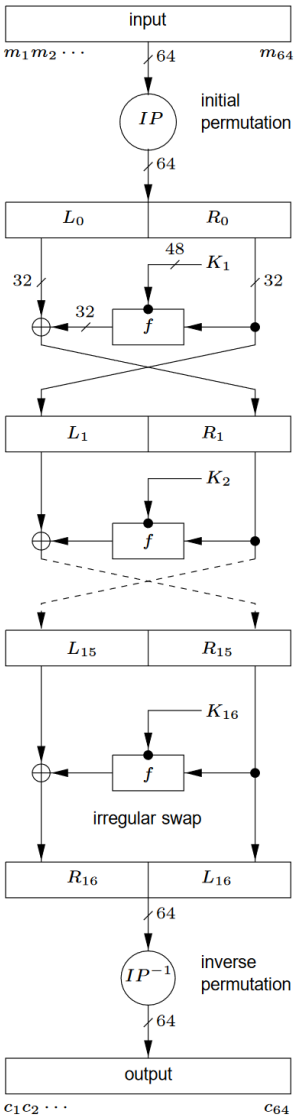
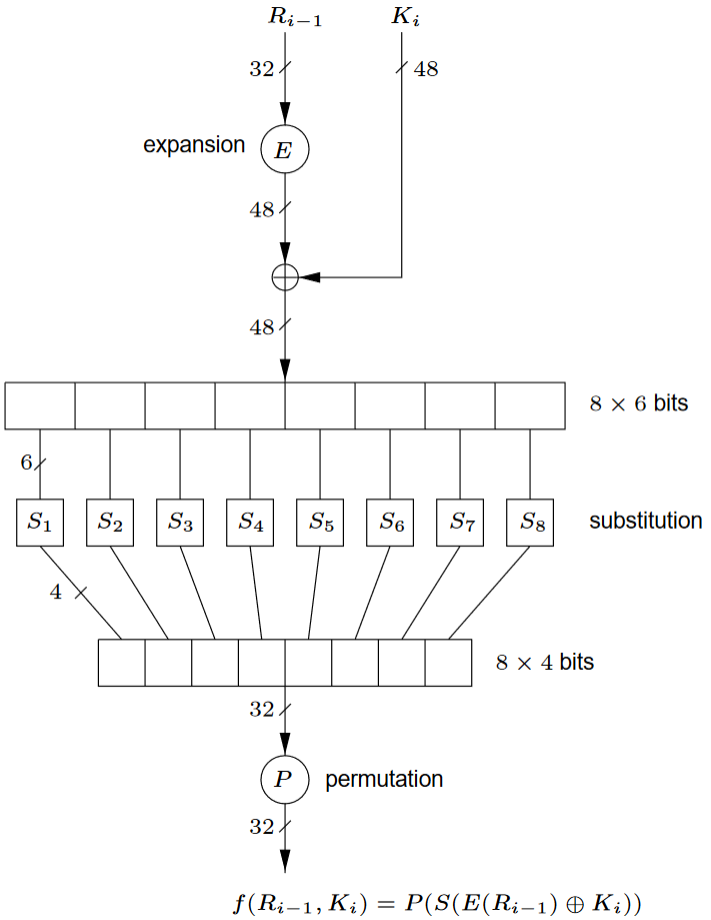
Bilel Bouallouche
22200593

Calcul sécurisé
Projet: Attaque par fautes sur le DES

Principe général

Une attaque par faute sur le DES consiste à introduire volontairement une erreur dans le système de chiffrement du DES afin d'en tirer des informations sur la clé de chiffrement.
On rappelle d'abord comment fonctionne le DES et sa fonction f associée:





On suppose l'attaquant capable d'injecter une faute sur la valeur de sortie de R_{15} de telle sorte à obtenir 32 chiffrés fautés, un chiffré juste C, et un clair M associé aux différents chiffrés. On note alors R_{15}^* la valeur fautée obtenue. Nous noterons dans la suite par un * toutes les valeurs fautées.

Retrouver 48 bits de la clé

Notre objectif dans cette partie va être de retrouver K_{16}, une portion de 48 bits de la clé.
Le schéma de Feistel nous donne les relations suivantes :

$$\begin{cases} L_{16} = f_{K_{16}}(R_{15}) \oplus L_{15} \\ R_{16} = R_{15} \end{cases} \qquad \begin{cases} L_{16}^* = f_{K_{16}}(R_{15}^*) \oplus L_{15} \\ R_{16}^* = R_{15}^* \end{cases} \tag{1}$$

On en déduit la relation :

$$L_{16} \oplus L_{16}^* = f_{K_{16}}(R_{15}) \oplus f_{K_{16}}(R_{15}^*)$$

Or la fonction f du schéma de Feistel pour le DES est connue : $f_{K_{16}}(R_{15}) = P(S(E(R_{15}) \oplus K_{16}))$ où :

- P est une permutation donnée dans la documentation du DES.
- S est la fonction qui modélise le comportement des s-boxs. Plus précisément, soit x de taille 8×6 bits, on a $S(x) = S_1(x_{[1:6]})||S_2(x_{[7:12]})|| \dots ||S_8(x_{[42:48]})$ avec $S_1, \dots, S_8$ les boites de substitutions (s-boxs) données dans la documentation du DES.
- E est une fonction d'expansion permettant de passer de 32 à 48 bits. Elle repose sur une permutation et le dédoublement de certains bits, son fonctionnement est donnée dans la documentation du DES.

Ainsi :

$$L_{16} \oplus L_{16}^* = P(S(E(R_{15}) \oplus K_{16})) \oplus P(S(E(R_{15}^*) \oplus K_{16}))$$

.

Par linéarité de la permutation :

$$L_{16} \oplus L_{16}^* = P\big(S(E(R_{15}) \oplus K_{16}) \oplus S(E(R_{15}^*) \oplus K_{16})\big)$$

Or, P est connue donc facilement inversible, on peut donc considérer l'équation:

$$P^{-1}(L_{16} \oplus L_{16}^*) = S(E(R_{15}) \oplus K_{16}) \oplus S(E(R_{15}^*) \oplus K_{16})$$

Puisqu'on sait quelles portions de bits $E(R_{15}^*)$ et de $E(R_{15})$ vont dans quelle s-box, on en déduit le système d'équation d'inconnue K_{16}:

$$\begin{aligned} P^{-1}(L_{16} \oplus L_{16}^*)_{[1:4]} &= S_1(E(R_{15})_{[1:6]} \oplus K_{16_{[1:6]}}) \oplus S(E(R_{15}^*)_{[1:6]} \oplus K_{16_{[1:6]}}) \\ &\vdots \\ P^{-1}(L_{16} \oplus L_{16}^*)_{[28:32]} &= S_8(E(R_{15})_{[42:48]} \oplus K_{16_{[42:48]}}) \oplus S(E(R_{15}^*)_{[42:48]} \oplus K_{16_{[42:48]}}) \end{aligned}$$

Tout d'abord, il est très simple de retrouver (L_{16}, R_{16}) et (L_{16}^*, R_{16}^*) : il suffit d'annuler la permutation IP^{-1} sur les chiffrés, en appliquant IP.

On propose de faire une attaque par recherche exhaustive sur chaque portion de 6 bits de K_{16} pour trouver un ensemble de solutions possibles pour chaque ligne du système. Il s'agira ensuite de reconstruire K_{16} en réitérant cette attaque pour chaque chiffré fauté et en gardant la portion résolvant sa ligne dans le système.

On obtient ainsi K_{16}, qui fait 48 bits. Nous verrons plus bas comment déduire de K_{16} la clé complète K (de taille 56 bits).

Application concrète

Retrouver K_{16}

On sait que K_{16} est la solution au système:

$$P^{-1}(L_{16} \oplus L^*_{16})_{[1:4]} = S(E(R_{15})_{[1:6]} \oplus K_{16_{[1:6]}}) \oplus S(E(R^*_{15})_{[1:6]} \oplus K_{16_{[1:6]}})$$
$$\vdots$$
$$P^{-1}(L_{16} \oplus L^*_{16})_{[28:32]} = S(E(R_{15})_{[42:48]} \oplus K_{16_{[42:48]}}) \oplus S(E(R^*_{15})_{[42:48]} \oplus K_{16_{[42:48]}})$$

Nous proposons donc d'attaquer chaque s-box.
Pour cela, nous allons établir une correspondance entre chaque chiffré fauté et la s-box qui a eu en entrée le bit fauté initialement. On sait que l'erreur sur R_{15} consiste en un changement de bit mais cette dernière peut se propager lors de l'expansion.

Nous allons donc regarder pour chaque chiffré faux les valeurs de $\Delta = R_{16} \oplus R^*_{16}$ et $E(\Delta)$.
Ce dernier représente un nombre de 48 bits, découpable en 8 blocs de 6 bits chacuns, qui interviendront dans la s-box. Le bit fauté sera alors dans un de ces 8 blocs (les autres blocs seront alors simplement 6 bits nuls) et permettra de déduire dans quelle s-box la faute entrera selon son indice dans la chaine de bits représentant $E(\Delta)$.

Algorithme 1 : Correspondance

Entrées : Liste de chiffres fautés Lfautes
Output : dictionnaire correspondance de la forme {numero de la s-box: liste L de fautés tel que l'erreur dans R^*_{15} } soit passé dans la s-box

```
correspondance = dictionnaire vide
R16J = IP(C)[32:]
pour faux dans Lfautes faire
    R16F = IP(faux)[32:]
    Δ = R16F ⊕ R16J
    exp = E(Δ)
    i = 0
    pour j variant de 1 à 48 avec un pas de 6 faire
        si exp[j:j+6] != '000000' alors
            correspondance[i+1].append(faux)
        i += 1
retourner correspondance
```

L'attaque par recherche exhaustive est alors simple:

1. On parcourt le dictionnaire de correspondance avec le tuple (s-box, liste de faux associé)
2. pour chaque faux de la liste, on retrouve L_{16}^* et R_{16}^*.
3. on reconstruit le système d'équation en veillant a conserver le slicing avec Python.
4. On teste tous les nombres binaires possibles composés de 6 bits
5. Lorsqu'on a égalité entre le membre de gauche et de droite, on considère le nombre binaire comme portion possible de clé que l'on associe a la s-box qu'on était en train d'attaquer.
6. On conserve ensuite les portions en commun pour chaque chiffré faux sur chaque s-box (on regarde l'intersection des solutions)
7. On concatène ensuite les portions de clé pour reconstituer K_{16}

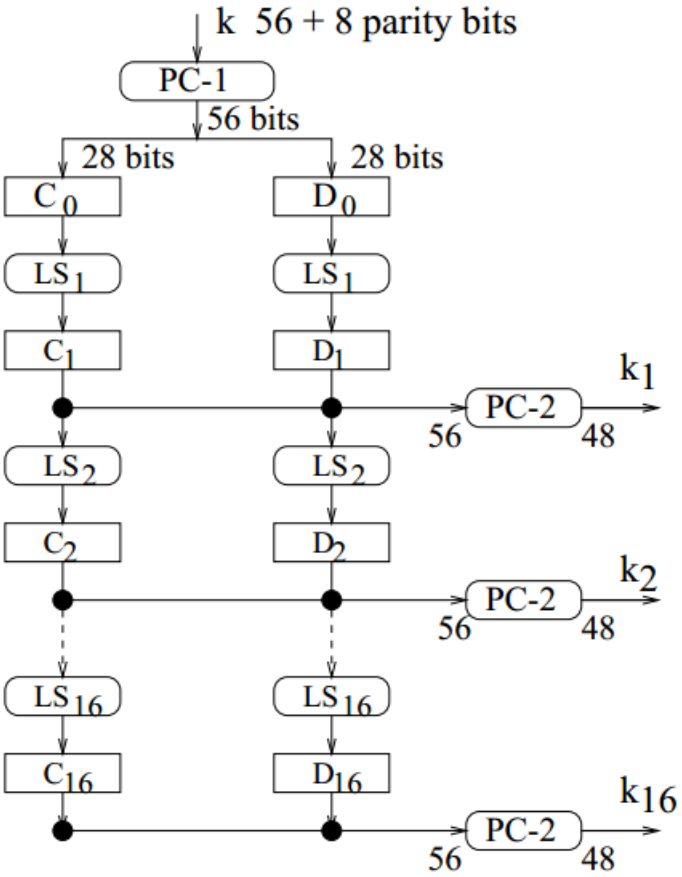
Pour le détail des implémentations, le code Python est donné en annexe.
A cette étape, on trouve

$$K_{16} = 7044C7DAABC3$$

.

Retrouver la clé complète

On propose pour déduire de K_{16} la valeur de Kd'attaquer l'algorithme de cadencement de la clé.



L'algorithme repose sur deux permutations connues, PC_1 et PC_2, ainsi que des opérations de décalages de bits.
Il est clair que $C_{16}||D_{16} = PC_2^-1(K_{16})$.
Or, $C_0 = C_{16}$ et $D_{16} = D_0$.
En effet, lors du cadencement des clés, nous allons faire plusieurs opérations de décalage des bits successifs C_i et D_i. La documentation du DES nous permet d'affirmer que les décalages vont nous donner $(C_16, D_16) = (C_0, D_0)$. En effet, si l'on compte le nombre de décalages appliqués, nous obtenons bien 28 décalages de 1 bit successifs ce qui correspond à la taille de C_0 et D_0.
Il vient donc que $K = PC_1^{-1}(PC_2^{-1}(K_{16}))$.

Néanmoins, l'inversion de PC_2 pose problème puisque $PC_2 : \{0,1\}^{56} \to \{0,1\}^{48}$. Ainsi son inverse est $PC_2^{-1} : \{0,1\}^{48} \to \{0,1\}^{56}$. Ainsi, 8 bits de K ne sont pas retrouvable simplement par cette méthode. Pour trouver ces 8 bits, nous allons d'abord devoir déterminer où devraient-ils se trouver puis faire une recherche exhaustive.
Les 8 bits sont les bons lorsque $\mathrm{DES}(M, K) = C$.

Nous posons $K_{tmp} = PC_1^{-1}(PC_2^{-1}(K_{16}))$. Un parcours des permutations permets de retrouver les bits manquant et de marquer leur position (on propose arbitrairement de mettre un * à la place). Dans notre cas, on a

$$K_{tmp} = 1011110000001 * *010 * *00100101011000100010011011001 0 * 10 * 100 * 1 * 0110$$

.

Il suffit donc de faire une simple recherche exhaustive sur les 8 bits manquants pour retrouver K.

On trouve alors

$$K = BC088256226C9226$$

Parité de la clé

On corrige la clé de telle sorte a rajouter des bits de parités. Il suffit de compter le nombre de 1 et de 0 a chaque octet de façon a ce que chacun ait un nombre impair de 1. On se sert pour cela d'une fonction implémentée en Python, on renvoie alors à l'annexe. On trouve finalement:

$$K = BC088357236D9226$$

Attaque sur les tours précédents

Analyse de la complexité dans le cas d'une attaque sur R_{15}

Il est important de noter que nous allons essayer de donner une complexité asymptotique théorique. L'attaque étant implémenté avec Python et l'usage de chaines de caractères et de tableaux, elle est en pratique beaucoup plus grande. Une solution pourrait etre d'implémenter cette attaque avec une bibliothèque Python qui gererait mieux les nombres binaires ou directement dans un langage bas niveau comme le C. On va donc supposer que les opérations du DES sont négligeables (les permutations, expansions et $\oplus$, sont en pratique implémentées en langage bas niveau ou directement sur le matériel).

Pour trouver K_{16}, on :

- établit une correspondance, ce qui consiste 32 tours d'une boucle for pour passer en revue tous les chiffrés faux et 8 tours d'une boucle for sur chacun d'eux. Pendant ces tours de boucles on ne fait que des opérations d'expansions, des $\oplus$ et des comparaisons de bits. On peut donc supposer sa complexité négligeable devant la recherche exhaustive
- la recherche exhaustive: elle consiste en 4 boucles for imbriquées. On parcourt d'abord chaque s-box, chaque faux de chaque s-box, chaque chaines de caractères de 6 bits (2^6 possibilités) puis une boucle for pour comparer avec le bon slicing (on peut considérer cette dernière comme négligeable). On a donc une complexité en $O(8 \times 32 \times 2^6) = O(2^{14})$

Pour trouver K à partir de K_{16}, on :

- applique des permutations (ce que l'on suppose d'une complexité négligeable)
- On fait une recherche exhaustive en parcourant l'ensemble des chaines de 8 bits (256 possibilités). On a donc une complexité théorique en $O(2^8)$.

Cela nous fait donc une complexité pour trouver K en $O(2^{14}+2^8)$. L'attaque est donc beaucoup plus rapide qu'une simple recherche exhaustive sur le DES.

Analyse de la complexité dans le cas d'une attaque sur R_{14}

Dans le cas d'une injection de faute sur R_{14}, on aura

$$\begin{cases} L_{15} = R_{14} \\ R_{15} = f_{K_{15}}(R_{14}) \oplus L_{14} \\ L_{16} = f_{K_{16}}(R_{15}) \oplus L_{15} \\ R_{16} = R_{15} \end{cases} \qquad \begin{cases} L^*_{15} = R_{14^*} \\ R^*_{15} = f_{K_{15}}(R^*_{14}) \oplus L_{14} \\ L^*_{16} = f_{K_{16}}(R^*_{15}) \oplus L^*_{15} \\ R^*_{16} = R^*_{15} \end{cases}$$

Ou encore :

$$\begin{cases} L_{16} = f_{K_{16}}(f_{K_{15}}(R_{14}) \oplus L_{14}) \oplus L_{15} \\ R_{16} = f_{K_{15}}(R_{14}) \oplus L_{14} \end{cases} \qquad \begin{cases} L^*_{16} = f_{K_{16}}(f_{K_{15}}(R^*_{14}) \oplus L_{14}) \oplus L^*_{15} \\ R^*_{16} = f_{K_{15}}(R^*_{14}) \oplus L_{14} \end{cases}$$

Ainsi, il va falloir faire deux recherches exhaustives sur 48 bits pour trouver K_{15}. Ainsi la complexité pour retrouver K est en $O((2^{14})^2 + 2^8)$.

L'attaque reste possible mais a un temps d'exécution plus long.

Complexité dans le cas d'une attaque sur R_i

On peut étendre le raisonnement fait précédemment de réécriture du système d'équation On aura alors :

tour fauté	Nombre de recherches exhaustives sur 48 bits	Complexité pour trouver K_{16}
15	1	$O(2^{14})$
14	2	$O(2^{28})$
13	3	$O(2^{42})$
12	4	$O(2^{56})$
11	5	$O(2^{70})$

Ainsi, a partir du 12 ème tour, l'efficacité de l'attaque n'est plus pertinente comparé a une simple attaque par recherche exhaustive de la clé.

Contre-mesures

On propose trois contre-mesures différentes.

- La première consiste à utiliser différents circuits électroniques effectuant le DES et de comparer leurs résultats en effectuant un $\oplus$ (ainsi la complexité n'est pas plus élévée).
  L'inconvénient est que cela augmente le coût de conception du matériel ainsi que sa taille et la puissance d'alimentation nécessaire.

- La seconde consiste à essayer de mettre au point des systèmes de détections d'erreurs (en se mettant d'accord sur une convention commune entre les usagers sur les bits de parité par exemple).

- La dernière consiste a empêcher l'attaquant de former le système d'équations que nous avons étudier. On s'est servi du fait que dans le cadre du DES, les portions de clés et des $E(R_{15})$ vont dans des s-boxs bien déterminées. On peut imaginer une dispersion des bits moins triviales dans les s-boxs construites à partir de la valeur de la clé elle-même.
  L'inconvénient est que ce n'est plus un standard du DES et cela rajoutera une étape supplémentaire qui rallongera le temps de chiffrement et de déchiffrement.

Annexe

On s'est servi de Python pour implémenter l'attaque ainsi que les packages requests et BeautifulSoup (l'exécution nécessite donc une connexion internet).

```
P = [
    16, 7,  20, 21,
    29, 12, 28, 17,
    1,  15, 23, 26,
    5,  18, 31, 10,
    2,  8,  24, 14,
    32, 27, 3,  9,
    19, 13, 30, 6,
    22, 11, 4,  25
]

IP = [
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9,  1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
]



E = [
    32, 1,  2,  3,  4,  5,
    4,  5,  6,  7,  8,  9,
    8,  9,  10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32, 1
]

sboxs = {
    1: [
        [14,    4,  13, 1,  2,  15, 11, 8,  3,  10, 6,  12, 5,  9,  0,  7],
        [0, 15, 7,  4,  14, 2,  13, 1,  10, 6,  12, 11, 9,  5,  3,  8],
        [4, 1,  14, 8,  13, 6,  2,  11, 15, 12, 9,  7,  3,  10, 5,  0],
        [15,    12, 8,  2,  4,  9,  1,  7,  5,  11, 3,  14, 10, 0,  6,  13]
    ],
    2: [
        [15,    1,  8,  14, 6,  11, 3,  4,  9,  7,  2,  13, 12, 0,  5,  10],
        [3, 13, 4,  7,  15, 2,  8,  14, 12, 0,  1,  10, 6,  9,  11, 5],
        [0, 14, 7,  11, 10, 4,  13, 1,  5,  8,  12, 6,  9,  3,  2,  15],
        [13,    8,  10, 1,  3,  15, 4,  2,  11, 6,  7,  12, 0,  5,  14, 9]
    ],
    3: [
```

```
        [10,  0,  9,  14, 6,  3,  15, 5,  1,  13, 12, 7,  11, 4,  2,  8],
        [13,  7,  0,  9,  3,  4,  6,  10, 2,  8,  5,  14, 12, 11, 15, 1],
        [13,  6,  4,  9,  8,  15, 3,  0,  11, 1,  2,  12, 5,  10, 14, 7],
        [1, 10, 13, 0,  6,  9,  8,  7,  4,  15, 14, 3,  11, 5,  2,  12]
    ],
    4: [
        [7, 13, 14, 3,  0,  6,  9,  10, 1,  2,  8,  5,  11, 12, 4,  15],
        [13,  8,  11, 5,  6,  15, 0,  3,  4,  7,  2,  12, 1,  10, 14, 9],
        [10,  6,  9,  0,  12, 11, 7,  13, 15, 1,  3,  14, 5,  2,  8,  4],
        [3, 15, 0,  6,  10, 1,  13, 8,  9,  4,  5,  11, 12, 7,  2,  14]
    ],
    5: [
        [2, 12, 4,  1,  7,  10, 11, 6,  8,  5,  3,  15, 13, 0,  14, 9],
        [14,  11, 2,  12, 4,  7,  13, 1,  5,  0,  15, 10, 3,  9,  8,  6],
        [4, 2,  1,  11, 10, 13, 7,  8,  15, 9,  12, 5,  6,  3,  0,  14],
        [11,  8,  12, 7,  1,  14, 2,  13, 6,  15, 0,  9,  10, 4,  5,  3]
    ],
    6: [
        [12,  1,  10, 15, 9,  2,  6,  8,  0,  13, 3,  4,  14, 7,  5,  11],
        [10,  15, 4,  2,  7,  12, 9,  5,  6,  1,  13, 14, 0,  11, 3,  8],
        [9, 14, 15, 5,  2,  8,  12, 3,  7,  0,  4,  10, 1,  13, 11, 6],
        [4, 3,  2,  12, 9,  5,  15, 10, 11, 14, 1,  7,  6,  0,  8,  13]
    ],
    7: [
        [4, 11, 2,  14, 15, 0,  8,  13, 3,  12, 9,  7,  5,  10, 6,  1],
        [13,  0,  11, 7,  4,  9,  1,  10, 14, 3,  5,  12, 2,  15, 8,  6],
        [1, 4,  11, 13, 12, 3,  7,  14, 10, 15, 6,  8,  0,  5,  9,  2],
        [6, 11, 13, 8,  1,  4,  10, 7,  9,  5,  0,  15, 14, 2,  3,  12]
    ],
    8: [
        [13,  2,  8,  4,  6,  15, 11, 1,  10, 9,  3,  14, 5,  0,  12, 7],
        [1, 15, 13, 8,  10, 3,  7,  4,  12, 5,  6,  11, 0,  14, 9,  2],
        [7, 11, 4,  1,  9,  12, 14, 2,  0,  6,  10, 13, 15, 3,  5,  8],
        [2, 1,  14, 7,  4,  10, 8,  13, 15, 12, 9,  0,  3,  5,  6,  11]
    ]
}

def permutation(bits, perm, perm_taille):
    """
        permute la chaine de caract res bits suivant le tableau de permutation perm, de
            taille perm_taille
    """
    res = []
    for i in range(0, perm_taille):
        p = perm[i]
        res.append(bits[p-1])
    return ''.join(res)

def inverse_perm(perm, taille_finale):
    """
        Inverse une permutation perm
    """
    res = []
    for i in range(1, len(perm)+1):
```

```
        j = 0
        while (j < len(perm)) and (perm[j] != i):
            j+=1
        if len(res) < taille_finale:
            res.append(j+1)
    return res

def expansion(R_im1):
    """
        Renvoie Expansion(R_im1), une des op rations faites dans la fonction de tour du
            sch ma de Feistel
    """
    return permutation(R_im1, E, 48)


IPinverse = inverse_perm(IP, 64)
Pinverse = inverse_perm(P, 32)

clair = ""
chiffre_juste = ""
chiffres_faux= []
with open("donnees.txt", "r") as f:
    line = f.readline()
    while 'faux' not in line:
        if 'clair' in line:
            clair = int(line.split(" : ")[1].strip().replace(" ", ""), 16)
        elif 'juste' in line:
            chiffre_juste = int(line.split(" : ")[1].strip().replace(" ", ""), 16)
        line = f.readline()
    line = f.readline()
    while line != '':
        chiffres_faux.append(int(line.strip().replace(" ", ""), 16))
        line = f.readline()

chiffre_juste = bin(chiffre_juste)[2:].zfill(64)
clair = bin(clair)[2:].zfill(64)

chiffres_faux = list(map(lambda x: bin(x)[2:].zfill(64), chiffres_faux))
print("chiffr  juste = ", chiffre_juste)
print("clair = ", clair)

ant = permutation(chiffre_juste, IP, 64)
L16 = ant[0:32]
R16 = ant[32:]
print("L16 = ", L16)
print("R16 = ", R16)

"""
chiffr  juste =  0011111001000111101001000110101011111010000001011110001001111110
clair =  0011010000100110001111010010101001111000101111100011100001101010 1
L16 =  11011010100100011010011100100010
R16 =  01010100110111011001100111011011

"""
```

```
156
157 R15_exp = expansion(R16)
158
159 def xor(a, b):
160     """
161         retourne a XOR b
162          o   a et b sont des chaines de caracteres representant des nombres binaires
163     """
164     return bin(int(a, 2) ^ int(b, 2))[2:].zfill(len(a))
165
166 def trouver_fautes():
167     """
168         fonctions retrouvant comment l'erreur sur un bit de R15 se propage apr s
                expansion et quelle(s)
169         sbox a en ant c dant les portions faut es.
170         Renvoie un dictionnaire de la forme {numero_de_la_sbox : liste de chiffr s faux}
171     """
172     faux_par_sbox = {i: [] for i in range(1, 9)}
173
174     R16_juste = permutation(chiffre_juste, IP, 64)[32:]
175
176     for faux in chiffres_faux:
177         R16_faux = permutation(faux, IP, 64)[32:]
178         delta = xor(R16_faux, R16_juste)
179         exp = expansion(delta)
180
181         i = 0
182         for j in range(0, 48, 6):
183             if exp[j:j+6] != '000000':
184                 faux_par_sbox[i+1].append(faux)
185             i+=1
186
187     return faux_par_sbox
188
189 print(trouver_fautes())
190
191 """
192 {
193     1: ['00111100010000101010010000101110111110100000010111100010011011110',
194         '00111110010001101010010011111010111110100000010110100010011011110',
195         '00111110010000111010010001111010011110110000010111100010011011010',
196         '00111110010000101010010001101010111110101000010111100010011011110',
197         '00111110010000101010010001101010111110100000010101100010011111010',
198         '00111110010100101010010001101110111110100000010111100010111010010'],
199     2: ['00101010010001111110010001111010111110100000010111100110010111110',
200         '10101010010001111010010101101010111110110000010110100010011111110',
201         '00111110110001111010010001101010111110110000010110100110011111110',
202         '00111110010001110010010001111010111110110000010110100010011111110',
203         '00111110010001101010010011111010111110100000010110100010011011110',
204         '00111110010000111010010001111010011110110000010111100010011010010'],
205     3: ['00101110010001111110010001001010111111100000010111110011001111110',
206         '00101110010001111110010001101010110111100000010111100011011111110',
207         '00111010010001111110010001101010111110100010010111100010011111110',
208         '00101010010001111010010001101010111110100000010111000010011111110',
```

```
        '00101010010001111111001000111101011111010000001011110011001011110',
        '10101010010001111101001010110101011111011000001011010001001111110'],
    4: ['01111110010001111011010001101010111110100100010111110011011110110',
        '01011110010001111011010001101010111111100000010111100011011111111',
        '00111110011001111010010001101010111111100000010111100011001111110',
        '00111110010001111000010001101010111110100000010111100011001111110',
        '00101110010001111111001000100101011111110000001011110011001111110',
        '00101110010001111111001000110101011011110000001011110001101111110'],
    5: ['00111110010001111011010001100011111110100001000111100011001111111',
        '01111110010001111011010001101011111100100001010111100011001111111',
        '01111110010001111011010001101010111110100000110111100011001111110',
        '01111110010001111011010001101010111110100100010111110101001111111',
        '01111110010001111011010001101010111110100100010111110011011110110',
        '01011110010001111011010001101010111111100000010111100011011111111'],
    6: ['00111111000001111010000001101011101110100001010111100011001111100',
        '00110111000001111010000001101010101010100001000111100011001111110',
        '00111110010011111010010001101010111110100001000111100011001111110',
        '00111110010001111010110001101011111110100001010111100011001111110',
        '00111110010001111011010001100011111110100001000111100011001111111',
         '01111110010001111011010001101011111100100001010111100011001111111'],
    7: ['00111111010101111010010001101100111010100000010011100011001111110',
        '00111111010101111010000000101110111010000000010011100011001111110',
        '00111111010001111010000001101010111110100000011111100011001111110',
        '00111111000001111010000001101010111010100000010111100000001111110',
        '00111111000001111010000001101011101110100001010111100011001111100',
        '00110111000001111010000001101010101010100001000111100011001111110'],
    8: ['00111100010000101010010000101110111110100000010111100011001101110',
        '00111110010101011010010001101010111110100000010011100011001111110',
        '00111110010101111010011000101110111110100000010011100011001111110',
        '00111111010101111010010001101100111010100000010011100011001111110',
        '00111111010101111010000000101110111010000000010011100011001111110',
        '00111110010100101010010001101110111110100000010111100011011101010']}
"""


import itertools
def generer_combinaisons_bits(n):
    """
    g n re l'ensemble des chaines de caract res repr sentant un nombre binaire de n
        bits
    """
    return [''.join(seq) for seq in itertools.product("01", repeat=n)]

K16s = generer_combinaisons_bits(6)

def appliquer_sbox(exp_R, s):
    """
        Applique a exp_R la sbox num ro s
    """
    b0 = str(exp_R[0])
    b1 = str(exp_R[1])
    b2 = str(exp_R[2])
    b3 = str(exp_R[3])
    b4 = str(exp_R[4])
```

```
    b5 = str(exp_R[5])
    ligne = int(b0+b5, 2)
    colonne = int(b1+b2+b3+b4, 2)
    return bin((sboxs[s][ligne])[colonne])[2:].zfill(4)

def inter(Ls):
    """
        Renvoie l'  lment   i qui se trouve dans toutes les listes stock s dans la liste
             de listes Ls
    """
    res = set(Ls[0])
    for e in Ls[1:]:
        res.intersection_update(e)
    return list(res)[0]

def attaquer_sbox():
    """
        Fonction de recherche exhaustive sur les sboxs.
        Son fonctionnement est d crit dans le rapport
    """
    candidat_par_sbox = {i: [] for i in range(1, 9)}
    res = []
    faux_par_sbox = trouver_fautes()

    for sbox, faux in faux_par_sbox.items():
        for f in faux:
            ant_faux = permutation(f, IP, 64)
            #L16_faux = liste_L16_faux[faux]
            L16_faux, R16_faux = ant_faux[0:32], ant_faux[32:]
            R15_faux = R16_faux
            R15_faux_exp = expansion(R15_faux)
            delta = xor(L16, L16_faux)
            y = permutation(delta, Pinverse, 32)
            sols = []
            for k16 in K16s:
                for b in range(0, 8):
                    ant_juste = xor(k16, R15_exp[b*6:(b+1)*6])
                    ant_faux = xor(k16, R15_faux_exp[b*6:(b+1)*6])
                    sortie_sbox_juste = appliquer_sbox(ant_juste, sbox)
                    sortie_sbox_faux = appliquer_sbox(ant_faux, sbox)
                    comp = xor(sortie_sbox_juste, sortie_sbox_faux)
                    if (comp == y[b*4:(b+1)*4]) and (comp != '0000'):
                        sols.append(int(k16, 2))
            candidat_par_sbox[sbox].append(sols)
        solution = inter(candidat_par_sbox[sbox])
        res.append(bin(solution)[2:].zfill(6))
    return res

print(attaquer_sbox())

"""
['011100',
 '000100',
 '010011',
```

```
 '000111',
 '110110',
 '101010',
 '101111',
 '000011']
"""

def retrouver_K16(morceaux_k16):
    """
        On reconstitue K_{16} par simple concat nation des portions trouv s avec l'
            attaque sur les sboxs
    """
    k16 = ""
    for m in morceaux_k16:
        k16 += m
    return k16

k16 = retrouver_K16(attaquer_sbox())
print("Valeur de K_{16} :")
print("--- en binaire --| ", k16)
print("---- en hexa ----| ",hex(int(k16, 2)))


"""
Valeur de K_{16} :
--- en binaire --|  011100000100010011000111110110101010101111000011
---- en hexa ----|  0x7044c7daabc3
"""

PC1 = [
    57, 49, 41, 33, 25, 17, 9,
    1,  58, 50, 42, 34, 26, 18,
    10, 2,  59, 51, 43, 35, 27,
    19, 11, 3,  60, 52, 44, 36,
    63, 55, 47, 39, 31, 23, 15,
    7,  62, 54, 46, 38, 30, 22,
    14, 6,  61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4
]

PC2 = [
    14, 17, 11, 24, 1,  5,
    3,  28, 15, 6,  21, 10,
    23, 19, 12, 4,  26, 8,
    16, 7,  27, 20, 13, 2,
    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32
]

def trouver_emplacements_manquants_PC(PC):
    n = 0
    if PC == PC2:
```

```
        n = 56
    if PC == PC1:
        n = 64
    L = [i for i in range(1, n+1)]
    for e in PC:
        if e in L:
            L.remove(e)
    return L

print("Bits manquant pour PC1 : ", trouver_emplacements_manquants_PC(PC1))
print("Bits manquant pour PC2 : ",trouver_emplacements_manquants_PC(PC2))

"""
Bits manquant pour PC1 :  [8, 16, 24, 32, 40, 48, 56, 64]
Bits manquant pour PC2 :  [9, 18, 22, 25, 35, 38, 43, 54]

"""

"""
    En se servant des informations sur les bits manquants, on peut
    reconstituer les inverses. On d cide arbitrairement de mettre
    a -1 les permutations des bits manquants.
"""
PC1inverse = [
    8,  16, 24, 56, 52, 44,
    36,-1,  7,  15, 23, 55,
    51, 43, 35,-1,  6,  14,
    22, 54, 50, 42, 34,-1,
    5,  13, 21, 53, 49, 41,
    33,-1,  4,  12, 20, 28,
    48, 40, 32,-1,  3,  11,
    19, 27, 47, 39, 31,-1,
    2,  10, 18, 26, 46, 38,
    30,-1,  1,  9,  17, 25,
    45, 37, 29, -1
]


PC2inverse = [
    5,  24, 7,  16, 6,  10,
    20, 18,-1,  12, 3,  15,
    23, 1,  9,  19, 2, -1,
    14, 22, 11,-1,  13, 4,
   -1,  17, 21, 8,  47, 31,
    27, 48, 35, 41,-1,  46,
    28,-1,  39, 32, 25, 44,
   -1,  37, 34, 43, 29, 36,
    38, 45, 33, 26, 42,-1,
    30,  40
]

def appliquer_PC2inverse():
    """
        Fonction qui applique PC_2^{-1}
```

```
    """
    res = []
    for b in PC2inverse:
        if b != -1:
            res.append(k16[b-1])
        else:
            res.append('*')
    return ''.join(res)

def appliquer_PC1inverse(ktmp):
    """
        Fonction qui applique PC_1^{-1}
    """
    res = []
    for b in PC1inverse:
        if ktmp[b-1] != '*':
            res.append(ktmp[b-1])
        else:
            res.append('*')
    return ''.join(res)

k_tmp = appliquer_PC1inverse(appliquer_PC2inverse())
print("Valeur de K_{tmp} :")
print("--- en binaire --| ", k_tmp)

"""
Valeur de K_{tmp} :
--- en binaire --|  1011110000001**010**00100101011000100010011011001 0*10*100*1*0110

"""

import requests
from bs4 import BeautifulSoup

def DES_en_ligne(k_test):
    """
        Fonction qui verifie si la cl  a tester K_test est la bonne
        en verifiant qu'elle permet bien de d chiffrer le chiffr  juste en obtenant le
            clair
        On se sert ici de https://emvlab.org/descalc/ comme calculatrice DES.
        En effet, si l'on va sur le site et qu'on l'utilise, on se rend compte que les
            donn es que l'on rentre
        sont transmises au serveur directement via l'URL sous la forme:
        https://emvlab.org/descalc/key=[CLE]'&iv=0000000000000000&input=[CLAIR]'&mode=ecb
            &action=Encrypt&output='
        On peut alors se servir du package requests de Python pour faire au serveur une
            requ te URL
        avec les donn es de notre probleme
        On inspecte ensuite le code HTML du site et on voit que la zone de texte est
            balis e par
        <textarea cols="48" id="output" name="output" rows="10"></textarea></p>
        On peut donc se servir du package BeautifulSoup pour lire la r ponse dans la
            textearea "Output data"
        (avec comme id "output") du site
```

```
    """

    url = "https://emvlab.org/descalc/?"
    url += 'key=' + k_test + '&iv=0000000000000000&input=' + hex(int(clair,2))[2:] + '&
        mode=ecb&action=Encrypt&output='

    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')

    if soup.find(id='output').getText() == hex(int(chiffre_juste,2))[2:].upper():
        return True
    return False

def positions_manquantes():
    """
        Trouve les positions manquantes dans la cl
        Consiste en un simple parcourt de cha ne de caract res
    """
    res = []
    for i in range(0, len(k_tmp)):
        if k_tmp[i] == '*':
            res.append(i)
    return res

def completer_k():
    """
        Attaque par recherche exhaustive pour completer la cl  K en retrouvant les 8
            bits manquants
    """
    positions_manq = positions_manquantes()
    possibilites = generer_combinaisons_bits(8)
    for combi in possibilites:
        #if combi != '00000000':
        k_test = []
        k_test[:0] = k_tmp
        for i, b in enumerate(combi):
            k_test[positions_manq[i]] = b
        k_test = ''.join(k_test)
        if(DES_en_ligne(hex(int(k_test,2))[2:].upper())):
            return k_test, hex(int(k_test,2))[2:]

def parite(k):
    """
        Reecriture des octets composants la cl  de telle sorte a ce que chacun ait un
            nombre impair de 1.
         Il suffit de compter le nombre de 1 et de 0 a chaque octet et de r  crire le
             dernier bit
    """
    k64 = ""
    for i in range(0, 64, 8):
        octet = []
        octet[:0] = k[i:i+8]
        if octet.count('1') % 2 == 0:
            octet[-1] = '1'
```

```
        else:
            octet[-1] = '0'
        k64 += ''.join(octet)
    return k64

k, k_hex = completer_k()
K = parite(k)
if(DES_en_ligne(hex(int(K, 2))[2:].upper())):
    print("La  c l   finale est ")
    print("--- en binaire -- |", K)
    print("---- en hexa ---- |", hex(int(K, 2))[2:].upper())


"""
La  c l   finale est
--- en binaire -- | 1011110000001000100000110101011100100011011011011001001000100110
---- en hexa ---- | BC088357236D9226
"""
```