

# Sécurité Informatique - CC1

Frédéric Fabre Ferber , M1 Informatique

9 avril 2020

## Résumé

Ce rapport résume le travail fait dans le cadre du cours de Sécurité Informatique. Il consiste en l'élaboration de différents algorithmes de chiffrement via Android Studio. Nous présenterons ceux mentionnés ci-dessous :

- Atbash
- César
- Vigenère
- Playfair
- Hill
- Transposition Rectangulaire
- DES
- Un algorithme surprise
- L'algorithme RSA

## 1 Fonctionnement de l'application

L'application Android pour ce projet est composé de deux activités :

- **MainActivity** qui consiste tout simplement en l'affichage d'une liste contenant tous les algorithmes de chiffrement via une **ListView**. Quand on clique sur un des éléments, on envoie son id et son nom à l'activité **CryptoDefaultActivity** avec un **Intent** (voir figure 1).
- **CryptoDefaultActivity**, c'est elle qui gère le chiffrement et le déchiffrement pour chaque algorithme. Elle se compose de 3 champs (*Message clair*, *Message chiffré* et *clé*). Un autre champ peut se rajouter ou s'enlever en fonction de l'algorithme (ex : Hill, Atbash). L'affichage est géré par le **layout** *crypto\_default\_layout.xml* (voir figure 2 et 3) . Quand on appuie sur le bouton chiffrer ou déchiffrer, on va alors modifier le champ **message chiffré** en sélectionnant l'algorithme correspondant à l'id reçu. Cela nous permet de n'avoir qu'un seul layout et qu'une seule activité pour le chiffrement/déchiffrement.

Aperçu du code :

```
if (!(message.isEmpty())){  
  
switch (chiffrementID){  
  
    /* Atbash */  
    case 0 :  
        messageChiffreEdit.setText(Crypto.atbash(message));  
        break;  
  
    /*César 3*/
```

```

        case 1 :
        if(!(cle.isEmpty())){
            int decalage = Integer.parseInt(cle);
            messageChiffreEdit.setText(Crypto.cesar(message , decalage ,chiffre));
            break;
        }

        /*Vigenère*/
        case 2 :

        if(!(cle.isEmpty())){
            messageChiffreEdit.setText(Crypto.vigenere(message , cle , chiffre));
            break;
        }

        .....

```

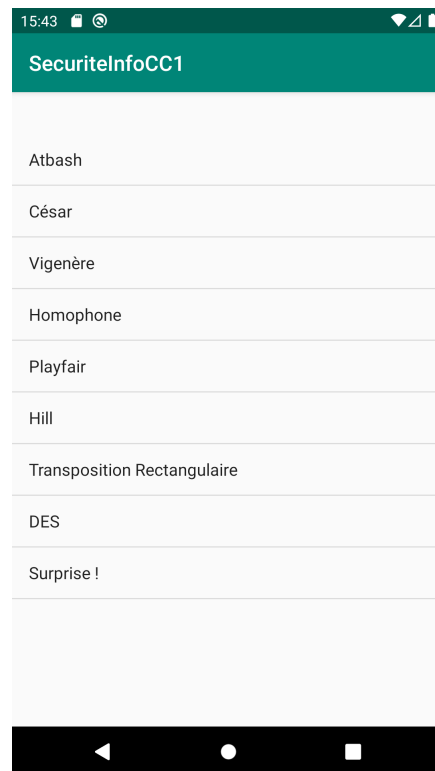
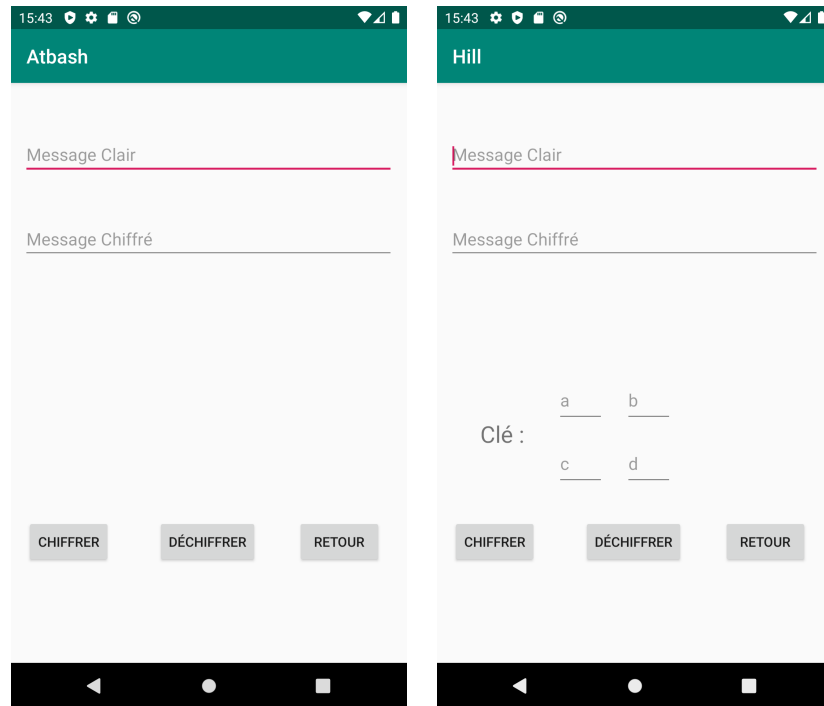


FIGURE 1 – Vue de l'activité principale **MainActivity**



Vue en sélectionnant Atbash

Vue en sélectionnant Hill

FIGURE 2 – Différentes vues relatives à l'activité **CryptoDefaultActivity**

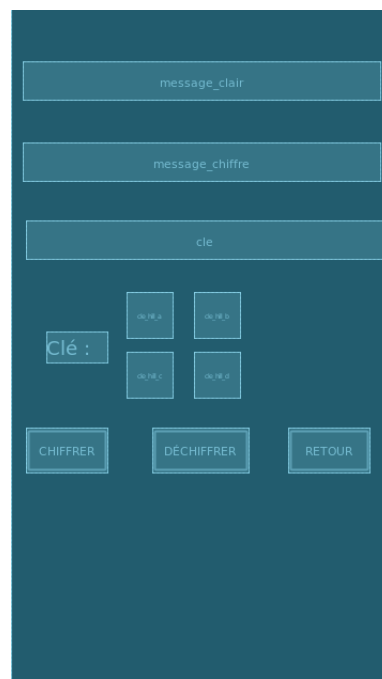


FIGURE 3 – Structure de l'activité **CryptoDefaultActivity** avec le fichier *crypto\_default\_layout.xml*

## 2 Algorithme de chiffrement/déchiffrement

Tous les algorithmes que nous allons mentionner sont présents dans la classe **Crypto**. Elle contient toutes les méthodes publiques permettant de chiffrer/déchiffrer (pour chaque algorithme). Chaque algorithme contient 3 paramètres :

- Message : Un **String** qui est le message à chiffrer ou déchiffrer.
- La clé : Un **String** (ou un **int[]** pour Hill).
- Un booléen **chiffre** qui va chiffrer s'il est vrai et déchiffrer sinon.

Elles renvoient toutes un **String** qui est le message chiffré ou déchiffré.

### 2.1 Table ASCII étendue

Nous travaillons pour la plupart des algorithmes de chiffrement avec la **table ASCII étendue**. En effet, elle est un prolongement de la table ASCII de base comprenant les codes de 0 à 127, en rajoutant 128 nouveaux caractères. Nous disposons des codes allant de 0 à 255 (**voir 4**).

Cependant plusieurs encodages de cette table existent, nous utiliserons la norme **CP437**. C'est la classe **ExtendedASCII.java** qui représente la table ASCII étendue. Elle contient plusieurs méthodes :

- **char getChar(int)** qui renvoie le caractère correspondant au code de la table ASCII étendue.
- **int getASCIICode(char)** qui renvoie le code correspondant au caractère de la table ASCII étendue.
- **int[] StringToAsciiCodeTable(String)** qui va transformer une chaîne de caractère en tableau d'entiers. Chaque entier étant le code ASCII correspondant.
- **String AsciiCodeTableToString (int[])** qui fait l'inverse de la méthode précédente.

Pour la plupart des algorithmes de chiffrement nous allons utiliser principalement ces deux dernières méthodes. En effet, le but sera de transformer le message en tableau d'entiers contenant les différents codes ASCII étendue avec **StringToAsciiCodeTable**. Puis, de travailler dessus pour enfin restituer la chaîne représentant le message chiffré ou déchiffré avec **AsciiCodeTableToString**. Ces deux méthodes gèrent également les caractères non affichables. La table contient plusieurs caractères de contrôle soit les codes ASCII de 0 à 31, le code 127 et le code 255. Ils seront représentés dans la chaîne par leur écriture en hexadécimal. Exemple :

- **AsciiCodeTableToString(new int[]{0 , 61 , 62 , 255})** renvoie la chaîne `\x00ab\xff`.
- **StringToAsciiCodeTable("Cc\x7f")** renvoie le tableau `[43,63,127]`.

128	Ç	144	É	160	á	176	⌘	193	⌞	209	⌠	225	β	241	±
129	ü	145	æ	161	í	177	⌙	194	⌟	210	⌡	226	Γ	242	≥
130	é	146	Æ	162	ó	178	⌚	195	⌠	211	⌢	227	π	243	≤
131	â	147	ø	163	ú	179		196	—	212	⌣	228	Σ	244	∫
132	ä	148	ö	164	ñ	180	⌡	197	⌢	213	⌤	229	σ	245	∫
133	à	149	ò	165	Ñ	181	⌣	198	⌢	214	⌥	230	μ	246	÷
134	â	150	û	166	•	182	⌤	199	⌣	215	⌦	231	τ	247	≈
135	ç	151	ù	167	°	183	⌥	200	⌤	216	⌧	232	Φ	248	°
136	ê	152	—	168	¿	184	⌧	201	⌥	217	⌨	233	Θ	249	·
137	ë	153	Ö	169	—	185	⌨	202	⌦	218	〈	234	Ω	250	·
138	è	154	Û	170	¬	186	〈	203	⌧	219	■	235	δ	251	√
139	ï	156	£	171	½	187	〉	204	⌨	220	■	236	∞	252	—
140	î	157	¥	172	¾	188	⌫	205	=	221	■	237	φ	253	²
141	ì	158	—	173	¡	189	⌬	206	〈	222	■	238	ε	254	■
142	Ä	159	ƒ	174	«	190	⌭	207	〈	223	■	239	∩	255	
143	Å	192	Ł	175	»	191	⌮	208	〉	224	α	240	≡		

FIGURE 4 – 128 nouveaux caractères de la table ASCII étendue avec la norme CP437

Nous pouvons maintenant parler de chaque algorithme de chiffrement.

## 2.2 Atbash

Le chiffage Atbash est un algorithme où l'on va substituer une lettre par son inverse dans la table ASCII étendue. Par exemple le caractère "2" avec pour rang 50 dans la table devient le caractère ayant pour rang 200 (250-50). En général cela consiste en un décalage de  $250 - i$ . Le chiffage et déchiffage est totalement le même et est géré par la méthode `String atbash(String message)` de la classe `Crypto`.

### 2.2.1 Exemple

- On chiffre la chaîne "Heavy Metal" représenté par les codes [72,101,97,118,121,32,77,101,116,97,108] deviennent [183,154,158,137,134,223,178,154,139,158,147] qui correspond à la chaîne "". (voir figure 6)
- On déchiffre la chaîne obtenu et on ré-obtient "Heavy Metal". voir figure 7)

## 2.3 César

Le chiffage de César consiste à substituer un caractère par un autre. On décale chaque caractère du message  $M$  par un certain nombre  $d$  pour obtenir le message chiffré  $C$ . On utilise la méthode `String vigenere(String,String,boolean)` de la classe `Crypto`. Si on chiffre, le nouveau caractère  $C_i$  sera :

$$C_i = (M_i + d) \bmod 256$$

Si on déchiffre, le nouveau caractère  $C_i$  sera :

$$C_i = (M_i - d) \bmod 256$$

### 2.3.1 Exemple

- Si on chiffre la chaîne "Blink" représenté par les codes [66,108,105,110,107] avec un décalage de 182 on obtient [248,34,31,36,33] qui donne la chaîne ""\x1f \$! (le code 31 représentant un caractère non affichable). (Voir figure 7)
- Si on déchiffre cette chaîne avec le même décalage on ré-obtient "Blink". (Voir figure 7)

## 2.4 Vigenère

Le chiffement de Vigenère marche comme celui de César. (voir 2.3) À la différence que pour chaque caractère  $M_i$  du message  $M$  on décalera du numéro du  $i_{eme}$  caractère (modulo la longueur de la clé) de la clé  $c$ . Donc pour  $d = c_i \bmod longueur(c)$  si on chiffre ou déchiffre le nouveau caractère  $C_i$  utilise la même formule que celle de César vu précédemment.

### 2.4.1 Exemple

- Si on prend le message "Iron" représentée par les codes [73,114,111,110] avec la clé "Maiden" représentée par les codes [77,97,105,100,101,110]. Le premier caractère est décalé de 77, le deuxième de 97 etc ... Ce qui donne les codes [150,211,216,210] ce qui donne la chaîne "". (voir figure 8)
- Si on reprend cette chaîne mais en déchiffrant on ré-obtient le message d'origine. (voir figure 8)

## 2.5 Chiffre de Playfair

Le chiffrement de Playfair consiste en l'utilisation d'un carré de Polybe que l'on va renverser pour construire un **chiffre de Playfair**. Grace à celui ci on va substituer chaque couple de lettres avec un autre couple de lettres. On utilise comme dit précédemment un **carré de Polybe** à la différence qu'ici, c'est un tableau de 6x6 cases contenant les lettres de l'alphabet et les chiffres.

On utilise la fonction interne à la classe **Crypto** `char[][] cleToPolybe(String cle)` qui transforme une clé en un carré de polybe et `char[][] polybeToPlayfair(char[][] polybe)` qui va prendre le carré de polybe et réarranger celui-ci en lisant chaque colonne pour construire chaque ligne du chiffre de Playfair.

Pour le chiffage et le déchiffage l'algorithme marche exactement pareil pour chaque couple de lettres :

- **si les deux lettres sont sur des lignes et des colonnes différentes**, alors chaque lettre est remplacée par la lettre située sur la même ligne qu'elle mais sur la colonne de l'autre ligne.
- **si les deux lettres sont sur la même ligne**, alors on remplace chaque lettre par la lettre immédiatement à sa droite (éventuellement, on revient à la première lettre de la ligne)
- **si les deux lettres sont sur la même colonne**, alors on remplace chaque lettre par la lettre immédiatement sous elle (éventuellement, on revient à la première lettre de la colonne).
- **si les deux lettres sont identiques** on insère entre les deux lettres une lettre rare (par exemple Q). On le fait en Java en utilisant un objet **StringBuffer** et la méthode `insert(int offset , char c)` qui permet d'insérer un caractère à un certain indice de la chaîne.

On réinsère ensuite les caractères spéciaux comme les espaces, la ponctuation etc en utilisant également un **String Buffer**.

### 2.5.1 Exemple

- On chiffre la chaîne "**Dream Theater**" avec la clé "**Metropolis**". On transforme cette chaîne en mettant toutes les lettres en majuscules et en retirant tous les caractères ne correspondant pas à des chiffres ou des lettres (DREAMTHEATER). On obtient ce **chiffre de playfair** avec la clé :

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

En faisant la substitution de chaque couple de lettres avec ce tableau et en respectant les 4 règles vu précédemment, **DR** devient **MH**, **EA** devient **IR** etc ... On obtient alors la chaîne **MHIRE RRFRSTO** en rajoutant l'espace présent dans la chaîne d'origine. (voir figure 9)

- On reprend ce message en utilisant la même clé et on ré-obtient le message "**DREAM THEATER**". (voir figure 9)

## 2.6 Chiffage de Hill

Le chiffement de Hill consiste en une substitution par groupe de lettres (ici nous utiliserons pour des groupes de deux lettres). Chaque groupe de lettres  $X_i$  du message  $X$ , sera codé par un vecteur  $\begin{pmatrix} x1 \\ x2 \end{pmatrix}$  par rapport au rang de chaque lettre dans la table ASCII étendue. Pour obtenir le bloc  $Y_i$  du message chiffré on va multiplier chaque vecteur  $X_i$  par une matrice  $M$  qui est la

clé de forme :  $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ . Celle ci est correcte uniquement si son determinant est premier avec 256. (On vérifie dans le code si le pgcd du determinant et de 256 est égal à 1 avec la fonction `int pgcd(int m, int n)` de la classe `Crypto`). On a donc :

$$Y_i = \begin{pmatrix} y1 \\ y2 \end{pmatrix} = \begin{pmatrix} x1 \\ x2 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Notons que chaque chiffre du bloc est ramené modulo 256. On obtient alors chaque caractère de la table ASCII étendue correspondant. On représente la matrice  $M$  et les blocs  $X_i$  et  $Y_i$  par un `int[] []` et des `int[]` en Java.

Pour déchiffrer le message on applique le même principe mais en utilisant cette fois la matrice inverse de  $M$  :  $M^{-1}$ . Pour ce faire on multiplie l'inverse du déterminant de  $M$  dans l'arithmétique modulo 256 p pour une certaine matrice  $M'$  :

$$M^{-1} = [det(M)]^{-1} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Pour calculer l'inverse du déterminant on utilise la fonction interne `int modInverse(int a , int m)`.

### 2.6.1 Exemple

- On chiffre la chaîne "**Pantera**" avec la clé  $\begin{pmatrix} 1 & 9 \\ 8 & 1 \end{pmatrix}$ . On rajoute un caractère "\*" pour que la longueur de la chaîne soit paire. On obtient 4 groupes de deux lettres avec leur code de la table ASCII étendue associé :

$$X_1 = \begin{pmatrix} 80 \\ 97 \end{pmatrix} X_2 = \begin{pmatrix} 110 \\ 116 \end{pmatrix} X_3 = \begin{pmatrix} 101 \\ 114 \end{pmatrix} X_4 = \begin{pmatrix} 97 \\ 42 \end{pmatrix}$$

On obtient alors les blocs  $Y_i$  :

$$Y_1 = \begin{pmatrix} 185 \\ 225 \end{pmatrix} Y_2 = \begin{pmatrix} 130 \\ 228 \end{pmatrix} Y_3 = \begin{pmatrix} 103 \\ 154 \end{pmatrix} Y_4 = \begin{pmatrix} 219 \\ 50 \end{pmatrix}$$

On obtient alors le message (**voir figure 10**) quand on regarde les caractères correspondants aux codes dans la table ASCII étendue. (**voir figure 10**)

- Si on déchiffre cette chaîne on ré-obtient le message "**Pantera\***" avec le caractère "\*" qui avait été ajouté. (**voir figure 10**)

## 2.7 Transposition Rectangulaire

La transposition rectangulaire est un algorithme de transposition. C'est à dire que l'on va changer l'ordre des caractères de la chaîne. Le but est d'utiliser un tableau qui contiendra tous les caractères du message.

(Dans notre cas pour préserver les caractères non affichables on utilise un tableau d'entier `int[] []` en Java).

Pour commencer soit  $n$  la longueur de la clé chaque ligne du tableau aura  $n$  éléments. Ensuite on crée une ligne  $R$  qui contiendra le "rang" de chaque caractère du tableau. Celui ayant son code le plus petit dans la table ASCII étendue aura comme "rang" **1**, puis le deuxième plus petit **2** etc ... (Dans notre code en Java elle n'est pas directement incluse dans le tableau vu précédemment mais stockée dans un `int[]` à part).

- **Si on chiffre** on va remplir le tableau ligne par ligne avec le message d'origine. Pour obtenir le message chiffré on va le construire en lisant la colonne correspondant à l'indice de l'élément de  $R$  numéroté 1, puis la colonne avec l'indice numéroté 2, etc ...
- **Si on déchiffre** on doit reconstituer le tableau pour savoir quelle taille font les différentes colonnes. Soit  $n$  la longueur de la clé, et  $m$  la longueur du message.

- Si  $n$  est un multiple de  $m$  alors toutes les colonnes font une taille de  $m/n$ .
- Sinon pour  $m/n$  on a  $q$  le quotient et  $r$  le reste. On aura alors les  $r$  premières colonnes qui ont une taille de  $q + 1$  et les  $n - r$  autres qui ont une taille de  $q$ .

On remplit alors le tableau avec chaque caractère du message par colonne. D'abord la colonne correspondant à l'indice de l'élément de  $R$  numéroté 1 puis 2 etc ... On lit ensuite le tableau ligne par ligne pour obtenir le message déchiffré.

La création de la ligne  $R$  se fait avec la fonction interne `int[] cleToNumeroAssocie(String cle)`.

### 2.7.1 Exemple

(Les cases vides seront notées avec le caractère  $\emptyset$ ).

- On chiffre la chaîne  $M$  "**Ich tu der weh**" avec la clé  $N$  "**Rammstein**". On regardant chaque code ASCII étendue des caractères de la clé. On obtient alors ce tableau  $R$  :

R	a	m	m	s	t	e	i	n
1	2	5	6	8	9	3	4	7

On obtient alors le tableau pour chiffrer en remplissant avec le message  $m$  :

R	a	m	m	s	t	e	i	n
1	2	5	6	8	9	3	4	7
I	c	h		t	u		d	e
r		w	e	h	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

En lisant la colonne 1 puis 2 etc ... On obtient la chaîne chiffrée  $C$  "**Irc dhw eethu**" (il y a deux caractères " " entre le c et le d). (**Voir figure 11**)

- **On déchiffre** la chaîne  $C$  "**Irc dhw eethu**". On garde le même tableau  $R$ .

On a la longueur du mot  $C$  :  $m = 14$  et la longueur de la clé  $N$  :  $n = 9$ .

Pour  $m/n$  on a alors  $q = 1$  et  $r = 5$ . Les 5 premières colonnes ont une taille de 2 ( $q + 1$ ) et les 4 autres une taille de 1 ( $q$ ). On remplit la colonne numérotée 1 avec "Ir" puis celle numérotée 2 avec "c " etc ... On ré-obtient alors le tableau de chiffage initial :

R	a	m	m	s	t	e	i	n
1	2	5	6	8	9	3	4	7
I	c	h		t	u		d	e
r		w	e	h	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

On lit ensuite le tableau ligne par ligne pour ré-obtenir le message  $M$  : "**Ich tu der weh**". (**Voir figure 11**)

## 2.8 DES

Le **DES** (Data Encryption Standard) est un algorithme de sur-chiffrement, c'est à dire que l'on va chiffrer ou déchiffrer avec des clés différentes à chaque fois. Il prend en entrée un bloc de 64 bits et une clé de 64 bits. Si le message fait moins de 64 bits (8 caractères sur 8 bits) on le complète avec des zero. Si il fait plus on sépare le message en blocs de 64bits et on appliquera l'algorithme pour chacun de ces blocs.

C'est la fonction `String[] messageToBlocs(int[] message)` qui transforme le tableau de code ASCII étendue de notre message en blocs de 64 bits. (Nos chaînes de bits sont représentées via des `String` en Java).

L'algorithme se déroule alors comme suit :

- **Diversification des clés** : À partir de la clé donnée en entrée on dérive 16 sous-clés de 48 bits. C'est la fonction `String[] diversificationCle(String cle)` qui crée un tableau



contenant les 16 clés  $K_i$ .

- **Permutation initiale :** Pour chaque blocs de 64 bits on applique une permutation initiale (on va changer chaque bits de place). C'est la fonction `String permutationInitiale(String bloc)` qui s'applique. On obtient alors un bloc de 64 bits que l'on va séparer en deux blocs de 32 bits  $G_0$  et  $D_0$ .
- **Schéma de Feistel :** On va calculer avec 16 itérations (de 1 à 16) tous les  $G_i$  et  $D_i$  en suivant ce schéma tel que :
  - $G_{i-1} = D_{i-1}$
  - $D_{i-1} = G_{i-1} \oplus f_{confusion}(D_{i-1}, K_i)$Les blocs  $D_i$  et  $G_i$  étant stockés dans deux tableaux de `String[]`. Pour l'opérateur  $\oplus$  on utilise une fonction interne `String XOR (String blocA,String blocB)` qui renvoie le résultat de  $A \oplus B$ . Enfin la fonction de confusion renvoie un bloc de 32 bits prenant en entrée un bloc de 32 bits et une clé diversifiée  $K_i$  de 48 bits, c'est la fonction `String fonctionConfusion(String bloc32 , String cleDiversifie)` qui calcule ce bloc.
- **Permutation finale :** On applique enfin à  $G_{16}$   $D_{16}$  une permutation finale et on obtient alors le message chiffré. On le fait avec la fonction interne `String permutationFinale(String bloc)`.

Pour déchiffrer on va appliquer le même principe mais en utilisant les clés  $K_i$  dans l'ordre inverse à chaque itération (d'abord  $K_{16}$  puis  $K_{15}$  etc ...).

*(L'algorithme dispose également d'une option pour chiffrer/déchiffrer uniquement avec un nombre hexadécimal en activant une checkbox).*

### 2.8.1 Exemple

**Pour des messages avec des caractères de la table ASCII étendue :**

- **On chiffre** le message "Gojira" avec la clé "`\x0123456789abcdef`". Il fait 6 caractères donc 48bits on le complète alors avec des 0 et on obtient le bloc en binaire :

$M = 0100\ 0111\ 0110\ 1111\ 0110\ 1010\ 0110\ 1001\ 0111\ 0010\ 0110\ 0001\ 0000\ 0000\ 0000\ 0000$

Après passage dans l'algorithme on obtient le bloc chiffré :

$C = 0101\ 0101\ 0000\ 1010\ 0101\ 0100\ 0111\ 0011\ 0100\ 0011\ 0100\ 0000\ 1101\ 1101\ 0100\ 0001$

Ce qui pour chaque octet représente les caractères de la table ascii étendue numéro [85,10,84,115,67,64,221,65] (un caractère étant codé sur 8 bits). (voir figure 12)

- **On déchiffre** la chaîne  $C$  avec la même clé. On ré-obtient alors le bloc  $M$  et le message "Gojira". (voir figure 12)

**Pour des messages en hexadécimal :**

- **On chiffre** le message en hexadécimal "`\x23456789abcdef01`" avec la clé "`\x0123456789abcdef`". On a alors le bloc à chiffrer :

$M = 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000\ 1001\ 1010\ 1011\ 1100\ 1101\ 1110\ 1111\ 0000\ 0001$

Après passage dans l'algorithme on obtient le bloc :

$C = 1010\ 0100\ 0111\ 0110\ 0000\ 0110\ 1010\ 1111\ 0001\ 0011\ 0010\ 1110\ 1110\ 1111\ 1111\ 0111$

Ce qui correspond à "`\xa47606af132eeff7`" en hexadécimal. (voir figure 13)

- Si on déchiffre  $C$  on ré-obtient le bloc  $M$  soit "`\x23456789abcdef01`". (voir figure 13)

























## 2.9 Algorithmme Surprise !

Nous allons présenter ici l'élaboration d'un algorithme surprise. Celui-ci prend en entrée un message représenté par une chaîne de caractères  $M$  dans la table ASCII étendue et une clé  $C$  elle aussi une chaîne de caractères. Il est composé de deux étapes :

- **Transposition du message  $M$**  : On va effectuer une **Transposition Rectangulaire** de  $M$  avec la clé  $C$ .
- **Substitution de  $M$**  : On va substituer chaque caractère de  $M$  par un caractère dans la **table des émojis**.

### 2.9.1 Table des émojis

Chaque emoji est représenté par un code **Unicode**. En effet toute la table des émojis est en fait une extension de la table **Unicode**. Actuellement on est à la version 13 de celle-ci (30/01/2020) et on compte **1809** émojis pour les codes allant de **U+1F600** à **U+1F3F4**.

76	<a href="#">U+1F628</a>																																																																																																																																																																																																														
----	-------------------------	---	---	---	---	---	---	---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	--	--	--	--	--	--	--	--	--	--

On crée pour cela une classe **EmojiTable** :

```
public class EmojiTable {

    Emoji[] emojiTable = EmojiManager.getAll().toArray(new Emoji[EmojiManager.getAll().size()]);
    Emoji[] emojiTable_simplified = new Emoji[848];

    public EmojiTable(){

        //Emoji simplifié de [0,143] [146,627] [638,836] [1048,1073]
        int compteur=0;
        for (int i=0 ; i<emojiTable.length; i++){

            if( (i>=0 && i<=143) || (i>=146 && i<=627) || (i>=641 && i<=836) || (i>=1048 && i<=1073)

                emojiTable_simplified[compteur] = emojiTable[i];
                compteur++;
            }
        }
        public Emoji[] getEmojiTable_simplified(){return this.emojiTable_simplified;}

        // récupère le numéro d'un emoji dans la table en fonction de son code hexHTML
        public int getEmojiNumber(String emoji_html_hexa){

            for (int i=0 ; i<tab.length ; i++){

                if(emoji_html_hexa.equals(this.emojiTable_simplified[i].getHtmlHexadecimal())) return i;

            }
            return -1;
        }
    }
}
```

Nous travaillons sur une **table d'emoji simplifiée**. En effet comme dit précédemment certains émojis disposent de plusieurs codes Unicode (par exemple les émojis disposant de plusieurs couleurs de peau) donc, de plusieurs nombres Hexadécimaux HTML. Cela rend compliqué le "parse" avec la méthode `EmojiUtils.hexHtmlify(String s)`. Nous gardons donc uniquement les émojis composés d'un seul code **Unicode**, ce qui nous donnent quand même 848 émojis affichables.

*(De plus la table des émojis de la librairie emoji-java n'est pas à jour avec la table actuelle ... )*

### 2.9.3 Fonctionnement de l'algorithme

Après avoir effectué une **transposition rectangulaire** sur le message d'origine on va décaler chaque caractère d'un certain nombre. On a  $M$  le message transposé,  $C_i$  chaque code ASCII étendue de la clé,  $R$  le tableau des numéros associé à la clé (le même que pour la transposition rectangulaire voir 2.7) et pour chaque code  $M_i$  on décale tel que :

$$M'_i = (M'_i + (|C| \times R_{i \bmod |C|} \times C_{i \bmod |C|})) \bmod 848$$

Pour chaque caractère  $C_i$  du message chiffré on prend dans la table des émojis (`Emoji[]`) l'unicode de l'emoji numéro  $M'_i$  avec la méthode de l'objet `Emoji` `getUnicode()`. On obtient ainsi un message uniquement composé d'émojis. (Android sachant afficher un emoji en connaissant l'unicode associé)

Pour le **déchiffrage** on effectue l'opération inverse on décale dans l'autre sens modulo 848 puis on fait une transposition rectangulaire en déchiffrant. Le soucis est que l'on a comme message à déchiffrer des émojis que **Java** ne sait pas reconnaître. On utilise alors l'objet **EmojiUtils** et la méthode `hexHtmlify(String s)` qui nous renvoie la représentation de chaque émoji en hexadécimal html (l'objet **EmojiUtils** ne disposant pas d'une méthode pour donner la représentation en Unicode).

Par exemple l'émoji "pomme" a pour code `&#x1f34e`.

#### 2.9.4 Exemple

- **On chiffre** le message "**Machine Head**" avec la clé "**Locust**". On transpose chaque caractère de la chaîne "**MecHa iandhe**" avec les codes ASCII `[77,101,99,72,97,32,105,97,110,100,104,101,]`. Chaque code est décalé et ramené modulo 848 on obtient alors les émojis numéro `[533,403,439,44,313,120,561,39]` de notre table des émojis. (voir figure 14)
- **On déchiffre** les émojis obtenus en récupérant pour chacun leur code html hexadécimal : `&#x1f345; &#x1f4dd; &#x1f3ac; &#x1f61f; &#x1f390; &#x261d; &#x1f5fd; &#x1f4ec; &#x1f47e; &#x1f478; &#x1f38b; &#x1f436;`

On peut alors récupérer chaque numéro d'émoji correspondant, en comparant leur code html hexadécimal avec `Emoji.getHtmlHexadecimal()`.

On peut déchiffrer le message et on ré-obtient la chaîne "**Machine Head**". (voir figure 14)

## 2.10 RSA

L'algorithme **RSA** est un chiffrement asymétrique, c'est à dire que l'on va utiliser une clé publique pour chiffrer et une clé privée pour déchiffrer.

La clé publique est un couple d'entiers naturels  $(e, n)$  et la clé privée un couple d'entiers naturels  $(d, n)$ .

Notre algorithme prend en entrée deux nombres premiers  $P$  et  $Q$ , un message  $M$  qui est un entier et renvoie un nombre  $C$  qui est le message chiffré. Il se déroule comme suit :

- On choisit  $P$  et  $Q$  deux nombres premiers.
- On calcule  $n = P \times Q$  et  $\phi(n) = (P - 1) \times (Q - 1)$ .
- On choisit un entier  $e$  tel que  $e$  est premier avec  $\phi(n)$  (dans notre cas on prend  $e > q$  tel que le pgcd de  $e$  et  $\phi(n)$  soit égal à 1).
- On calcule  $d$  qui est l'inverse de  $e$  dans l'arithmétique modulo  $\phi(n)$  (on utilise la fonction interne `modInverse(int a ,int m)` utilisée dans le chiffage de **Hill** voir 2.6).
- On a maintenant la clé publique  $(e, n)$  et la clé privée  $(d, n)$ .

Pour obtenir le message chiffré  $C$  on a :  $C \equiv M^e \pmod{n}$

Pour déchiffrer et ré-obtenir le message  $M$  on a :  $M \equiv C^d \pmod{n}$

Si le message  $M$  ou  $C$  est supérieur ou égal à  $n$  alors, on les divise en blocs inférieurs strictement à  $n$ . On le fait avec la fonction interne `String[] mToStringBlocs(int m, int n)` qui va prendre un nombre  $m$  et  $n$  et qui va diviser le message  $m$  en blocs inférieurs strictement à  $n$ .

Enfin pour le calcul de  $M$  ou  $C$  le soucis est que  $M^e$  ou  $C^d$  sont des très grand nombres. Java ne peut donc pas les stocker efficacement dans un type `int` et effectuer le modulo. On utilise alors l'objet **BigInteger** qui déjà stocke ces très grands nombres. Et notamment il dispose d'une méthode `BigInteger modPow(BigInteger exponent , BigInteger modulus)` qui nous permettra de calculer  $M$  et  $C$ .

```
//chiffage ou déchiffage des blocs
String[] blocsM = mToStringBloc(M , n);
```

```
//utilisation de BigInteger

BigInteger e_big = BigInteger.valueOf((long) e);
BigInteger d_big = BigInteger.valueOf((long) d);
BigInteger n_big = BigInteger.valueOf((long) n);

for (int i=0 ; i<blocsM.length ; i++){

BigInteger bloc_chiffre_dechiffre = BigInteger.valueOf(0);
    BigInteger bloc = BigInteger.valueOf(Long.parseLong(blocsM[i]));

    if(chiffre) bloc_chiffre_dechiffre = bloc.modPow(e_big,n_big);

    if(!chiffre) bloc_chiffre_dechiffre = bloc.modPow(d_big , n_big);

resultat+=bloc_chiffre_dechiffre.toString();

}

return resultat;
```

### 2.10.1 Exemple

- On prend  $P = 83$  et  $Q = 89$  :
  - On a  $n = 7387$  et  $\phi(n) = 7216$
  - On choisit  $e = 91$  et on calcule  $d \equiv 91^{-1} \pmod{7987}$
  - On a la clé publique  $(91, 7387)$  et la clé privée  $(5075, 7387)$
- **On chiffre** le message  $M = 6882326$ .  $M \geq n$  donc on divise  $M$  en deux blocs  $M_1 = 6882$  et  $M_2 = 326$ .  
 On alors  $C_1 \equiv 6882^{91} \pmod{7387} = 7301$  et  $C_2 \equiv 326^{91} \pmod{7387} = 1658$   
 Le message chiffré  $C$  est **73011658 (voir figure 15)**
- **On déchiffre** le message  $C = 73011658$ .  $C \geq n$  donc on divise  $C$  en deux blocs  $C_1 = 7301$  et  $C_2 = 1658$ .  
 On alors  $M_1 \equiv 7301^{5075} \pmod{7387} = 6882$  et  $M_2 \equiv 1658^{5075} \pmod{7387} = 326$   
 Le message déchiffré  $M$  est **6882326 (voir figure 15)**

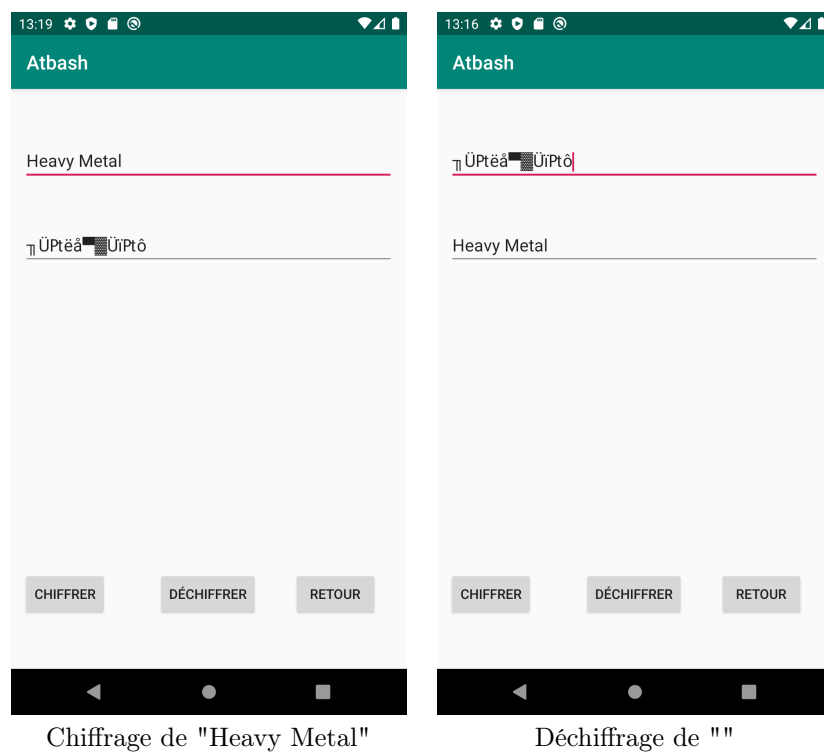
## 3 Conclusion

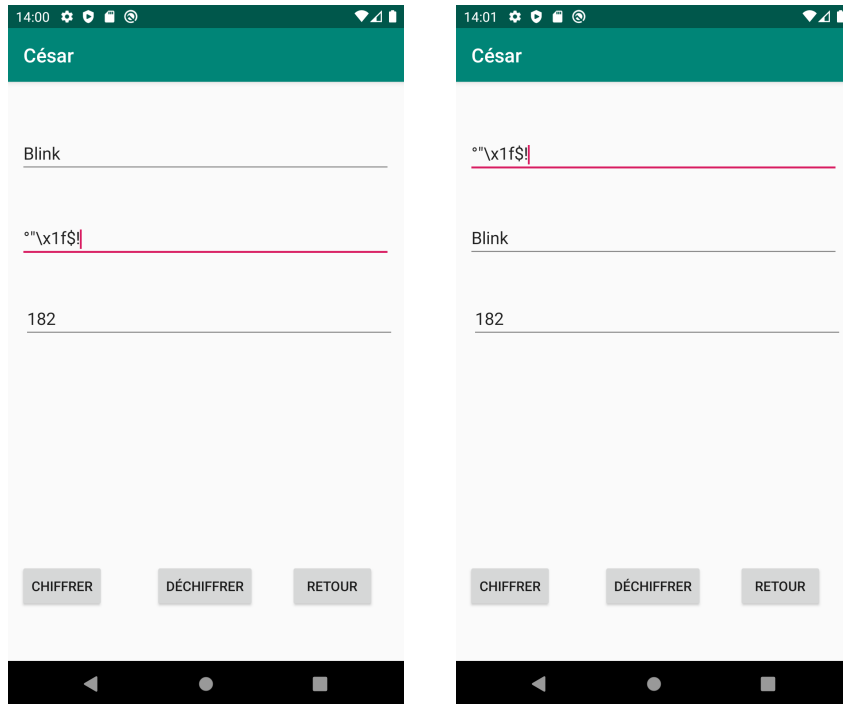
Nous avons présenté plusieurs algorithmes de chiffrement durant ce rapport. Chacun d'entre eux repose sur des techniques particulières de chiffrement, mais la plupart sont très peu résistant à des attaques de type "brute force" ou à des attaques statistiques même en utilisant plus de caractères (table ASCII étendue). Le **DES** quand à lui a été remplacé car il était sensible à des attaques sur les clés. Son vrai successeur est l'**AES** (Advanced Encryption Standard) car celui-ci n'utilisant pas de schéma de Feistel a une complexité plus faible que celle du DES. Il utilise moins de mémoire et est beaucoup plus résistant aux attaques (utilisant des longueurs de clés de 128, 192, 256 bits).

L'**algorithme RSA** est quand à lui très résistant aux attaques à condition d'utiliser des nombres premiers très grands pour la génération des clés(c'est pour cela qu'il est très utilisé dans l'échange de données confidentielles). En effet "casser" RSA nécessite d'utiliser une factorisation en produits de facteurs premiers, qui est un algorithme ayant une complexité **exponentielle**. Cependant l'algorithme de **Shor** un algorithme quantique est capable de le faire pour un entier

$N$  en un temps  $O((\log N)^3)$  ce qui pourrait complètement mettre à mal la sécurité de **RSA**, bien que l'on soit encore très loin d'avoir des ordinateurs quantiques le permettant.

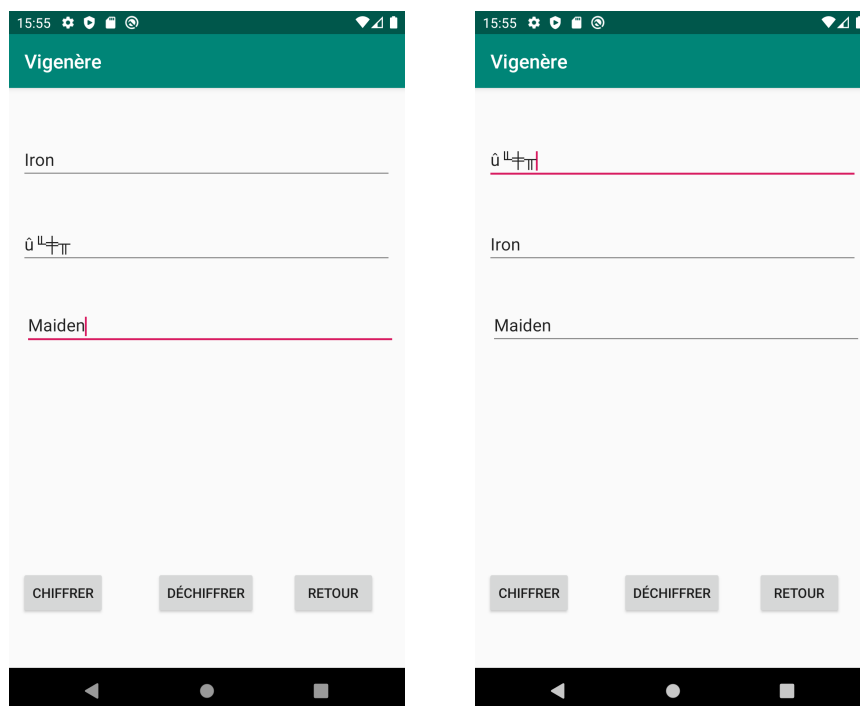
## 4 Annexe

FIGURE 6 – Chiffre et déchiffre avec **Atbash**



Chiffage de "Blink" avec  $d=182$     Déchiffage de "\x1f \$" avec  $d=182$

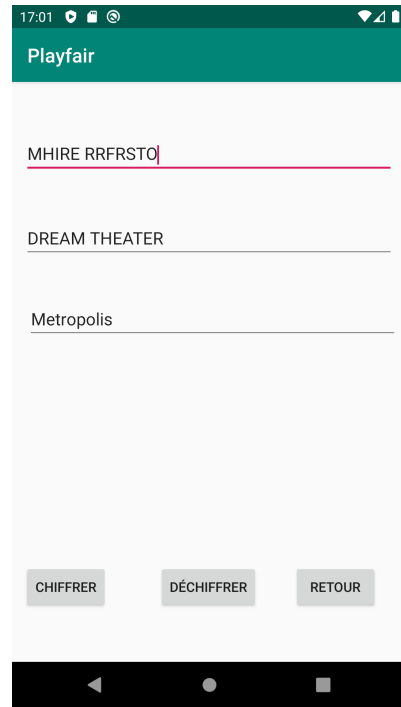
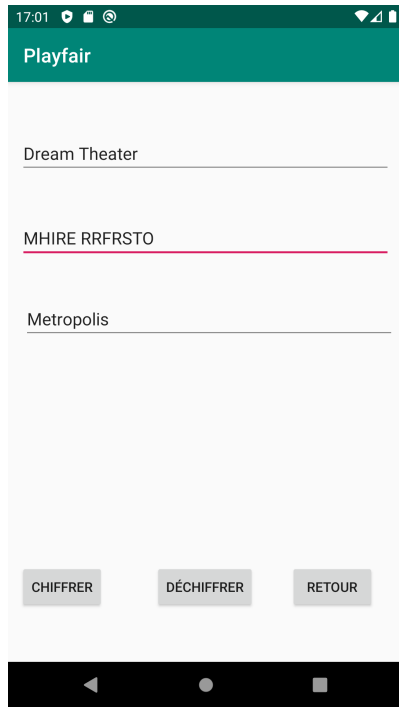
FIGURE 7 – Chiffage et déchiffage avec **César**



Chiffage de "Iron" avec la clé "Maiden"

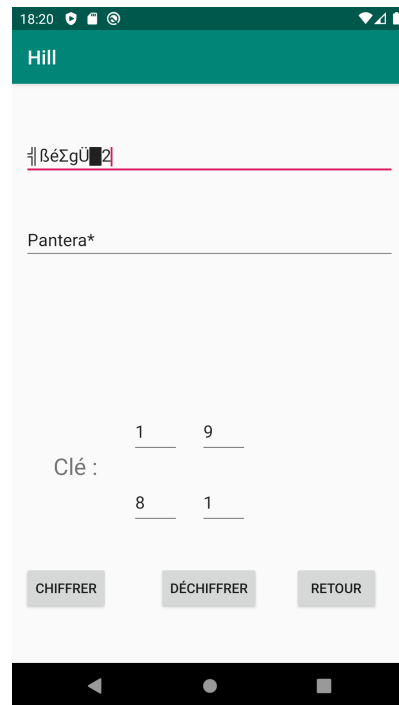
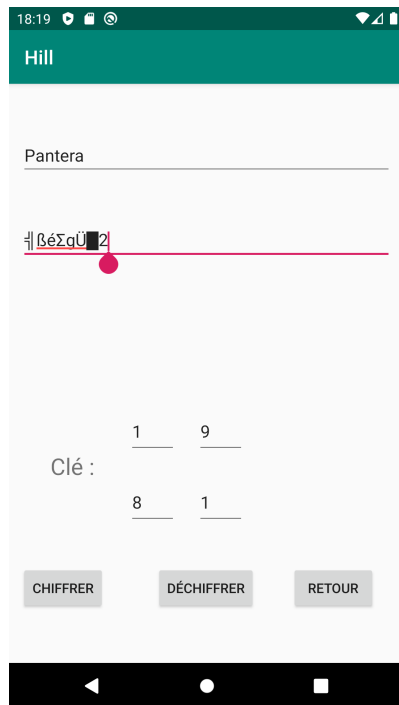
Déchiffage

FIGURE 8 – Chiffage et déchiffage avec **Vigenère**



Chiffage de "Dream Theater" avec la clé "Metropolis"    Déchiffage de "MHIRE RRFRSTO"

FIGURE 9 – Chiffage et déchiffage avec **Playfair**

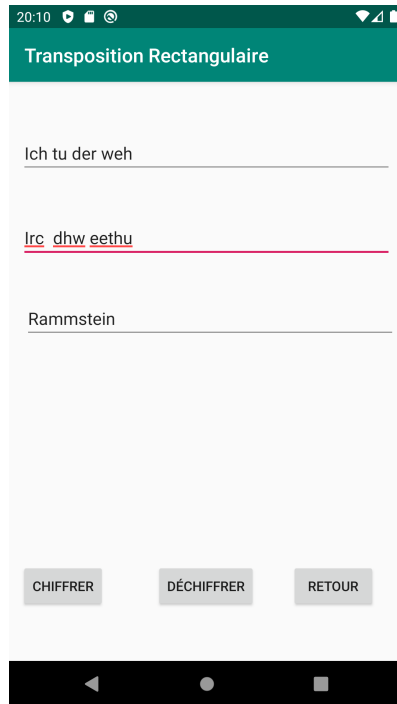


Chiffage de "Pantera" avec la clé  $\begin{pmatrix} 1 & 9 \\ 8 & 1 \end{pmatrix}$

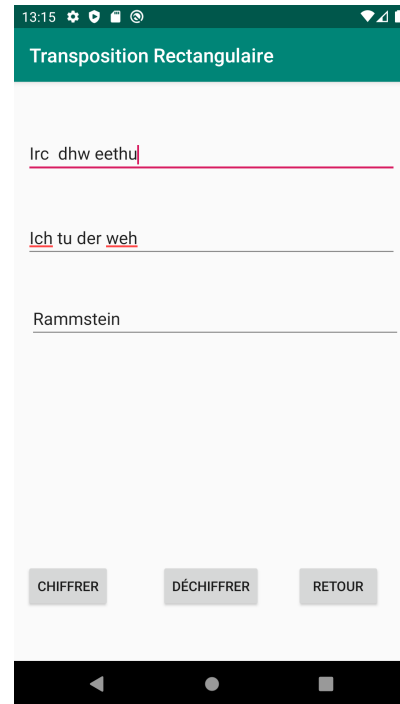
Déchiffage

FIGURE 10 – Chiffage et déchiffage avec **Hill**



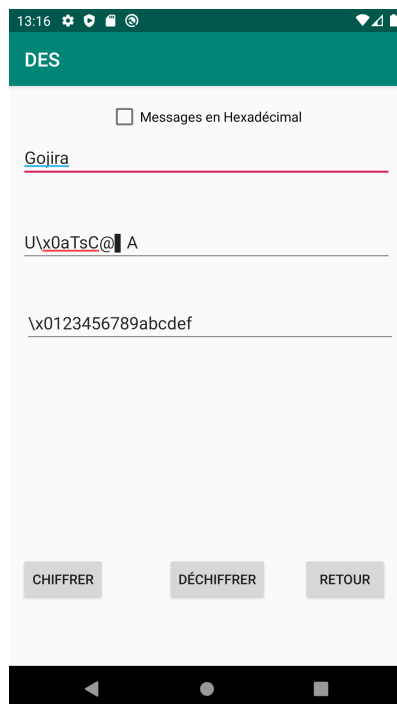


Chiffage de "Ich tu der weh" avec la clé "Rammstein"



Déchiffage

FIGURE 11 – Chiffage et déchiffage avec une **Transposition Rectangulaire**

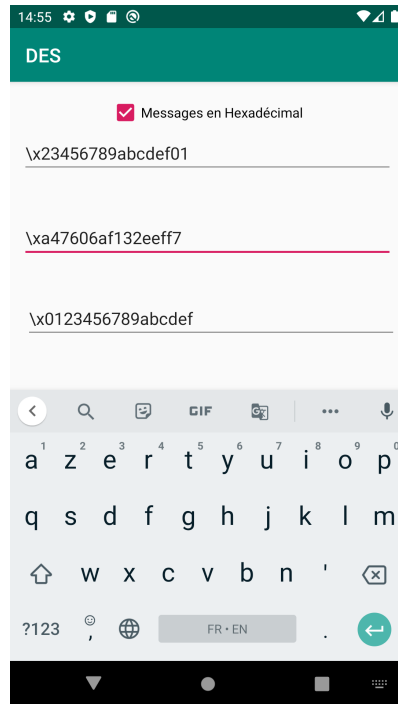


Chiffage de "Gojira" avec la clé "\x0123456789abcdef"

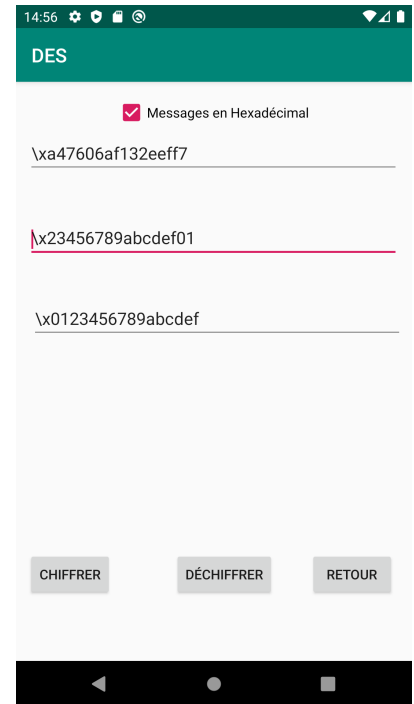


Déchiffage

FIGURE 12 – Chiffage et déchiffage avec le **DES**

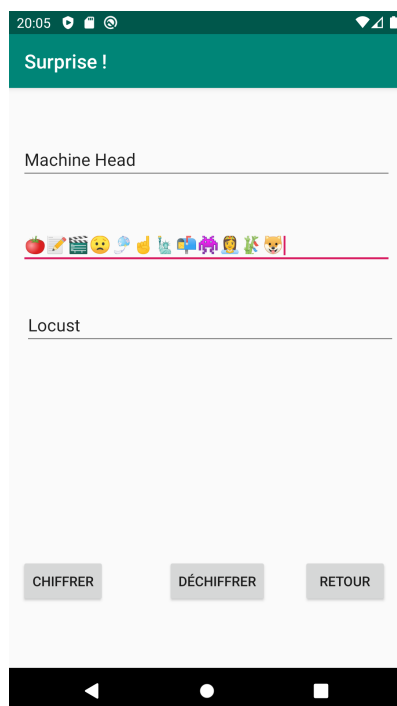


Chiffage de "\x23456789abcdef01" avec la clé "\x0123456789abcdef"

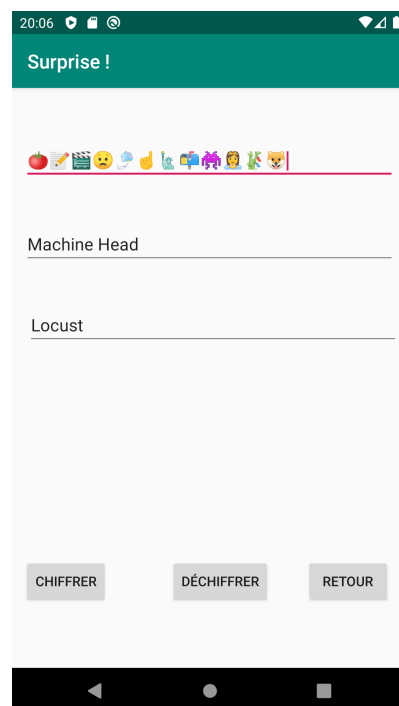


Déchiffage

FIGURE 13 – Chiffage et déchiffage en hexadécimal avec le **DES**

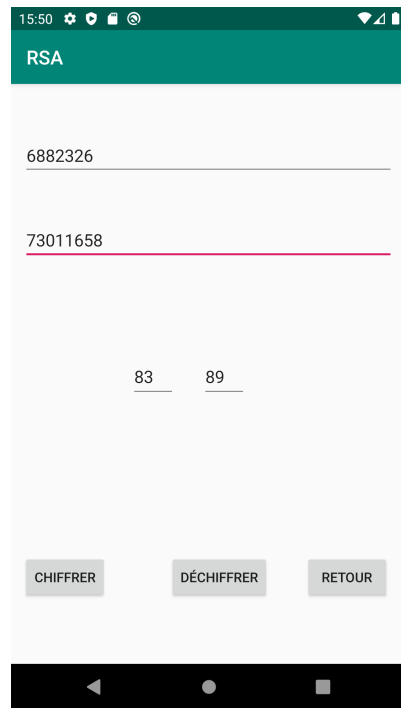


Chiffage de "Machine Head" avec la clé "Locust"

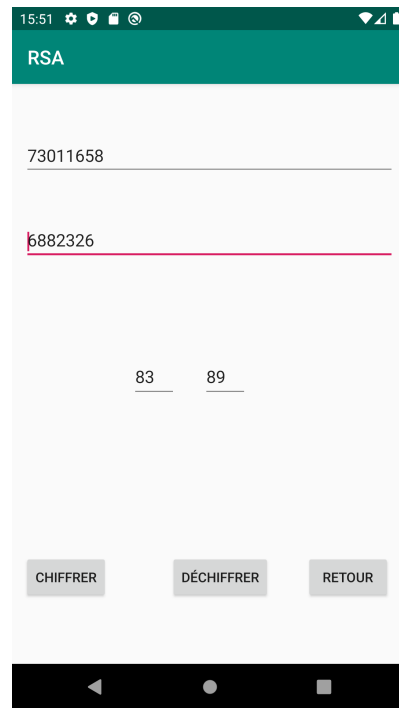


Déchiffage

FIGURE 14 – Chiffage et déchiffage avec un **algorithme surprise**



Chiffage de **6882326** pour  $P = 83$  et  $Q = 89$



Déchiffage

FIGURE 15 – Chiffage et déchiffage avec un **algorithme surprise**

## Références

- [1] Bibmath. La saga du des. <http://www.bibmath.net/crypto/index.php?action=affiche&quoi=moderne/des>.
- [2] Bibmath. Le chiffre atbash. <http://www.bibmath.net/crypto/index.php?action=affiche&quoi=ancienne/atbash>.
- [3] Bibmath. Le chiffre de playfair. <http://www.bibmath.net/crypto/index.php?action=affiche&quoi=ancienne/playfair>.
- [4] GeeksForGeeks. BigInteger class in java. <https://www.geeksforgeeks.org/biginteger-class-in-java>.
- [5] github kcthota. emoji4j. <https://github.com/kcthota/emoji4j>.
- [6] github vdurmont. emoji-java. <https://github.com/vdurmont/emoji-java>.
- [7] javatpoint. Java stringBuffer insert()method. <https://www.javatpoint.com/java-stringbuffer-insert-method>.
- [8] Stack Overflow. Converting an int to a binary string representation in java? <https://stackoverflow.com/questions/2406432/converting-an-int-to-a-binary-string-representation-in-java>.
- [9] Stack Overflow. Is there anyway to convert emoji to text in java? <https://stackoverflow.com/questions/34802721/is-there-anyway-to-convert-emoji-to-text-in-java>.
- [10] Stack Overflow. Modulus with big numbers in java. <https://stackoverflow.com/questions/54165293/modulus-with-big-numbers-in-java>.
- [11] SUPINFO. Chiffre affine et chiffre de hil. <https://www.supinfo.com/cours/1ARI/chapitres/05-chiffre-affine-chiffre-hill>.
- [12] UNICODE. Full emoji list, v13.0. <https://unicode.org/emoji/charts/full-emoji-list.html>.
- [13] Wikipedia. Algorithme de shor. [https://fr.wikipedia.org/wiki/Algorithme\\_de\\_Shor](https://fr.wikipedia.org/wiki/Algorithme_de_Shor).
- [14] Wikipedia. Algorithme d'euclide étendu. [https://fr.wikipedia.org/wiki/Algorithme\\_d'Euclide\\_étendu](https://fr.wikipedia.org/wiki/Algorithme_d'Euclide_étendu).
- [15] Wikipedia. Chiffre de hill. [https://fr.wikipedia.org/wiki/Chiffre\\_de\\_Hill](https://fr.wikipedia.org/wiki/Chiffre_de_Hill).
- [16] Wikipedia. Chiffrement rsa. [https://fr.wikipedia.org/wiki/Chiffrement\\_RSA](https://fr.wikipedia.org/wiki/Chiffrement_RSA).
- [17] Wikipedia. Data encryption standard. [https://fr.wikipedia.org/wiki/Data\\_Encryption\\_Standard](https://fr.wikipedia.org/wiki/Data_Encryption_Standard).
- [18] Wikipedia. How to print the extended ascii code in java from integer value. <https://stackoverflow.com/questions/22273046/how-to-print-the-extended-ascii-code-in-java-from-integer-value>.
- [19] Wikipedia. Page de code 437. [https://fr.wikipedia.org/wiki/Page\\_de\\_code\\_437](https://fr.wikipedia.org/wiki/Page_de_code_437).
- [20] Wikipedia. Triple des. [https://fr.wikipedia.org/wiki/Triple\\_DES](https://fr.wikipedia.org/wiki/Triple_DES).