de choisir aléatoirement un couple en utilisant random.
Dans les logs, on peut apercevoir la matrice et les coordonnées du mot qu'on recherche. Cependant par soucis d'adaptation au projet mobile, nous decidons de renvoyer à l'utilisateur la matrice + les coordonnées dans une chaine de caractères.

2.4.2 Decryptage

La clé de chiffrement étant la matrice et les coordonnées renvoyés en chaine de caractère par le cryptage, le déchiffrement se fait donc par reconstitution de la matrice grâce aux 36 premiers caractères et le reste de la clé correspondant aux coordonnées de chaque caractère du message.

Et enfin grâce aux coordonnées, on recherche dans la matrice les caractères correspondant, ce qui nous donne notre message décrypter.

2.5 Hill

2.5.1 Cryptage

Le chiffrement de Hill consiste à remplacées chaque lettre par leur rang dans l'alphabet. Par exemple : A devient 0, B devient 1,..., Z devient 25 etc... . On regroupe les nombres obtenus par m (exemple : m=2). Pour chaque bloc de m nombres à coder x1,x2,...,xm, on calcule le texte codé en effectuant des combinaisons linéaires :

```
public ArrayList<Integer> combinaison_lineaire(ArrayList<List> mot, int a, int b,
                                               int c, int d) {
    ArrayList<Integer> res = new ArrayList<Integer>();

    for (int i = 0; i < mot.size(); i++) {
        final int x1 = (Integer) mot.get(i).get(0);
        final int x2 = (Integer) mot.get(i).get(1);
        final int y1 = ((a * x1) + (b * x2)) % 256;
        final int y2 = ((c * x1) + (d * x2)) % 256;
        res.add(y1);
        res.add(y2);
    }
    return res;
}
```

Pour chaque combinaison, on le module à 256 (table ascii extended) et on fait correspondre ces indices à la table ascii pour avoir notre message crypté.

```
pos_lettre = combinaison_lineaire(indice_de(mot, m), a, b, c, d);
for (int i = 0; i < pos_lettre.size(); i++) {
    crypter += alphabets.get(pos_lettre.get(i));
}
```

2.5.2 Decryptage

Pour déchiffrer un message codé en Hill, connaissant la clé, on procède exactement de la même façon. On découpe donc en blocs de m lettres, mais cette fois il faut inverser les relations données par les combinaisons linéaires.
La clé est composé de valeurs a, b, c, d. Grâce à cela, on fait une soustraction de produit ((a*d) - (b*c)) et avec ce résultat, on cherche un déterminant A. Ainsi on fait le pgcd entre ce résultat et 256 modulo 256.

```
  public ArrayList<Integer> pgcd(int a, int b) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    int r = a;
    int u = 1;
    int rp = b;
    int up = 0;
    while (rp != 0) {
        int q = r / rp;
        int rs = r;
        int us = u;
        r = rp;
        u = up;
        rp = (rs - q * rp);
        up = (us - q * up);
    }
    res.add(r);
    res.add(u);
    return res;
}

public int detA(int a) {
    ArrayList<Integer> res = pgcd(a, 256);
    int y = (int)res.get(1);
    return ((y%256) + 256 )% 256;
}
```

Ayant notre déterminant, on peut maintenant inverser les relations, trouver de nouvelle valeurs de a, b, c, d et refaire une combinaison linéaire afin de décrypter le message codé.

```
        int inv_A = detA(((a * d) - (b * c)));
        d = (inv_A * d) % 256;
        b = (((inv_A * (-b)) % 256) + 256) % 256;
        c = (((inv_A * (-c)) % 256) + 256) % 256;
        a = (inv_A * a) % 256;

        pos_lettre = combinaison_lineaire(indice_de(mot, m), d, b, c, a);

        for (int i = 0; i < pos_lettre.size(); i++) {
            decrypter += alphabets.get(pos_lettre.get(i));
        }
```

2.6 DES

2.6.1 Cryptage

DES est un chiffrement par blocs et chiffre les données en blocs de 64 bits chacun, ce qui signifie que 64 bits de texte brut sont transmis en entrée à DES, qui produit 64 bits de texte chiffré. Le même algorithme et la même clé sont utilisés pour le chiffrement et le déchiffrement, avec quelques différences. La longueur de clé est de 56 bits.

Pour ce projet, l'utilisateur va devoir entrer une clé en hexadécimale (16 bits) et un texte clair, son message va lui être traduit en code binaire de 64 bits. D'où la nécessité de créer des fonctions qui convertit l'ASCII en Hexadécimale, l'Hexadécimale en Binaire et vice-versa.

DES se compose de 16 étapes. A chaque tour on effectue les étapes de substitution et de transposition. Il faudra donc générer 16 clés différentes.

```
// Generer 16 clé différentes
public String[] getKeys(String key){
    String keys[] = new String[16];
    // 1er clé permuter
    key = permutation(PC1, key);
    for (int i = 0; i < 16; i++) {
        key = Permutation_circulaire(  key.substring(0, 7), rotations[i]) +
            Permutation_circulaire(key.substring(7, 14), rotations[i]);
        // 2ième clé permuter
        keys[i] = permutation(PC2, key);
    }
    return keys;
}
```

Les étapes générales du DES :
1. Dans la première étape, le bloc de texte brut (message de 64 bits) est remis à une fonction de permutation initiale (PI).

```
// Permutation entre le mot en hexadecimale et la table choisit.
public String permutation(int[] sequence, String mot){
    String res = "";
    mot = hextoBin(mot);
    for (int i = 0; i < sequence.length; i++)
        res += mot.charAt(sequence[i] - 1);
    res = binToHex(res);
    return res;
}
```

2. Ensuite, la permutation initiale (PI) produit deux moitiés du bloc permuté. On indique le texte brut à gauche (LEFT) et le texte brut à droite (RIGHT).

3. Maintenant, chaque LEFT et RIGHT doivent passer par 16 tours de processus de cryptage.

```
// 16 itérations
for (int i = 0; i < 16; i++) {
    message = round(message, keys[i], i);
}
```

4. Au final, LEFT et RIGHT sont rejoints et une permutation finale (PF) est effectuée sur le bloc combiné

```
// 32-bit swap
message = message.substring(8, 16) + message.substring(0, 8);

// Permutation finale
message = permutation(PI_reverse, message);
```

5. Le résultat de ce processus produit un texte chiffré.

2.6.2 Decryptage

Le même algorithme est utilisé pour déchiffrer sauf qu'on doit parcourir les tours à l'envers.

```
// 16 itérations
for (int i = 15; i > -1; i--) {
    message = round(message, keys[i], 15 - i);
}
```

2.7 RSA

2.7.1 Cryptage

Le chiffrement RSA est un chiffrement asymétrique : il utilise une paire de clés composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer des messages confidentiels. Bien que le chiffrement de base fait intervenir 2 personnes, nous avons décidé de les rassembler dans une seul et unique classe pour faciliter la communication des clés. Dans un premier temps on met en place 2 très grands entiers premiers p et q de façon aléatoire. On fait ensuite la multiplication de ces entiers pour obtenir notre premiere clé publique N. Toujours avec p et q on va choisir un entier e tel que e soit premier avec (p-1).(q-1), on obtient ainsi notre deuxième clé publique. Nous pouvons désormais crypter notre message grace à N et e

```
public static byte[] encrypt(byte[] message){
        return (new BigInteger(message)).modPow(e, N).toByteArray();
    }
```

2.7.2 Decryptage

Pour le décryptage nous avons besoin de recevoir le message à décrypter ainsi que les clés p et q qui ont servi à crypter ce même message. C'est pourquoi notre programme effectue le cryptage d'un message ainsi que son décryptage à la suite. Car si nous ne sommes pas au courant des clés p et q qui ontpermis le cryptage du message, il sera impossible d'obtenir un décryptage correct. A la réception du message à décrypter nous allon créer un nouvelle clé qui sera celle du décryptage qu l'on appellera $d$qui correspond à l'inverse de e modulo $(p-1)(q-1)$

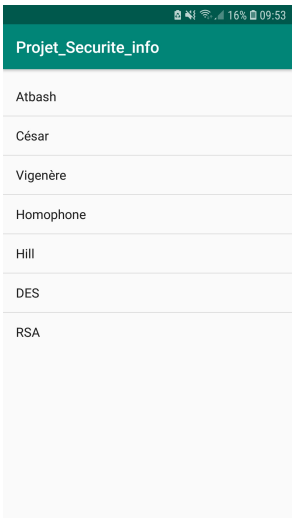
```
phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
d = e.modInverse(phi);
```

Une fois d créée il ne nous reste plus qu'à déchiffrer en utilisant N et d de la même façon que pour le cryptage

```
public static byte[] decrypt(byte[] message){
        return (new BigInteger(message)).modPow(d, N).toByteArray();
    }
```

2.8 MainActivity

Notre application se présente sous la forme de 2 activity, la premiere étant *MainActivity*. Cette activity n'est rien de plus qu'une ListView dont chaque item de la liste sera chacun de nos chiffrementspréalable dans les ressources).

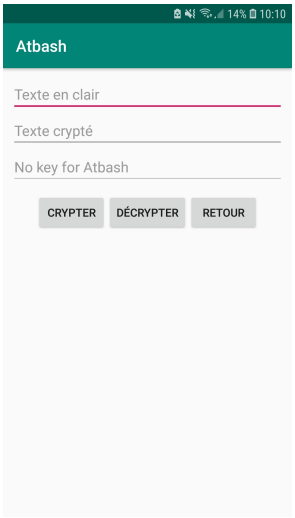


Il ne nous reste plus qu'à cliquer sur le chiffrement souhaité pour obtenir une redirection vers la nouvelle acitivity correspondante.

```
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView, View view, int i, long l) {
                Intent intent = new Intent(MainActivity.this, SecondActivity.class);
                intent.putExtra("crypto", listView.getItemAtPosition(i).toString());
                startActivity(intent);
            }
        });
```

2.9 SecondActivity

Cette Seconde activity sera celle qui permettra de crypter et décrypter selon le chiffrement choisi à l'étape d'avant. Elle se présente comme ceci :



Bien que son style ne change pas selon le chiffrement choisi, les actions des bouttons, elles, y correspondent bien. En effet pour chaque chiffrement nous avons donc une condition permettant de savoir sur quel chiffrement se baser. Et pour cela nous verifions le Titre de l'activity en question et mettons à jour les actions des bouttons selon le chiffrement correspondant.

```
if(this.getSupportActionBar().getTitle().equals("César"))
```

Par exemple si l'utilisateur aurait choisi d'utiliser Atbash on change les actions des bouttons comme ceci :

```
btn_crypter.setOnClickListener(new View.OnClickListener() {
                    @Override
                    public void onClick(View view) {
                        String mess = mess_clair.getText().toString();
                        if(mess.isEmpty()){
                            (...)
                        }else {
                            Atbash a = new Atbash(mess);
                            resultat.setText(a.getResultat());
                            // Au cas ou si on veut decrypter un mot de l'ASCII extended
                            mess_crypter.setText(resultat.getText());
                         } }});
        btn_decrypter.setOnClickListener(new View.OnClickListener() {
                    @Override
                    public void onClick(View view) {
                        String mess = mess_crypter.getText().toString();
                        if(mess.isEmpty()){
                            (...)
                        }else {
                            Atbash a = new Atbash(mess);
                            resultat.setText(a.getResultat());
                        } }});
```

3 Bibliographie

1. https://www.dcode.fr/fr
2. http://www.bibmath.net/crypto/
3. https://www.nymphomath.ch/crypto/